U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

PhD

3rd CYCLE

FCUP
2026

U.PORTO

Theoretical and Practical Security of Passwordless Authentication

Luís Esquível

FC

Theoretical and Practical Security of Passwordless Authentication

Luís Esquível
Doctoral Program in Computer Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2026

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

PhD

Theoretical and Practical Security of Passwordless Authentication

Luís Esquível

Thesis carried out as part of the Doctoral Program in Computer Science

Department of Computer Science

2026

Supervisor
Manuel Barbosa, Associate Professor, Faculty of Sciences of the University of Porto



Acknowledgements

This work was financially supported by Fundação para a Ciência e a Tecnologia (FCT), through the PhD grant 2021.07225.BD, co-funded by the FSE (Fundo Social Europeu) within the scope of the PDQI (Programa Demografia, Qualificações e Inclusão).

First and foremost, I am deeply grateful to my supervisor, Manuel Barbosa, for his mentorship over the last four years. I am especially thankful for his willingness to supervise my PhD when I first reached out to him, and for his remarkable patience in managing my constant stream of questions and the various hurdles we encountered along the way. It has, without a doubt, been an absolute privilege to have worked with and learned from you.

I would also like to extend my gratitude to all my research colleagues, whose technical expertise and collaborative spirit were instrumental to the success of this work.

To my family, I owe an enormous thank you for your unwavering support throughout this four-year adventure. I truly doubt that I would have made the decision to embark on this journey, or have the commitment to complete it, without your encouragement.

To my friends, thank you for the much needed distractions throughout these years. Your friendship was a fantastic remedy for the stresses of research, and I am grateful for all the laughter and support you have given me.

And last, but certainly not least, an extra special thank you to my girlfriend, Isabel. You have been the strongest pillar in my life throughout this entire process, having stood by me with unwavering support since my master's thesis, particularly during those sometimes months-long processes where I was perhaps a bit too consumed by this work. You have been there since the very beginning of this long journey, and your presence was paramount in maintaining my focus and motivation. This thesis is as much mine as it is yours!

Resumo

A rápida adoção de métodos de autenticação *passwordless*, em grande medida impulsionada pelo protocolo FIDO2, representa uma mudança fundamental na forma como é feita a autenticação *online*. Os mecanismos tradicionais, baseados em palavras-passe, permanecem inerentemente vulneráveis a uma grande quantidade de ameaças, incluindo *phishing* e *data breaches*. Ao eliminar completamente o uso de palavras-passe, o protocolo FIDO2 procura oferecer resistência às vulnerabilidades sistémicas que têm comprometido os sistemas de autenticação ao longo de décadas. No entanto, a segurança deste protocolo não é absoluta, e embora algum trabalho de investigação já tenha sido efetuado na análise das suas garantias de segurança, ainda persistem certos elementos na arquitetura do protocolo que podem ser explorados para comprometer a sua segurança.

Neste trabalho, examinamos esses elementos e as suas implicações práticas na segurança do FIDO2, e de seguida apresentamos, formalmente, medidas concretas para as mitigar. Começamos por demonstrar que determinadas escolhas no *design* do protocolo CTAP2, um dos componentes do FIDO2, criam vulnerabilidades possíveis de serem exploradas, através da implementação de três ataques que comprometem a segurança de uma conta de utilizador, e mostramos que um adversário determinado pode executar estes ataques de uma forma realista. De seguida, focamo-nos na análise formal do FIDO2, baseando-nos em trabalhos prévios [1, 2]. Introduzimos duas modificações concretas ao CTAP2 que provamos que o torna resistente a estes ataques, e demonstramos que o FIDO2, com as nossas alterações, alcança garantias de segurança mais fortes. Por fim, integramos as modificações propostas em implementações comerciais de FIDO2, demonstrando que são simples de implementar, retrocompatíveis com o protocolo atual, e apresentam um impacto mínimo no desempenho.

Em suma, os nossos resultados mostram que o FIDO2 pode ser reforçado para proporcionar garantias de segurança ainda mais robustas, com pouco ou nenhum custo perceptível em cenários de utilização real.

Palavras-chave: Criptografia Pós-Quântica, Criptografia, Segurança, Ciência de Computadores, FIDO2, CTAP2, WebAuthn, Provas de segurança, Passwords, Passwordless, Autenticação, Ataques, Vulnerabilidades

Abstract

The rapid adoption of passwordless authentication, largely driven by the FIDO2 standard, represents a fundamental shift in online authentication. Traditional password-based mechanisms remain inherently vulnerable to a myriad of threats, including phishing and database breaches. By completely eliminating the reliance on passwords, FIDO2 aims to provide resistance against the systemic vulnerabilities that have compromised authentication systems for decades. However, the security of FIDO2 is not absolute, and while existing research has explored its security guarantees, some flaws in the protocol's design that can be exploited to undermine the protocol's security still persist.

In this work, we examine these flaws and their practical security implications, and then introduce formal countermeasures to address them. We begin by demonstrating that certain protocol design choices in CTAP2, one of the FIDO2 component protocols, create exploitable vulnerabilities, through the implementation of three proof-of-concept attacks that lead to the compromise of user account security, and show that a determined adversary can realistically carry out these attacks. We then shift our focus to the formal analysis of FIDO2, based on prior work from [1, 2], where we introduce two concrete modifications to CTAP2 that make it provably resistant to these attacks. We show that FIDO2, with our modifications, achieves stronger security guarantees. Finally, we integrate our proposed modifications in commercial implementations of FIDO2, demonstrating that they are simple to implement, have complete backwards compatibility with the current standard, and have minimal performance overhead.

Our findings demonstrate that FIDO2 can be hardened to deliver even stronger security guarantees, while incurring little to no perceivable cost in real-world usage scenarios.

Keywords: Post-Quantum Cryptography, Cryptography, Security, Computer Science, FIDO2, CTAP2, WebAuthn, Provable Security, Passwords, Passwordless, Authentication, Attacks, Vulnerabilities

Table of Contents

List of Figures vi

1. Introduction 1
   1.1. Context and motivation 1
   1.2. Prior work on the security of FIDO2 4
   1.3. Work conducted and associated publications 5
   1.4. Structure of this thesis 10
2. Preliminaries 11
   2.1. Notation 11
   2.2. Security Definitions 11
   2.3. The ML-KEM scheme 15
   2.4. Password Authenticated Key Exchange (PAKE) Protocols 16
3. The FIDO2 Protocol 21
   3.1. History of FIDO 21
   3.2. Protocol overview 22
   3.3. The WebAuthn protocol 25
   3.4. The CTAP2 protocol 31
   3.5. Summary 39
4. Practical attacks on FIDO2 40
   4.1. Design choices in CTAP2 41
   4.2. Impersonation Attack 42
   4.3. Rogue Key Attack 48
   4.4. Double registration attack 51
   4.5. Implementation and deployment strategy 52
   4.6. Results 59
5. FIDO2 security against rogue key attacks 64
   5.1. Background 64
   5.2. Our results 70
   5.3. CTAP 2.1 and CTAP 2.1+ security analysis 72
   5.4. PIA and WebAuthn 95
   5.5. PIA+mPACA composed security analysis 103
6. FIDO2 security against active binding attacks 116
   6.1. Background 117
   6.2. Our results 119
   6.3. Updated authentication model 119

6.4. sCTAP 2.1+ as an mPACA protocol 122
6.5. Authentication security of sCTAP 2.1+ 130
6.6. Instantiating the PAKE protocol in sCTAP 2.1+ 134
7. Experimental results 139
7.1. Implementation of CTAP 2.1+ 140
7.2. Implementation of sCTAP 2.1+ 141
7.3. Performance results 146
8. Conclusions and future work 152
8.1. Future work 153

Bibliography 155
A. The WebAuthn Protocol 163
A.1. Registration 163
A.2. Attestation 171
A.3. Authentication 177
B. The CTAP2 protocol 181
B.1. Low-level encoding and transport 181
B.2. Authenticator API 184
B.3. Differences between versions 201
C. CTAP 2.1 proof for PIN/UV Auth Protocol 1 203
D. IND-1$PA-LHPC Proof for CBC 204
E. CTAP 2.1+ SUF-t Authentication Proof 214
F. sCTAP 2.1+ SUF Authentication Proof 251

List of Figures

2.1. The SUF-CMA security experiment. 12
2.2. The IND-1CPA security experiment. 12
2.3. The IND-1$PA security experiment. 13
2.4. The IND-1$PA-LPC security experiment. 14
2.5. The IND-1$PA-LHPC security experiment. 14
2.6. The IND-1$PA-LHPC-M security experiment. 15
2.7. The pfs, ma and sid-coll security experiments. 19

3.1. The simplified FIDO2 protocol flow. 23
3.2. The simplified WebAuthn protocol. 27
3.3. The simplified CTAP2 protocol. 33

4.1. Simplified diagram of the man-in-the-middle attack. 43
4.2. The *impersonation attack*. 45
4.3. The *rogue key attack*. 50
4.4. Simple example of function hooks on the *read* and *write* system calls. 54

5.1. (m)PACA authentication security experiments and winning conditions. 77
5.2. Oracles for the (m)PACA security experiments. 78
5.3. CTAP 2.1 as a PACA protocol and CTAP 2.1+ as an mPACA protocol. 81
5.4. CTAP 2.1 and CTAP 2.1+ protocol functions. 82
5.5. The PIN/UV Auth Protocol 2. 84
5.6. The PIN/UV Auth Protocol 1. 84
5.7. The WebAuthn protocol. 97
5.8. The WebAuthn protocol functions. 99
5.9. Security experiment for the PIA passwordless authentication security experiment. 102
5.10. Oracles of the ua security experiment for the composed model. 106
5.11. The ua security experiment and winning conditions for PIA+mPACA. 107
5.12. First hop of the composed model proof. 110
5.13. Second hop of the composed model proof. 111
5.14. The rogue key attack (against a registration run) in Basic attestation. 113

6.1. mPACA SUF experiment winning conditions. 120
6.2. Oracles for mPACA SUF security experiment. 121
6.3. sCTAP 2.1+ as an mPACA protocol. 123
6.4. sCTAP 2.1+ protocol functions. 125
6.5. The sCTAP 2.1+ PIN/UV Auth Protocol 3. 127

7.1. Nitrokey firmware binary sizes, in bytes, for each configuration tested. 149
7.2. Performance benchmarks for Bind using PIN/UV Auth Protocol 1, 3 and 4. 150

A.1. The PublicKeyCredentialCreationOptions dictionary from the WebAuthn specification. 164
A.2. The CollectedClientData dictionary from the WebAuthn specification. 167
A.3. Structure of the Attestation Object. 173
A.4. Structure of the Attestation Statement. 173
A.5. Structure of the Authenticator Data. 173

A.6. The PublicKeyCredentialRequestOptions dictionary from the WebAuthn specification.178

B.1. Initialization packet structure. 183
B.2. Continuation packet structure. 183
B.3. Key differences between the different versions of CTAP2. 202

1. Introduction

The security of online authentication has never been more important. Password-based authentication has been the norm for decades, yet its limitations and inherent insecurities are driving growing interest in passwordless alternatives. Among them, FIDO2 is emerging as the leading solution, with reported advantages of strong security guarantees, user-friendly experience, and resistance to the most common attacks against password-based systems [3–5]. Released in 2018 by the FIDO Alliance [6], the protocol has entered mainstream use, with major organizations including Microsoft, Apple, and Google (all members of the Alliance) contributing to its increased adoption by supporting *passkeys*[1] as an alternative primary form of authentication.

On the other hand, prior to this work, FIDO2 had attracted relatively little attention in the cryptography literature, with only a handful of publications [1, 2, 7, 8] dedicated to formally analyse its security (and privacy) guarantees and study their practical implications.

This work examines FIDO2 in depth, building on top of the existing literature. We first showcase that some of the protocol's design choices are exploitable in practice, by implementing proof-of-concept attacks that break the protocol's expected security guarantees. We then propose fixes to FIDO2, and prove that they strengthen the protocol by making it resistant to these attacks. Finally, we implement and benchmark the fixes on commercial implementations of FIDO2, where we show that not only are they fairly easy to add to existing codebases, they also incur small (essentially unnoticeable) performance penalties.

This Chapter is organized in 4 parts. Section 1.1 describes the main motivation for our work on FIDO2, where we contextualize the current digital landscape regarding online authentication, which motivated our work on this protocol. Section 1.2 sets the background for our work, by briefly explaining how prior publications on the theoretical security of FIDO2 paved the way for our contributions. Section 1.3 provides an overview of our work, and identifies associated publications. Finally, Section 1.4 describes the structure for the rest of the document.

1.1. Context and motivation

For most people, passwords remain the default way to authenticate online. They are easy to use, require no extra hardware, prior configuration or technical expertise, and

[1] See subsection 3.2.4 of Chapter 3.

can be used anywhere, at any time. To offer this level of convenience, passwords must be human-memorable: a relatively simple string of characters that a person can realistically remember without having to write it down. For this reason, the use of passwords introduces several security limitations.

Shortcomings of password-based authentication In the current digital landscape, the average user maintains a relatively high number of online accounts[2]. Maintaining a unique password for each account that is both memorable and resistant to guessing or brute-force attacks has, therefore, become increasingly harder, leading many to reuse the same password (or simple variations) across multiple services.

Reusing passwords is risky, as a leak from one account may compromise the security of other accounts sharing the same password. Then, even if someone maintains a unique, strong password for each account (which is facilitated by modern tools such as password managers), its security always depends on how it is managed by each service. A strong password may offer little protection if stored insecurely.

Phishing attacks are also a serious threat to the security of online accounts. Users with strong, unique passwords which are correctly handled by a service can still end up compromising their accounts by being fooled by an email and ending up inputting their passwords on a fake website controlled by an attacker.

MFA authentication Multifactor authentication (MFA) mitigates some of the shortcomings associated with password-based authentication, by requesting one or more additional forms of authentication from a user. In addition to providing a password, the owner of an account may, for example, also provide an ephemeral code received via SMS, or approve the sign in through its phone using a specific application.

MFA authentication offers additional protection to using just a password, but the latter still plays a role in the authentication process. As such, it can still be insecurely stored in a service's database and be leaked, or be stolen through a phishing attack. The latter can also be performed against *most* MFA mechanisms, through similar procedures where an attacker convinces a user to input a received or generated code on a fake website it controls.

[2]Over 100 accounts according to a 2024 survey from Nordpass[3]. However, this survey considers all types of accounts, including those that are rarely used or "forgotten". A different survey from Bitdefender puts the number of active accounts between 3 and 10.

[3]https://nordpass.com/blog/how-many-passwords-does-average-person-have/

Passwordless authentication and FIDO2 Passwordless authentication addresses the shortcomings of passwords by completely getting rid of them in favor of other types of authentication. FIDO2 is a widely adopted passwordless authentication protocol that leverages public key cryptography, using a trusted device called an *authenticator* or *token*, to register and authenticate a user to a service. It is also phishing-resistant, since a registration or authentication reveals no sensitive information like passwords or other secret cryptographic material, and because an attacker cannot gain access to a user's account on a website by performing a man-in-the-middle attack[4] where it redirects a legitimate authentication request to the user's authenticator through the user's browser.

While we mention FIDO2 as a protocol, in reality it is a combination of two protocols: WebAuthn [9] and CTAP2 [10]. WebAuthn is the protocol that governs the registration and authentication of an authenticator with an online service, called a *relying party*. This interaction happens through a browser or another client-side application, called a *client* in FIDO2, which acts as an intermediary between the relying party and the authenticator. CTAP2 is a low-level access control protocol that governs the communications between a client and an authenticator. In the context of FIDO2 passwordless authentication, authenticators are generally protected against unauthorized use. One of the primary responsibilities of CTAP2 is to ensure that a client application can access an authenticator only with the explicit consent of the authenticator's owner (e.g. by collecting a PIN).

FIDO2 adoption Since its introduction by the FIDO Alliance, FIDO2 has become the *de-facto* standard for passwordless authentication. In 2022, when we began our work on FIDO2, global awareness of passkeys by consumers was at around 39% [11]. In 2025, this number grew to 74%, with 54% of consumers considering passkeys more convenient than passwords, and 53% considering them more secure [12]. At the same time, in 2025, 48% out of the top 100 websites now support passkeys as a form of authentication, up from 20% in 2022 [13], with over 15 billion online accounts now eligible for using passkeys [11, 14].

In April 2025, the FIDO Alliance launched the Passkey Pledge [14, 15], a voluntary commitment for organisations to work towards the goal of making FIDO2 passwordless authentication the default authentication mechanism for online accounts. At the time of writing, over 200 organisations, including Apple, Microsoft, Google and Amazon, have signed this pledge.

[4] Not to be confused with the man-in-the-middle attack on CTAP2 described in Chapter 4.

Adoption of passkeys in the enterprise world has also been substantial. A 2025 FIDO Alliance survey [16] found that 87% of companies have already deployed, or are on the verge of deploying, passkeys for employee authentication. Among them, 47% indicated that they are enrolling both device-bound passkeys and synchronized (cross-device) passkeys.

Motivation for this work We began our work on FIDO2 in 2022. At the time, the protocol was already gaining significant traction, but had received limited scrutiny in the cryptographic literature. In particular, to the best of our knowledge, only a single publication [1] had performed an in-depth formal security analysis of FIDO2[5]. Given the protocol's potential to strengthen the security of online authentication, and its significant industry backing from the FIDO Alliance, we concluded that a comprehensive investigation of the protocol's security, spanning both its formal security guarantees and their practical implications, would present a highly compelling research opportunity with significant value to the cryptographic literature.

1.2. Prior work on the security of FIDO2

As previously mentioned, the authors in [1] performed the first provable security analysis of FIDO2. In this work, they pointed out that, due to some design choices, FIDO2 cannot be proven secure against an attacker that can actively interfere with the communications between the client and a user's authenticator, i.e. an attacker that can actively interfere with CTAP2 communications. Such an attacker can break the protocol's security guarantees by actively attacking CTAP2 through a *man-in-the-middle attack* on an unauthenticated Diffie-Hellman key exchange, which can be leveraged to gain unauthorized access to the user's authenticator.

These findings were confirmed by a new publication [2] that expanded the formal security analysis to cover a new version of FIDO2, using a more detailed security model and protocol syntax that was based on [1].

Assumptions and trust model In order to prove FIDO2 secure, both analyses discard the man-in-the-middle attack by assuming that an attacker does not actively interfere with the CTAP2 key exchange. Additionally, the analysis from [2] assumes the whole registration procedure (i.e., account creation) to be authenticated, while the analysis from [1] assumes

[5] There were previous publications that formally analysed older versions of FIDO [17], as well as WebAuthn [7, 8]. However, this publication was the first to formally analyse the security guarantees of FIDO2 as a composition of WebAuthn and CTAP2.

the relying party on which the user is registering to know *a priori* the exact authenticator that will be used. As it turns out, these assumptions are necessary to avoid an additional *rogue key attack* on CTAP2 (although this is not explicitly mentioned in either analysis), where an adversary can hijack a user's registration by delivering the final message of the registration flow to the client from its own authenticator, allowing it to register its own authenticator on a user's account, instead of the user's own authenticator.

Naturally, a question that may follow from these analyses is whether these assumptions are realistic. Could a motivated attacker perform these active attacks on a user's machine, or is such an attacker unlikely to exist?

This is important for the security of FIDO2. If these assumptions are realistic, then the aforementioned attacks may not present much of a threat. However, if they prove to be too optimistic, it means that it is plausible that a user's machine may become compromised in such a way that the aforementioned attacks can be performed, and therefore that the environment on which FIDO2 (or, more specifically, CTAP2) is executed plays a larger role on the security of the protocol. In this work, we provide real-world attacks that serve as motivation for further research on how the protocol can be changed to offer security against an attacker that can completely manipulate messages exchanged between a client and an authenticator.

In short, we first investigated the feasibility of attacking FIDO2[6] via CTAP2. Then, after showing the feasibility of such attacks, and thus that the assumptions from [1, 2] are too optimistic, we developed concrete modifications to CTAP2 that provably make it resistant to these attacks.

1.3. Work conducted and associated publications

Our work can be divided in three parts, with each resulting in a new publication: (i) the implementation of several attacks on FIDO2, meant to show how an adversary *might* perform them on real-world scenarios; (ii) the proposal of two design changes (or fixes) to CTAP2, accompanied with complete security proofs that show these modifications make FIDO2 resistant against the aforementioned attacks; and (iii) the integration and subsequent benchmark of our proposed fixes in commercial implementations of FIDO2, to assess their implementation cost and performance overhead.

[6] Although the attacks exploit vulnerabilities in CTAP2, the consequences affect the whole FIDO2 execution, since they allow an adversary to obtain unauthorized access to a user's account.

1.3.1. Practical attacks on FIDO2

To answer the question on whether the assumptions made in [1, 2] hold in real-world settings, we first set out to explore how an attacker could realistically carry out the previously described attacks. To do so, we built proof-of-concept implementations for three attacks, with each targeting one or more design choices of CTAP2. Concretely, we were able to: (i) leverage a man-in-the-middle attack to perform an *impersonation attack* to log into a user's account without its knowledge; (ii) perform a *rogue key attack* where an attacker registers its own authenticator on a user's account instead of the user's authenticator; and (iii) combine a man-in-the-middle with a rogue key attack to perform a *double registration attack*, registering an attacker's authenticator on a user's account alongside the user's own authenticator.

The rogue key attack FIDO2 clients send messages to authenticators through CTAP2 commands. These commands are authenticated with a key, called a pinUvAuthToken, that is sent encrypted by the authenticator to the client. The former then only accepts commands authenticated with the pinUvAuthToken. The rogue key attack exploits a design choice that allows an authenticator's response to a client's command to be hijacked by an attacker, because the authenticator's response is not authenticated. A client can send a command to an authenticator, but the response is not guaranteed to have come from that same authenticator.

The man-in-the-middle attack The man-in-the-middle attack that was at the center of our impersonation attack exploited an unauthenticated Diffie-Hellman key exchange in CTAP2, which is performed to derive an ephemeral, shared secret symmetric key between the client and the authenticator. This key is used by the authenticator to send an encrypted pinUvAuthToken to the client. An attacker that performs this attack can obtain the pinUvAuthToken, allowing it to gain almost complete access to the user's authenticator by enabling it to send authenticated commands (an ability only the client is supposed to have, under the user's approval with a PIN or biometric authentication).

The double registration attack This attack leverages both the lack of token-side authentication and the unauthenticated key exchange. It first performs a rogue key attack to register the attacker's token on the user's account, and then starts a new registration run with the same relying party to register a new key. The request sent by the relying party is properly authenticated with the pinUvAuthToken obtained by performing a man-in-the-middle attack.

Feasibility of attacks These implementations, even as proofs of concept, show that attacking FIDO2 is plausible in a real-world scenario, and therefore that the assumptions from [1, 2] are too optimistic with respect to the real-world threat environment of FIDO2. Our attacks are deployed through a malicious shared library that can be installed via an attack vector on a user's computer without requiring special privileges. For demonstration, we employed a simple malicious spreadsheet with a *macro* that downloads and installs the library, representing a scenario where a user is tricked into downloading and opening the file (e.g., via a phishing email). In practice, however, a skilled attacker could employ more sophisticated delivery methods to install the malicious library.

Additionally, a recent work on USB injection attacks [18] demonstrated that it is possible to perform message injection attacks on USB communications between a machine and a legitimate USB device, tricking the host into accepting a response from a purpose-built malicious USB device. In the paper, the authors manage to build devices that can inject key strokes by passing itself as a keyboard or alter files that are transfered from the legitimate USB device to the target machine. These attacks strengthen our claim on the plausibility of attacking FIDO2 at the CTAP2 level, since the methods used for injecting USB messages on the target machine might be repurposed by a FIDO2 attacker to inject CTAP2 messages that fool the client into believing that it is communicating with a legitimate authenticator.

This work resulted in the following publication:

Manuel Barbosa, André Cirne, Luís Esquível: Rogue key and impersonation attacks on FIDO2: From theory to practice. ARES 2023: 14:1-14:11.

Our main contributions for this paper were the design and implementation of the attacks, with a smaller contribution on the deployment strategy.

1.3.2. Improving FIDO2

After demonstrating the practicality of performing these active attacks on CTAP2, we set out to show how the protocol could be changed such that it could be proven secure without requiring the same assumptions.

Fixing the rogue key attack Our fix to the rogue key attack consists of authenticating token responses, using a key that is derived from the pinUvAuthToken. In our modified CTAP2 protocol, which we called CTAP 2.1+ (based on CTAP 2.1, the concrete version of CTAP2 we worked on), the client and the token each derive two unique keys from the

pinUvAuthToken, where one is used to authenticate a client's command to the token and the other to authenticate the token's response to the client.

We provide a full formal security analysis of CTAP 2.1+, where we specify the complete protocol syntax and security model in pseudocode, based on the work from [2], and provide a security proof that shows that our modifications make FIDO2 with CTAP 2.1+ resistant against rogue key attacks. This security proof is based on our proof for CTAP 2.1, which fixes some oversights from [2].

This work resulted in the following publication:

Manuel Barbosa, Alexandra Boldyreva, Shan Chen, Kaishuo Cheng, Luís Esquível: Privacy and Security of FIDO2 Revisited. Proc. Priv. Enhancing Technol. 2025(3): 304-332 (2025)

The paper formally analyses the privacy and security guarantees of FIDO2, while also proposing fixes to improve both aspects of the protocol. Our work focused on the authentication security of FIDO2, with the aforementioned results for CTAP 2.1 and CTAP 2.1+. The analysis of the FIDO2 privacy guarantees led to a fix on top of CTAP 2.1+, where authenticator ECDH key pairs are freshly generated every time the client requests a key exchange (which is not the case for CTAP 2.1 or CTAP 2.1+). However, this was not part of our core contributions, and is therefore not included in this thesis.

Fixing the man-in-the-middle attack Our man-in-the-middle attack fix involves the replacement of the unauthenticated key exchange with a Password-Authenticated Key Exchange (PAKE) [19, 20], a different type of cryptographic scheme that is resistant against man-in-the-middle attacks. In our modified protocol, which we call sCTAP 2.1+, the client and the token derive a shared secret key using the underlying PAKE protocol, and use that as the new pinUvAuthToken.

As with CTAP 2.1+, we present the modified protocol syntax and security model, as well as a security proof that shows our modifications make FIDO2 with sCTAP 2.1+ resistant against man-in-the-middle attacks.

1.3.3. Implementation of CTAP 2.1+ and sCTAP 2.1+

We complete our work on improving FIDO2's security by successfully implementing and benchmarking our proposed fixes in CTAP 2.1+ and sCTAP 2.1+ on commercial implementations of FIDO2. Our security analyses show that these fixes make FIDO2 resistant

against rogue key and man-in-the-middle attacks, but provide no answers to (i) the implementation costs of adding our fixes to already existing FIDO2 implementations, (ii) how our fixes impact the protocol's performance, and (iii) how (and if) they can implemented in a way that provides full backwards compatibility with the current versions of FIDO2.

To answer these questions, we implemented both fixes on client-side and token-side Rust implementations of FIDO2, maintained by Mozilla and Nitrokey respectively.

Implementing CTAP 2.1+ Implementing the rogue key fix from CTAP 2.1+ involved developing the whole token-to-client authentication mechanism, but did not require the addition of any new cryptographic primitive or protocol.

Implementing sCTAP 2.1+ Implementing the man-in-the-middle fix from sCTAP 2.1+ required more modifications to the base implementations of FIDO2 we used, as it required the addition of a new PAKE protocol to the CTAP2 flow. Out of multiple existing PAKE protocols we could have used for our implementation, we chose CHIC [21], a recent protocol based on post-quantum Key Encapsulation Mechanisms (KEM), and for which there already existed a reference C implementation from the original authors that instantiated the KEM with a reference implementation of CRYSTALS-Kyber [22, 23]. We adapted this CHIC implementation to Rust, and built additional modular support for different implementations of Kyber (and ML-KEM).

Verified Post-Quantum KEM Adding this support opened the possibility of including a verified implementation of ML-KEM, written in Jasmin[7], to our CHIC implementation, which could then be used on sCTAP 2.1+. However, because Jasmin is a very low level language, each implementation must target a specific architecture. At the time, there already existed a Jasmin x86 implementation of ML-KEM, maintained by the Formosa Crypto project[8] in the libjade[9] library. However, the authenticator on which we ran our implementations has a Cortex-M33 processor, which is an ARM-based processor for embedded devices. As a result, we wrote the first ARM version of the Jasmin ML-KEM implementation, added it to CHIC, and integrated and benchmarked it successfully on sCTAP 2.1+.

This work contributed to the following publication:

José Bacelar Almeida, Gustavo Xavier Delerue Marinho Alves, Manuel Barbosa, Gilles Barthe, Luís Esquível, Vincent Hwang, Tiago Oliveira, Hugo Pacheco, Peter Schwabe, Pierre-Yves

[7] https://github.com/jasmin-lang/jasmin
[8] https://formosa-crypto.org/
[9] https://github.com/formosa-crypto/libjade

Strub: Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt. SP 2025: 3820-3838

This paper presents novel techniques for verifying the correctness of optimized cryptographic implementations, dramatically simplifying proofs for cross-architecture implementations. Our contribution was the development of the Jasmin ML-KEM implementation for ARM.

1.4. Structure of this thesis

This thesis is organized into eight chapters. Chapter 1 contains this introduction, contextualizing the motivation for our work and explaining our main contributions and associated publications. Chapter 2 includes the necessary preliminaries for the rest of the document, including notation and relevant definitions. Chapter 3 introduces FIDO2, starting with a brief history and high level overview of the protocol, and then introducing its two core components, WebAuthn and CTAP2, in greater detail. Chapter 4 details our implementation and deployment work on the aforementioned attacks on FIDO2. Chapters 5 and 6 present our work on the provable security of CTAP 2.1+ and sCTAP 2.1+, respectively. In Chapter 7, we detail our implementations of CTAP 2.1+ and sCTAP 2.1+, as well as the experimental results obtained from benchmarking them against the base CTAP2 implementations. Finally, Chapter 8 concludes this thesis.

2. Preliminaries

This Chapter introduces the mathematical notation and formal security definitions relevant to the work conducted in this thesis, presented respectively in Sections 2.1 and 2.2. Section 2.3 introduces the ML-KEM scheme. Finally, we conclude with Section 2.4, where we establish the syntax and security properties of PAKE protocols.

2.1. Notation

Throughout this thesis, $\{0,1\}^n$ represents the set of all n-length bit strings and $\{0,1\}^*$ the set of all finite length bit strings, including the empty string ϵ. We write $x \leftarrow F$ (resp. $x \xleftarrow{\$} F$) to denote that x is the return value of a deterministic (resp. probabilistic) algorithm F, $x \leftarrow y$ to assign y to variable x, and $x \xleftarrow{\$} \mathcal{S}$ to assign an element from $\mathcal{S}$, chosen uniformly at random (unless otherwise specified), to x. When initializing an empty set $\mathcal{L}$, we write $\mathcal{L} \leftarrow \emptyset$. We use $\perp$ to denote any uninitialized variable, as well as the return value from an algorithm that executes incorrectly.

2.2. Security Definitions

Definition 2.1. For any function $f : \mathbb{N} \to \mathbb{R}_0^+$, we say that f is *negligible* if, for every positive polynomial p, there exists an integer N such that for all $n > N$,

$$f(n) < \frac{1}{p(n)}.$$

2.2.1. Symmetric cryptography

Definition 2.2. Let $\mathsf{H} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ be a family of functions such that $\mathsf{H}(k,\cdot) = \mathsf{H}_k(\cdot)$ is efficient for all k. We say that H is collision resistant if the advantage of any efficient adversary $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{A}) = \mathsf{Pr}[k \xleftarrow{\$} \mathcal{K}, (m_1, m_2) \xleftarrow{\$} \mathcal{A}(k) : m_1 \neq m_2 \ \wedge \ \mathsf{H}_k(m_1) = \mathsf{H}_k(m_2)] \ .$$

Definition 2.3. For any message authentication code $\mathsf{MAC} = (\mathsf{Kg}, \mathsf{Auth}, \mathsf{Ver})$, we say MAC is $\mathsf{SUF\text{-}CMA}$ secure if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}$ (Fig. 2.1), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A}) = \mathsf{Pr}[\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A}) = 1] \ .$$

$\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A})$:
- $K \xleftarrow{\$} \mathsf{MAC.KG}()$
- $S \leftarrow \emptyset$
- $(m,t) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Auth}},\mathcal{O}_{\mathsf{Ver}}}()$
- **return** $\mathcal{O}_{\mathsf{Ver}}(m,t) \wedge (m,t) \notin S$

$\mathcal{O}_{\mathbf{Auth}}(m)$:
- $t \leftarrow \mathsf{MAC}(K,m)$
- $S \leftarrow S \cup (m,t)$
- **return** t

$\mathcal{O}_{\mathbf{Ver}}(m,t)$:
- $t' \leftarrow \mathsf{MAC}(K,m)$
- **return** $t' = t$

Figure 2.1: The SUF-CMA security experiment and corresponding oracles $\mathcal{O}_{\mathsf{Auth}}$ and $\mathcal{O}_{\mathsf{Ver}}$.

Definition 2.4. Let $\mathsf{F} : \{0,1\}^l \times \{0,1\}^m \to \{0,1\}^m$ be a keyed family of permutations such that for every $k \in \{0,1\}^l$, $\mathsf{F}_k(\cdot) = \mathsf{F}(k,\cdot)$ is a permutation over $\{0,1\}^m$ and both F_k and F_k^{-1} are efficiently computable. Let $\mathcal{F}$ denote the set of all permutations over $\{0,1\}^m$.

We say F is a pseudo-random permutation if for every efficient adversary $\mathcal{A}$, the advantage defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{prp}}_{\mathsf{F}}(\mathcal{A}) = \left| \mathsf{Pr}[\mathsf{k} \xleftarrow{\$} \{0,1\}^l : \mathcal{A}^{\mathsf{F}_k,\mathsf{F}_k^{-1}}() = 1] - \mathsf{Pr}[\mathsf{f} \xleftarrow{\$} \mathcal{F} : \mathcal{A}^{\mathsf{f},\mathsf{f}^{-1}}() = 1] \right| \ .$$

Definition 2.5. For any symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, we say SKE is IND-1CPA (one-time IND-CPA) secure if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}$ (Fig. 2.2), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{A}) = \left| \mathsf{Pr}[\mathsf{Expt}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| \ .$$

$\mathsf{Expt}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{A})$:
- $b \xleftarrow{\$} \{0,1\}$
- $K \xleftarrow{\$} \mathsf{SKE.KG}()$
- $\mathsf{state} \leftarrow 0$
- $(m_0,m_1) \xleftarrow{\$} \mathcal{A}()$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0,m_1)$
- $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}}(c_{\mathsf{chal}})$
- **return** $(b = b')$

$\mathcal{O}_{\mathsf{LR}}(m_0,m_1)$:
- **if** $\mathsf{state} \neq 0$ **then**
  - **return** $\perp$
- **if** $|\, m_0 \,| \neq |\, m_1 \,|$ **then**
  - **return** $\perp$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathsf{SKE.Enc}(K, m_b)$
- $\mathsf{state} \leftarrow 1$
- **return** c_{chal}

Figure 2.2: The IND-1CPA security experiment, and corresponding Left-Right oracle $\mathcal{O}_{\mathsf{LR}}$.

Definition 2.6. For any symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, we say SKE is IND-1$PA secure[1] if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\text{IND-1\$PA}}_{\mathsf{SKE}}$ (Fig. 2.3), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\text{IND-1\$PA}}_{\mathsf{SKE}}(\mathcal{A}) = \left| \Pr[\mathsf{Expt}^{\text{IND-1\$PA}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| .$$

$\mathsf{Expt}^{\text{IND-1\$PA}}_{\mathsf{SKE}}(\mathcal{A})$:
- $b \xleftarrow{\$} \{0,1\}$
- $K \xleftarrow{\$} \mathsf{SKE.KG}()$
- $\mathsf{state} \leftarrow 0$
- $(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
- $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}}(c_{\mathsf{chal}})$
- **return** $(b = b')$

$\mathcal{O}_{\mathsf{LR}}(m_0, m_1)$:
- **if** $\mathsf{state} \neq 0$ **then**
  - **return** $\perp$
- **if** $|\, m_0 \,| \neq |\, m_1 \,|$ **then**
  - **return** $\perp$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathsf{SKE.Enc}(K, m_b)$
- $\mathsf{state} \leftarrow 1$
- **return** c_{chal}

$\mathcal{O}_{\mathsf{RAND}}(l)$:
- $m'_0, m'_1 \xleftarrow{\$} \{0,1\}^l$
- $c' \xleftarrow{\$} \mathsf{SKE.Enc}(K, m'_b)$
- **return** (m'_0, m'_1, c')

Figure 2.3: The IND-1$PA security experiment, and corresponding Left-Right oracle $\mathcal{O}_{\mathsf{LR}}$ and RAND oracle $\mathcal{O}_{\mathsf{RAND}}$.

Definition 2.7. For any symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, we say SKE is IND-1$PA-LPC secure[2] if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}$ (Fig. 2.4), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A}) = \left| \Pr[\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| .$$

Definition 2.8. For any symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, we say SKE is IND-1$PA-LHPC secure if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}$ (Fig. 2.5), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{A}) = \left| \Pr[\mathsf{Expt}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| .$$

[1] This security definition was introduced in [1].

[2] This security definition was introduced in [2], and builds on the IND-1$PA definition from [1] by introducing a left plaintext checking oracle $\mathcal{O}_{\mathsf{LPC}}$.

$\mathsf{Expt}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{A})$:
- $b \xleftarrow{\$} \{0,1\}$
- $K \xleftarrow{\$} \mathsf{SKE.KG}()$
- $\mathsf{state} \leftarrow 0$
- $(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
- $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}, \mathcal{O}_{\mathsf{LPC}}}(c_{\mathsf{chal}})$
- **return** $(b = b')$

$\mathcal{O}_{\mathsf{RAND}}(l)$:
- $m'_0, m'_1 \xleftarrow{\$} \{0,1\}^l$
- $c' \xleftarrow{\$} \mathsf{SKE.Enc}(K, m'_b)$
- **return** (m'_0, m'_1, c')

$\mathcal{O}_{\mathsf{LR}}(m_0, m_1)$:
- **if** $\mathsf{state} \neq 0$ **then**
  - **return** $\perp$
- **if** $|\, m_0 \,| \neq |\, m_1 \,|$ **then**
  - **return** $\perp$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathsf{SKE.Enc}(K, m_b)$
- $\mathsf{state} \leftarrow 1$
- **return** c_{chal}

$\mathcal{O}_{\mathsf{LPC}}(c)$:
- **if** $c = c_{\mathsf{chal}}$ **then**
  - **return** $\perp$
- **return** $m_0 = \mathsf{SKE.Dec}(K, c)$

Figure 2.4: The IND-1$PA-LPC security experiment, and corresponding Left-Right oracle $\mathcal{O}_{\mathsf{LR}}$, RAND oracle $\mathcal{O}_{\mathsf{RAND}}$ and LPC oracle $\mathcal{O}_{\mathsf{LPC}}$.

$\mathsf{Expt}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{A})$:
- $b \xleftarrow{\$} \{0,1\}$
- $K \xleftarrow{\$} \mathsf{SKE.KG}()$
- $\mathsf{state} \leftarrow 0$
- $(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
- $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}, \mathcal{O}_{\mathsf{LHPC}}}(c_{\mathsf{chal}})$
- **return** $(b = b')$

$\mathcal{O}_{\mathsf{RAND}}(l)$:
- $m'_0, m'_1 \xleftarrow{\$} \{0,1\}^l$
- $c' \xleftarrow{\$} \mathsf{SKE.Enc}(K, m'_b)$
- **return** (m'_0, m'_1, c')

$\mathcal{O}_{\mathsf{LR}}(m_0, m_1)$:
- **if** $\mathsf{state} \neq 0$ **then**
  - **return** $\perp$
- **if** $|\, m_0 \,| \neq |\, m_1 \,|$ **then**
  - **return** $\perp$
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathsf{SKE.Enc}(K, m_b)$
- $\mathsf{state} \leftarrow 1$
- **return** c_{chal}

$\mathcal{O}_{\mathsf{LHPC}}(c)$:
- **if** $c = c_{\mathsf{chal}}$ **then**
  - **return** $\perp$
- **return** $\mathsf{H}(m_0) = \mathsf{SKE.Dec}(K, c)$

Figure 2.5: The IND-1$PA-LHPC security experiment, and corresponding Left-Right oracle $\mathcal{O}_{\mathsf{LR}}$, RAND oracle $\mathcal{O}_{\mathsf{RAND}}$ and LHPC oracle $\mathcal{O}_{\mathsf{LHPC}}$.

Definition 2.9. For any symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, we say SKE is IND-1$PA-LHPC-M secure if for any efficient adversary $\mathcal{A}$ against the security experiment $\mathsf{Expt}^{\text{IND-1\$PA-LHPC-M}}_{\mathsf{SKE}}$ (Fig. 2.6), the advantage of $\mathcal{A}$ defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\text{IND-1\$PA-LHPC-M}}_{\mathsf{SKE}}(\mathcal{A}) = \left| \Pr[\mathsf{Expt}^{\text{IND-1\$PA-LHPC-M}}_{\mathsf{SKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| .$$

2.2.2. Asymmetric cryptography

Definition 2.10. Let $\mathbb{G}$ be a cyclic group of prime order q and generator g. The Strong Computational Diffie-Hellman (sCDH) assumption states that given $a, b \xleftarrow{\$} \mathbb{Z}_q$, g, g^a, g^b, and an oracle $\mathcal{O}_a$ which, for any group elements $Y, Z \in \mathbb{G}$ checks if $Y^a = Z$, it is computationally infeasible for any efficient adversary $\mathcal{A}$ to compute g^{ab}. That is, the advantage of any efficient adversary $\mathcal{A}$ defined below is negligible in the security parameter.

$\mathsf{Expt}^{\text{IND-1\$PA-LHPC-M}}_{\mathsf{SKE}}(\mathcal{A})$:
$b \xleftarrow{\$} \{0,1\}$
$K \xleftarrow{\$} \mathsf{SKE.KG}()$
$\mathsf{state} \leftarrow 0$
$(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
$c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
$t_{\mathsf{chal}} \xleftarrow{\$} \mathsf{MAC}(K, c_{\mathsf{chal}})$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}, \mathcal{O}_{\mathsf{LHPC}}}(c_{\mathsf{chal}}, t_{\mathsf{chal}})$
return $(b = b')$

$\mathcal{O}_{\mathsf{LR}}(m_0, m_1)$:
if $\mathsf{state} \neq 0$ **then**
  return $\perp$
if $|\, m_0 \,| \neq |\, m_1 \,|$ **then**
  return $\perp$
$c_{\mathsf{chal}} \xleftarrow{\$} \mathsf{SKE.Enc}(K, m_b)$
$\mathsf{state} \leftarrow 1$
return c_{chal}

$\mathcal{O}_{\mathsf{RAND}}(l)$:
$m'_0, m'_1 \xleftarrow{\$} \{0,1\}^l$
$c' \xleftarrow{\$} \mathsf{SKE.Enc}(K, m'_b)$
return (m'_0, m'_1, c')

$\mathcal{O}_{\mathsf{LHPC}}(c)$:
if $c = c_{\mathsf{chal}}$ **then**
  return $\perp$
return $\mathsf{H}(m_0) = \mathsf{SKE.Dec}(K, c)$

Figure 2.6: The IND-1$PA-LHPC-M security experiment, and corresponding Left-Right oracle $\mathcal{O}_{\mathsf{LR}}$, RAND oracle $\mathcal{O}_{\mathsf{RAND}}$ and LHPC oracle $\mathcal{O}_{\mathsf{LHPC}}$.

$$\mathsf{Adv}^{\mathsf{sCDH}}_{\mathbb{G}}(\mathcal{A}) = \Pr[g^{ab} \leftarrow \mathcal{A}(\mathbb{G}, g, g^a, g^b, \mathcal{O}_a)] \ .$$

Definition 2.11. A Key Encapsulation Mechanism (KEM) is composed of three polynomial-time algorithms $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$:

- $\mathsf{KeyGen}(\lambda) \rightarrow (pk, sk)$: a probabilistic key generation algorithm that inputs a security parameter λ and outputs a pair of public and private keys (pk, sk).
- $\mathsf{Encaps}(pk) \rightarrow (c, K)$: a probabilistic encapsulation algorithm that inputs a public key pk and outputs a ciphertext c and a secret key K.
- $\mathsf{Decaps}(sk, c) \rightarrow K$: a deterministic decapsulation algorithm that inputs a secret key sk and a ciphertext c and outputs a secret key K.

Correctness imposes that for any public/private key pair $(pk, sk) \xleftarrow{\$} \mathsf{KeyGen}(\lambda)$, if $(c, K) \leftarrow \mathsf{Encaps}(pk)$ then $K = \mathsf{Decaps}(sk, c)$.

2.3. The ML-KEM scheme

The ML-KEM [24] scheme is an IND-CCA2-secure KEM built upon the presumed hardness of the Module Learning With Errors (MLWE) problem [25]. It has been standardized by the National Institute for Standards and Technology (NIST), and is derived from CRYSTALS-Kyber [22, 23]. Its core algebraic operations are defined over small vectors and matrices over the cyclotomic polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, with fixed parameters $n = 256$ and $q = 3329$. To achieve IND-CCA2 security, ML-KEM applies a (modified) Fujisaki-Okamoto transformation [26] to an IND-CPA-secure public-key encryption scheme.

The size of vectors and matrices is defined by a module rank $k \in \{2, 3, 4\}$, which differs for each parameter set defined by the scheme: ML-KEM-512 has $k = 2$ (NIST security category 1), ML-KEM-768 has $k = 3$ (NIST security category 3) and ML-KEM-1024 has $k = 4$ (NIST security category 5). The default parameter set recommended by NIST is ML-KEM-768 [24].

Differences to CRYSTALS-Kyber While ML-KEM is similar to the originally proposed CRYSTALS-Kyber, it introduces some changes that are worth mentioning. First and foremost, the length of the output shared secret key from Encaps and Decaps can be variable in Kyber, while it is fixed to 256 bits in ML-KEM.

Second, the Kyber Encaps function hashes the random 256-bit string generated in the beginning before it is used as input to the SHA3-512 function, which no longer happens in ML-KEM. The Kyber Encaps and Decaps functions also generate the final shared secret key K as the output of running a KDF (instantiated with SHAKE-256) where the input includes the hash of the generated ciphertext. This no longer occurs in ML-KEM, where K is simply part of the output of running the aforementioned SHA3-512.

Finally, ML-KEM enforces explicit input validation in Encaps, where it is ensured that, when decoded, polynomial coefficients are integers modulo q without any modular reductions.

2.4. Password Authenticated Key Exchange (PAKE) Protocols

A PAKE protocol is an interactive protocol between two parties: an Initiator I and a Responder R. Executing a PAKE protocol allows each party to establish a high entropy session key (which may for example be used to symmetrically encrypt or authenticate data) from a low entropy password that is shared by the two parties before-hand. In this work, we consider only PAKE protocols that follow a specific syntax and satisfy certain security properties, introduced below.

2.4.1. Syntax

We consider a class of PAKE protocols that fit a general, high-level syntax composed of 5 functions, with a 4-message flow of execution:

initStart: this function is executed by a client C (the Initiator), and inputs a low-entropy password pw, a unique client identifier id_C and a unique token identifier id_T (previously established outside the PAKE protocol) and outputs a client-side PAKE internal state state_C, which contains information that is PAKE-specific, and a message msg_1.

respStart: this function is executed by a token T (the Responder), and inputs a low-entropy password pw, a unique client identifier id_C, a unique token identifier id_T and msg_1 and outputs a token-side PAKE internal state state_T, which contains information that is PAKE-specific, and a message msg_2.

initAccept: this function is executed by a client C. It inputs msg_2 and the client-side PAKE state state_C, and outputs a message msg_3 and the updated state state_C.

respEnd: this function is executed by a token T. It inputs msg_3 and the token-side PAKE state state_T, and outputs a message msg_4 and a session key K, if T successfully validates message msg_3, or $\perp$ if it doesn't.

initEnd: this function is executed by a client C. It inputs a message msg_4 and the client-side PAKE state state_C and returns a session key K (which may be $\perp$) and a success bit b. If C cannot validate msg_4 successfully, the procedure fails.

2.4.2. Security properties

Throughout this work, we assume that any PAKE protocol under consideration satisfies the following three security properties: *explicit mutual authentication* (ma), *perfect forward secrecy* (pfs) and *sid collision resistance* (sid-coll). All definitions are described below, and the security experiments can be seen in Figure 2.7. Each security property is defined with a security experiment against a PAKE protocol PAKE. The security model we consider for these experiments is based on those from [1, 27], which are themselves based on the BPR model from [19]. We briefly introduce the session and experiment oracles, as well as how partnership between two parties is defined, and then define the security properties.

Session oracles We define session oracles for parties P in $\mathcal{P} \in \mathcal{I} \cup \mathcal{R}$, where $\mathcal{I}$ is the set of all Initiators and $\mathcal{R}$ the set of all Responders. Each party P has a unique identifier id_P. A session oracle π_P^i represents the i^{th} instance of party P.

Partnership Two instances π_I^i and π_R^j are partnered if and only if they both reach *accept* and agree on the same session key K and session identifier sid, which is a public value that uniquely identifies a session and is usually considered to be the trace of communications. It starts as $\perp$ in the beginning of the protocol's execution, and its value is set by each party on *accept*.

Accept* vs *terminate We recall the distinction between *accept* and *terminate* from [19]. The former is reached when a party sets its sid and holds a session key K, although it may

still require confirmation that its expected partner has also derived the same session key before using it. The latter is reached when a party no longer expects to send or receive any new messages, and commits to the derived session key K[3].

Experiment oracles In the beginning of the security experiment, the challenger runs an Initialization oracle which samples a password pw for each pair of parties (I, R) from the set of all valid passwords, according to some distribution $\mathscr{D}$. Then, $\mathcal{A}$ has access to the following oracles: Execute, Send^k, Corrupt, Reveal and Test.

The Execute oracle inputs an instance π_I^i and an instance π_R^j, performs a passive execution of the protocol between them, and returns the full trace of communications. The Send^k (with $k \in \{0,1,2,3,4\}$) oracle inputs an π_P^i and a message m and returns the resulting message to $\mathcal{A}$. Depending on k, Send executes different parts of the protocol (one of the 5 functions defined in the syntax). Corrupt inputs two parties P and P' and returns their shared password to $\mathcal{A}$[4]. The Reveal oracle inputs an instance π_P^i and returns its session key K, if it has accepted, and $\perp$ otherwise. Both π_P^i and its partner instance are then marked as unfresh. Finally, the Test oracle inputs an instance π_P^i and, if π_P^i is fresh, returns the session key K of the input instance π_P^i when the challenge bit b is 0 or an independent random session key $\tilde{K}$ if b is 1.

Instance freshness An instance π_P^i is considered *fresh* if (i) it and its partner weren't queried to Reveal *and* (ii) either $\mathcal{A}$ did not call Corrupt on π_P^i and its partner before calling Test *or* the protocol execution involving π_P^i was passive (i.e., $\mathcal{A}$ did not use Send to feed any altered message to π_P^i or its partner). The *freshness* of an instance is crucial because it rules out trivial attacks when querying Test in the pfs experiment.

We now briefly introduce the security definitions all PAKE protocols used in sCTAP 2.1+ must satisfy. The security experiments can be seen in Figure 2.7.

[3] In the PAKE syntax defined above, the Initiator always reaches *accept* first, with initAccept, then waits for the Responder to reach *accept* and *terminate* with respEnd, and finally reaches *terminate* with initEnd upon receiving msg_4.

[4] This deviates from [27], where Corrupt inputs a party P and returns all passwords shared between P and any other party.

$\mathsf{Expt}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{A})$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $() \leftarrow \mathcal{O}^{\mathsf{Initialization}}()$
3: $b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathcal{P})$
4: **return** $(b = b')$

$\mathsf{Expt}^{\mathsf{ma}}_{\mathsf{PAKE}}(\mathcal{A})$:
1: $() \leftarrow \mathcal{O}^{\mathsf{Initialization}}()$
2: win-ma $\leftarrow 0$
3: $() \leftarrow \mathcal{A}^{\mathcal{O}}(\mathcal{P})$
4: **if** unique partnership is broken $\wedge$ pw was not corrupted **then**
5: win-ma $\leftarrow 1$
6: **return** win-ma

$\mathsf{Expt}^{\mathsf{sid\text{-}coll}}_{\mathsf{PAKE}}(\mathcal{A})$:
1: $() \leftarrow \mathcal{O}^{\mathsf{Initialization}}()$
2: win-sidcoll $\leftarrow 0$
3: $() \leftarrow \mathcal{A}^{\mathcal{O}}(\mathcal{P})$
4: **if** sid collision occurs **then**
5: win-sidcoll $\leftarrow 1$
6: **return** win-sidcoll

Figure 2.7: The pfs, ma and sid-coll security experiments. $\mathcal{P}$ represents the entire set of parties involved in the PAKE protocol. $\mathcal{O}$ denotes all experiment oracles available to $\mathcal{A}$.

Perfect forward secrecy The pfs security of a PAKE protocol PAKE is defined with a security experiment $\mathsf{Expt}^{\mathsf{pfs}}_{\mathsf{PAKE}}$, executed between an adversary $\mathcal{A}$ and a challenger. In this experiment, the challenger samples a bit b. If $b = 0$, then it outputs the real session key K when $\mathcal{A}$ queries Test on an instance π^i_P. If $b = 1$, it outputs an independent random session key $\tilde{K}$. $\mathcal{A}$ wins if it outputs a guess bit b' such that $b = b'$. We say that PAKE provides pfs security if the advantage defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{A}) = \left| \Pr[\mathsf{Expt}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{A}) = 1] - \frac{1}{2} \right| .$$

Explicit mutual authentication A PAKE protocol PAKE provides ma security if and only if for any instance π^i_P that *terminates* successfully, there exists a *unique* partner instance $\pi^j_{P'}$ that has *accepted* with the same session key K and session identifier sid, except with negligible probability. This security goal is defined with a security experiment $\mathsf{Expt}^{\mathsf{ma}}_{\mathsf{PAKE}}$, executed between an adversary $\mathcal{A}$ and a challenger, where $\mathcal{A}$ wins if it can make an instance π^i_P terminate without a unique partner and without corrupting its password. We say that PAKE provides ma security if the advantage defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\mathsf{ma}}_{\mathsf{PAKE}}(\mathcal{A}) = \Pr[\mathsf{Expt}^{\mathsf{ma}}_{\mathsf{PAKE}}(\mathcal{A}) = 1] .$$

Session identifier collision resistance A PAKE protocol PAKE achieves sid-coll security if and only if for any instance π^i_P that terminates successfully, there *does not exist* another instance $\pi^j_{P'}$, where $(P, P') \in (\mathcal{I} \times \mathcal{I})$ or $(P, P') \in (\mathcal{R} \times \mathcal{R})$ and $P \neq P'$, such that $\pi^j_{P'}$ terminated and $\pi^i_P.\mathsf{sid} = \pi^j_{P'}.\mathsf{sid}$, except with negligible probability. This is defined with a security experiment $\mathsf{Expt}^{\mathsf{sid\text{-}coll}}_{\mathsf{PAKE}}$, executed between an adversary $\mathcal{A}$ and a challenger,

where $\mathcal{A}$ wins if it can make two instances of parties belonging to the same set terminate with the same sid. We say that PAKE provides sid-coll security if the advantage defined below is negligible in the security parameter.

$$\mathsf{Adv}^{\text{sid-coll}}_{\mathsf{PAKE}}(\mathcal{A}) = \mathsf{Pr}[\mathsf{Expt}^{\text{sid-coll}}_{\mathsf{PAKE}}(\mathcal{A}) = 1] \; .$$

3. The FIDO2 Protocol

In this Chapter, we introduce FIDO2, a protocol for secure authentication online. At its core, FIDO2 aims to end the reliance on password-based mechanisms by introducing a hardware-backed passwordless form of authentication. We begin in Section 3.1, where we present a very brief overview of the protocol's history, and then provide a simplified, high-level overview of FIDO2 in Section 3.2. Sections 3.3 and 3.4 describe the WebAuthn and CTAP2 protocols respectively, which are the two core (sub)protocols that, together, make FIDO2. Our descriptions are focused on the protocol's elements that are most relevant to this work. Interested readers seeking a more detailed account of WebAuthn or CTAP2 are referred to Appendices A and B, respectively. We conclude the Chapter in Section 3.5, with a summary of the covered topics.

3.1. History of FIDO

The FIDO (Fast Identity Online) standard was created by the FIDO Alliance, and consists of a set of protocols that leverage the use of public-key cryptography to offer a more secure and seamless form of authentication online. The first version of FIDO was released in 2014, and was composed of two protocols: UAF (Universal Authentication Framework) and U2F (Universal 2nd Factor). UAF is a passwordless authentication protocol that uses an on-device software stack and a local authentication mechanism such as biometric verification or a PIN to authenticate that device with a remote service, without needing a password. U2F is a second factor authentication protocol which aims to be more secure than other methods such as time-based one-time passwords (TOTP) by using external trusted hardware devices which, combined with a password, authenticate the user to a remote service.

In 2018, the FIDO Alliance, alongside the World Wide Web Consortium (W3C), introduced the second version of FIDO, called FIDO2, a new standard for authentication online that improves on UAF and U2F by offering a unified and standardized way of performing passwordless and multifactor authentication using both external and local trusted hardware-based authentication mechanisms. The former are usually known as *roaming* authenticators (e.g. FIDO USB security keys, smart cards, smartphones), and the latter as platform authenticators (e.g. Windows Hello, Touch ID). For the remainder of this thesis, we refer to these simply as *authenticators* or *tokens*.

3.2. Protocol overview

FIDO2 is composed of two protocols: WebAuthn, created by the World Wide Web Consortium (W3C), and the Client To Authenticator Protocol (CTAP), created by the FIDO Alliance. The WebAuthn specification defines an API that can be implemented in browsers or other client applications, offering a standard and well defined high-level mechanism for applications to interact with authenticators. Communications between clients (usually browsers) and authenticators are governed by CTAP. As of this writing, WebAuthn is currently in version 3 [28], although all of our work was conducted when it was still in version 2 [9]. The first major version of CTAP was the U2F protocol, now renamed to CTAP1. The second, which is included with FIDO2, is called CTAP2. As of this writing, CTAP is on version 2.2 [10].

3.2.1. Simplified flow of execution

FIDO2 is an online authentication protocol. Therefore, as with traditional password-based authentication, its general usage scenario involves two distinct, but related, procedures. The first is *registration*, where a user leverages its authenticator to register with a server (e.g. creating an account), and the second is *authentication*, where the same authenticator is used to prove its (the user's) identity to the service (e.g. logging in to an account). Figure 3.1 illustrates the basic flow of messages exchanged between the four participants of the protocol during any registration or authentication procedure: (i) the server, which in FIDO2 is called the *relying party*; (ii) the client, which is usually a common browser; (iii) the (human) user, which is trying to register or login with a relying party; (iv) and the authenticator, or *token*, which is the trusted hardware device owned by the user. Any FIDO2 execution starts with the user, which requests to register or authenticate with the server (*step 1*). The server issues a challenge, which is an ephemeral and unique random bitstring, and sends it to the client (*step 2*), which is most likely the user's browser. With the user's permission (*step 3*), the client then sends this challenge to the authenticator (*step 4*), which produces a response that is sent back to the server through the client (*steps 6 and 7*). To produce this response, the authenticator typically requires the user to interact with it in some way, such as through the press of a button (*step 5*).

The contents of the response depend on the procedure being executed. Concretely, registration is the process of an authenticator creating a signing public-private key pair, called a *FIDO2 credential*, and sending the public part to the server, where it is stored for later

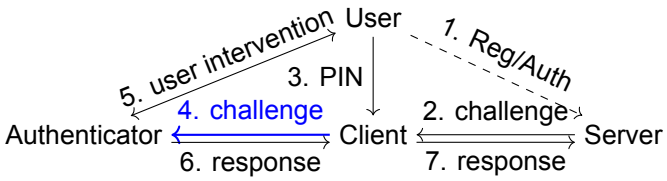


Figure 3.1: The simplified FIDO2 protocol flow, adapted from Fig. 2 in [1], where the CTAP2 authorized command is highlighted in blue. The dashed line means the communication with the server is established through the client.

use. Therefore, during a registration run, the response the authenticator produces contains this credential public key. Later, when the user starts an authentication run, the challenge issued by the server is signed using the private part of the credential generated in registration. This signature is part of the response that is sent to the server, where it is verified using the corresponding public key.

3.2.2. User verification and presence

Besides being the initiator of a FIDO2 flow, the user also plays a crucial role in ensuring that a client interacts with its authenticator *only* with its explicit permission. On FIDO2, this is called *user verification* (UV), and it is required for most registration and authentication flows, either via a PIN that the user knows, or through some other biometric mechanism like fingerprint reading. In this work, we assume user verification is *always* required.

In addition to providing user verification, the user is also required to interact with the authenticator in some way *before* it produces and outputs its response. In FIDO2, this is called proof of *user presence* (UP), and its purpose is to ensure that any registration or authentication is performed *only* when the user is present and aware of the procedure. User presence is *always* required for any ceremony to be successful, since it is enforced by the relying party itself.

3.2.3. The importance of user verification

Multifactor authentication is an extremely effective way of improving the security of online accounts. Colloquially, the most common three factors of authentication can be categorized as ”something you know”, ”something you have”, and ”something you are”. The most widely used factor is probably the password, i.e., ”something you know”. When FIDO U2F is used as a second factor of authentication, it adds a ”something you have” factor, which is the hardware authenticator.

When using a passwordless authentication flow with FIDO2, the password is removed from the authentication flow. Without user verification, this would make the hardware authenticator the only authentication factor remaining, which defeats the purpose of having multifactor authentication, like a bank card without a PIN. Indeed, if this was the case, stealing the card would grant the thief full access to the user's bank account until the card was canceled.

In this scenario, one can consider user verification with a PIN as the new ”something you know” factor, which now works similarly to the bank card, where this PIN is used to prove ownership of the authenticator without having to store this sensitive information on a remote server[1].

3.2.4. FIDO2 and passkeys

FIDO2 credentials are commonly known in the industry today as *passkeys* [29], and as mentioned in Chapter 1 have experienced increased adoption in the industry, with major players like Google, Apple and Microsoft[2] implementing this standard to provide users with the option to use passwordless authentication in their accounts. The work in this thesis is focused on FIDO2 use cases where credentials are created and stored in trusted hardware authenticators like FIDO2 security keys (e.g. YubiKeys). These are typically known as device-bound credentials. However, it is worth noting that the FIDO Alliance describes *passkeys* as ”discoverable FIDO credentials for passwordless authentication” [30, p. 42]. Under this definition, a *passkey* may be device-bound or synced across multiple devices via services like Apple Keychain, which offer greater convenience at the cost of additional trust being put on the companies that provide these features. A comparison between different types of *passkeys* can be consulted in [31, p. 6].

3.2.5. Considered usage scenarios

The FIDO2 protocol can be used in very different scenarios. Different relying parties will have different policies on how a user can perform a registration or authentication. Likewise, different authenticators, and different authenticator types (USB keys, NFC cards, smartphones), offer a varying set of functionalities that can affect how a user interacts with the protocol.

[1] If biometric authentication is used, then the PIN is not necessary to maintain two authentication factors, because a fingerprint scan, for example, would constitute the ”something you are” factor, which is also used to prove ownership of the authenticator.

[2] A list of services that support passkeys can be consulted in `https://fidoalliance.org/passkeys-directory/`.

In this work, we consider specific usage scenarios. First and foremost, we assume all FIDO2 flows require *user verification* and *user presence*. Second, we consider all user verification to be PIN-based, i.e., the browser always obtains its permission to access the authenticator by collecting the user's PIN. Third, we consider all authenticators to be USB authenticators, which must be plugged into the user's machine when used to register or authenticate with an account. We come back to our considered usage scenarios in Section 5.3 of Chapter 5. Finally, we consider only cases where FIDO2 credentials are *discoverable* (introduced later).

3.3. The WebAuthn protocol

The WebAuthn protocol [9] defines an API that clients (often standard browsers) can integrate and applications running on these clients can use to provide passwordless authentication to their users. As mentioned above, the entity consuming this API is referred to as WebAuthn Relying Party (or just relying party), which usually consists of some server-side and client-side logic, i.e. the application's backend and front-end respectively. On the other side of the interaction is the user who can, via a client, request to register or authenticate with the relying party, using an authenticator. This API connects the relying party application to the user's authenticator via the `navigator.credentials.create` and `navigator.credentials.get` methods, used for registration and authentication respectively. The responsibility of the relying party is in providing all the necessary information to the client through this API, and then handling its response. The API methods, implemented by the client, process this information and handle all the communication with the authenticator. From the relying party's point of view, these two methods are essentially a black box, where data is sent and received, with all the responsibility of handling communications with the authenticator resting on the client.

In this Section, we briefly describe the WebAuthn protocol, explaining in a very simplified way how a registration or authentication ceremony is executed, and leave out most details that are not as relevant to our work. The interested reader can read a more detailed account of the protocol in Appendix A.

Protocol versions Throughout the thesis, whenever we refer to WebAuthn, we are referring to Version (or Level) 2 [9], since this was the most recently published version of the protocol during our work. Nevertheless, the current WebAuthn Level 3 [28] does not seem to introduce any changes that contradict or are in any way incompatible with this work,

especially regarding the cryptographic core of the protocol. Therefore, this description should still apply to WebAuthn Level 3.

3.3.1. WebAuthn simplified flow

At a high level, the WebAuthn flow is extremely similar to the FIDO2 flow described in Section 3.2, with WebAuthn and FIDO2 even being sometimes informally used as synonyms. Indeed, when a user registers or logs into an account using FIDO2, it is interacting with the relying party directly. Excluding the user verification and user presence steps, it does not directly interact with CTAP2, and may even perceive these steps as parts of the same protocol (i.e., as part of WebAuthn). As such, WebAuthn is the ”visible” part of FIDO2 with which most users ever come into contact.

This protocol defines the data that must be exchanged between the relying party and the user's authenticator, and how the client must process, validate and repackage the information that is sent to or received from either party involved. Details on how the client (securely) communicates with the authenticator are out of scope of WebAuthn, and are completely abstracted from this protocol's specifications. That matter is handled by CTAP2.

Figure 3.2 shows a simplified message flow between a relying party, a client, and the user's authenticator during a *registration* and an *authentication* run.

3.3.2. Registration

When user requests to register, the relying party starts by generating a unique challenge, a random string of *at least* 16 bytes that must be signed by the authenticator, and a user identifier userID, which is a unique byte-string that will uniquely identify the user's account in all future authentication runs. These fields, especially the challenge, are usually generated server-side and sent to the client-side relying party implementation through a secure channel (established, for example, with TLS). Then, alongside the rpID, which is the relying party's unique identifier (RP ID), usually corresponding to its domain (the URL), they are sent to the client (not to be confused with the client-side relying party implementation) through the navigator.credentials.create method. At this point, the control of execution is passed to the client.

The client starts by validating the information provided by the relying party, to ensure it is correct and follows the protocol's specifications. The most important validation, where the client plays a vital role in this protocol's security, is perhaps the verification that the

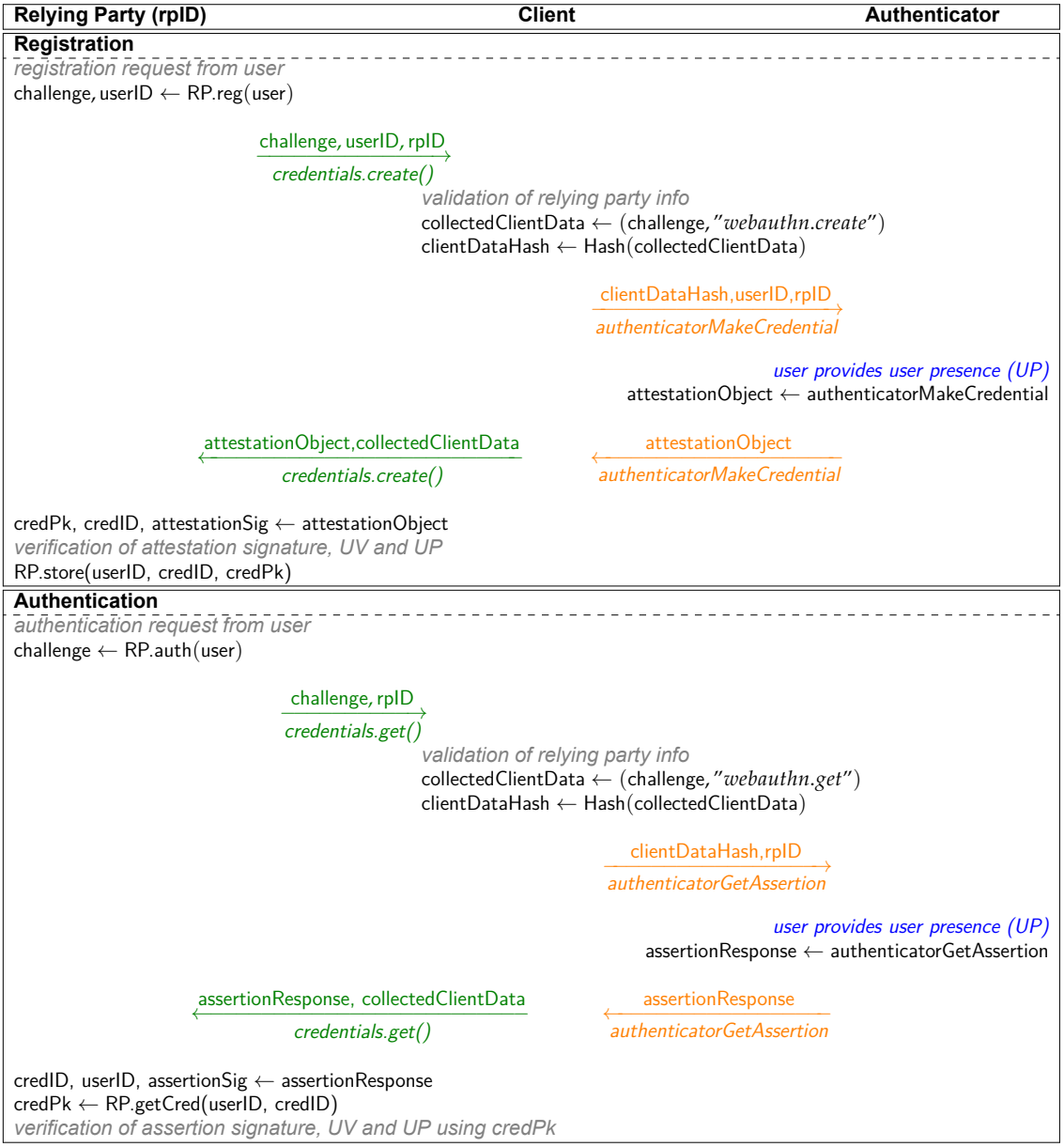


Figure 3.2: The (simplified) WebAuthn protocol. Arrows marked in green represent information exchanged by the relying party and the client through a secure channel (for example, through TLS). Arrows in orange represent the comunications between the client and the user's authenticator, protected by CTAP2. The blue text represents the user's actions for providing *user verification* and *user presence*.

received rpID matches the domain with which the client established a comunication channel[3]. If this validation fails, the registration ceremony is aborted. If not, then the client creates a new structure called collectedClientData, that contains among other things the received challenge and a hardcoded string "webauthn.create". This structure is then serialized and hashed into a clientDataHash field, which is sent alongside the rpID and the userID to the authenticator.

[3]If this validation did not occur, one relying party could attempt to pass itself as a different relying party by providing a different identifier that does not match with its domain.

The authenticator, upon receiving the information from the client, proceeds to generate a new credential key pair, with an identifier credID. The private key credSk is stored in the authenticator's permanent storage, indexed by the rpID and userID, and the public key credPk is signed, alongside the clientDataHash and other authenticator information, to produce an *attestation* signature attestationSig. This signature, the credential public key credPk and the credential identifier credID are then returned to the client as an attestationObject[4]. The client then returns this attestationObject, and the previously created collectedClientData, back to the relying party (this is the information returned from navigator.credentials.create).

Upon receiving the attestationObject and collectedClientData, the relying party verifies the attestation signature with an attestation public key (that will usually come in an attestation certificate inside the attestationObject) on the hashed collectedClientData (which must be identical to the clientDataHash signed by the authenticator), the received credential public key credPk, and some other authenticator information also included in the attestationObject. If the signature is valid, the relying party stores the new credPk, indexed by its identifier credID and the user identifier userID.

Attestation

In the registration flow described above, the authenticator produces an attestation object, which among other fields contains an *attestation* signature. In WebAuthn, an attestation signature is meant to provide some[5] assurance to the relying party regarding the origin of an authenticator. For example, the relying party may require the authenticator producing the credential to be a hardware authenticator from a trusted manufacturer instead of a software emulation.

Attestation is a relatively complex topic in WebAuthn, which we explain further in Appendix A, but in a nutshell, the relying party may, or may not, require the authenticator to produce an attestation signature. This information is conveyed to the authenticator alongside the challenge and the identifiers seen in Figure 3.2, which we omitted for simplicity. If attestation is *not* required, then the returned attestation object has no signature, and there exists no verification (and therefore no assurance) on the relying party's side to check if

[4]This is a very simplified account of what an attestation object is. As previously mentioned, a more detailed description can be consulted in subsection A.2.2 of Appendix A.

[5]An attestation signature may not identify a unique authenticator to the relying party, but it may still provide a guarantee that it was produced by a hardware authenticator.

the response came from a hardware authenticator. This is known as the None attestation type. If the relying party requires attestation, then the authenticator must somehow produce an attestation signature.

There are five types of attestations: None, Self, Basic, AttCA and AnonCA, but for the purposes of this work we consider only the first three[6]. When attestation type is None, no attestation signature is produced by the authenticator. With attestation type Self, the authenticator produces the attestation signature with the newly generated credential private key. With attestation mode Basic, the authenticator possesses an attestation private key that has been included by the manufacturer, and uses that key to produce the signature. The certificate with the public key is then included alongside the signature in the attestation object, and validated by the relying party through the PKI. Clearly, only the Basic attestation type is capable of providing assurance to the relying party that the response came from a physical authenticator (assuming the attestation private key was not leaked).

One important aspect regarding attestation key pairs in Basic attestation is that they are not unique for each authenticator. In fact, each key pair can be shared by up to 100000 authenticators, usually of the same make and model (see Section 14.4.1. of the specification [9]). This is a design choice of WebAuthn to improve anonymity (and why this type is also called Batch attestation).

Discoverable credentials

Client-side discoverable credentials, or simply discoverable credentials (previously known as resident keys), refer to generated credential private keys that are stored locally in the authenticator, bound to a specific RP ID and user identifier (also refered to as user handle). This allows a relying party to request a signature from the authenticator during an authentication ceremony using only the RP ID, without needing to provide the credential identifier that was returned by the authenticator during registration and stored in the relying party's servers. When an authenticator generates a discoverable credential, the credential identifier is a random byte-string of at least 16 bytes.

An authenticator can save, at most, one discoverable credential per relying party per user handle. As a consequence, one notable characteristic about discoverable credentials is that it is possible for a relying party to start an authentication ceremony without first identifying the user. Indeed, when an authenticator receives an authentication request, it can check if there are any credentials stored under the provided RP ID. If there is only

[6] A description of every attestation type can be consulted in subsection A.2.3 of Appendix A.

one, then the user handle associated with it must correspond to the user account trying to login. If there exists more than one credential, then the user must choose with which account it is trying to authenticate (through visual input from the authenticator or, if that is not possible, from the client). The correct user handle is returned to the relying party in the authenticator's response.

Server-side credentials, or non-discoverable credentials (previously non-resident keys), are not stored locally in the authenticator (maybe the authenticator doesn't support discoverable credentials). Instead, the authenticator encrypts the generated credential private key and sends it to the relying party to be stored. In these scenarios, the credential identifier *is* the ciphertext, which can only be decrypted by the authenticator. Later, when the relying party performs an authentication run, this credential identifier *must* be sent to the authenticator, so the credential private key can be recovered. With non-discoverable credentials, the relying party must always know the identity of the user to start an authentication ceremony, because it must know which credential identifier (or identifiers) must be provided to the authenticator. The same user handle can maintain multiple non-discoverable credentials for the same RP ID.

Authenticator data

The attestation object contains a field called authenticatorData, which is a structure that contains the credential public key generated by the authenticator and two very important flags: the UV flag, which is set by the authenticator if user verification was performed, and the UP flag, which is set if the proof of user presence was provided. Because the authenticatorData structure is signed by the authenticator, alongside the clientDataHash, the relying party can always be sure if user verification and user presence were performed when responding to a registration or authentication ceremony, and reject if its expectations weren't met (for example, if the UP flag is not set to 1). This structure is returned by the authenticator during a registration or authentication run, but in the latter it does not come as a part of an attestation object and does not include a credential public key.

3.3.3. Authentication

Authentication and registration ceremonies have a very similar flow of execution. At its core, an authentication run is meant to prove to the relying party that the user that is attempting to log into an account possesses, and controls, the authenticator that previously registered a credential for that account.

When a user requests to authenticate, the relying party generates a new random challenge that is passed to the client, along with the relying party identifier, using the navigator.credentials.get method.[7]

After initial validations, identical to those in registration, the client creates the collectedClientData structure containing the received challenge and a hardcoded string "webauthn.get". The structure is then hashed to create the clientDataHash, which is sent to the authenticator with the relying party identifier rpID.

When the authenticator receives this information, it fetches the credential private key that was stored for that relying party (which was indexed by its identifier and the user identifier for the account the user is trying to log into) and uses it to sign the clientDataHash and other data that is part of the aforementioned authenticatorData structure, including the UV and UP flags. The resulting *assertion* signature is then returned, alongside the credential idenfitier credID and the associated user identifier userID, back to the client, which redirects this information and the collectedClientData back to the relying party.

When the relying party receives the client's response, it uses the userID and credID to fetch the correct credential public key and uses it to verify the assertion signature. If this verification is successful, the authentication procedure is completed and the user is authenticated to its account. If not, then the authentication procedure fails.

3.4. The CTAP2 protocol

In the WebAuthn specification, the client acts as an intermediary between the relying party application and the user's authenticator. The communications between the relying party and the client are performed through a secure channel established, for example, with TLS. The communications between the client and the authenticator are mediated through CTAP2.

The CTAP2 protocol's main responsibility is to ensure that a client application can only access an authenticator with its owner's (the user) approval. In this context, having access means that a client is capable of producing and sending *authenticated* commands, such as for creating a new credential or signing a challenge, that are accepted by the

[7] If the authenticator had registered one or more non-discoverable credentials for that user account, the relying party would have to fetch and send those credentials (which are associated to the unique user identifier) to the authenticator in a list, which would then choose one to sign the challenge. This is not the case in Figure 3.2, because we are only considering discoverable credentials.

authenticator. In other words, CTAP2 establishes an authenticated (but not encrypted) channel between the client and the authenticator.

In this Section, we briefly introduce CTAP2, describing the protocol's simplified flow of execution and its most important elements. As in Section 3.2, this is an introductory explanation of CTAP2, which focuses on the protocol's details that are more relevant for this work, and we refer the interested reader to a more detailed description in Appendix B.

Protocol versions At the time of writing, there exist three versions of the CTAP2 protocol: CTAP 2.0, which was officially published in 2019 [32], CTAP 2.1, officially published in 2021 [33], and CTAP 2.2, officially published in 2025 [10]. This work focuses on CTAP 2.1, as this was the most recent version available at the time. Nevertheless, the differences introduced by CTAP 2.2 do not in any way affect our results. Therefore, though we do refer to CTAP 2.1 throughout this thesis, all results still apply to CTAP 2.2.

On the other hand, CTAP 2.0 and CTAP 2.1 have substantial differences, some of which are important in the context of this work. Therefore, we sometimes differentiate between these two versions whenever the differences are relevant to the subject being discussed. In scenarios where this is not the case, we may simply refer to the protocol as CTAP2.

3.4.1. CTAP2 simplified flow

Unlike WebAuthn, CTAP2 operates on a lower level, with a flow of execution that works almost as a "background" process during a WebAuthn registration or authentication ceremony. This is apparent in the diagram from Figure 3.2, where the orange arrows correspond to exchanges of information that occur through a CTAP2 authenticated channel, even though the details on how this channel is established are abstracted away.

Before the client can send the relying party information to the authenticator, the following procedure must be executed: (i) the client requests and receives information from the authenticator regarding its capabilities; (ii) based on the received information, it collects the user's PIN and performs a key exchange with the authenticator to derive a shared secret key; (iii) the client hashes and encrypts the user's PIN with the shared secret key, and sends it to the authenticator; (iv) the authenticator decrypts and verifies the PIN, uses the same shared secret to encrypt a new key, called a `pinUvAuthToken` (or `pinToken` in CTAP 2.0), and sends it to the client; (v) the client decrypts the ciphertext and obtains the `pinUvAuthToken`; (vi) finally, the client uses the `pinUvAuthToken` to authenticate commands

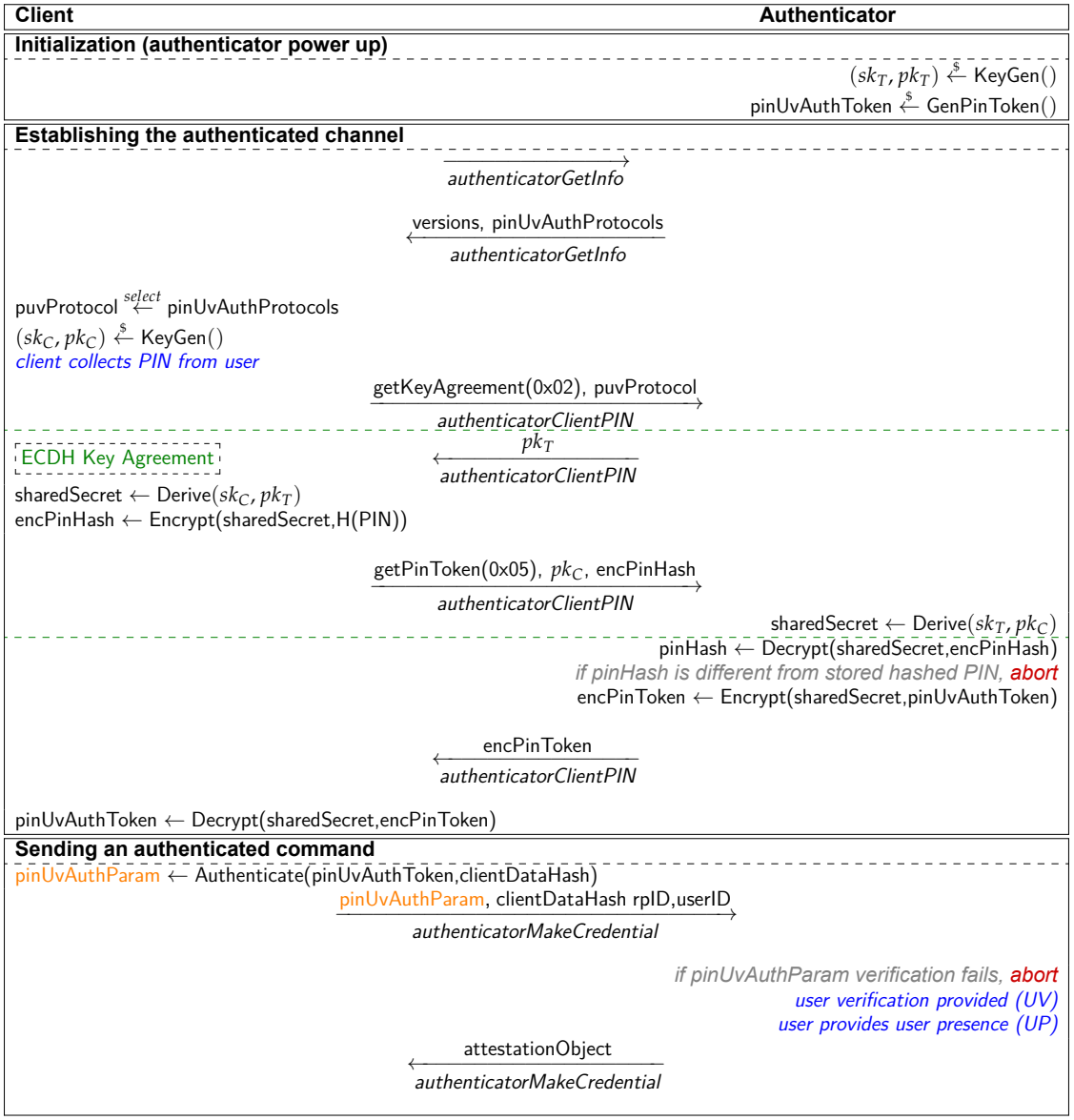


Figure 3.3: The (simplified) CTAP2 protocol. The green section represents the flow of the ECDH key exchange, starting with the return of the token's public key pk_T to the client. Parts in orange are related to command authentication. The blue text represents the user's actions for providing *user verification* and *user presence*.

for creating credentials and obtaining assertions from the authenticator. Figure 3.3 illustrates this (simplified) flow of execution, which we separate in three steps: powering up the authenticator, establishing an authenticated channel, and sending an authenticated command (note that this is a logical separation and simplification of the protocol's flow, and not part of the specification).

3.4.2. Key CTAP2 concepts

Before describing the CTAP2 flow from Figure 3.3, it's important to introduce some key concepts that are essential for understanding the protocol: the PIN/UV Auth Protocol, the

pinUvAuthToken, and authenticator commands.

The PIN/UV Auth Protocol

The PIN/UV Auth Protocol is an abstract interface for several low level cryptographic primitives and operations performed during the CTAP2 flow, such as encrypt, decrypt, authenticate, verify, encapsulate, decapsulate, among others. Each abstract function is then instantiated with a concrete implementation. The concept of the PIN/UV Auth Protocol was introduced with CTAP 2.1, where two instantiations were introduced: PIN/UV Auth Protocol 1 and PIN/UV Auth Protocol 2.

In CTAP 2.0, this concept hadn't been introduced yet, and thus there existed only one implementation of these cryptographic functions, although there were already plans for introducing more instantiations in future versions (which is why the client still provides the puvProtocol parameter seen in the diagram, which in CTAP 2.0 must always be 1). With the release of CTAP 2.1, these implementations correspond to the PIN/UV Auth Protocol 1 instantiation, which may be considered as a legacy instantiation that can be used by CTAP 2.1 clients and authenticators when communicating with others that only support CTAP 2.0. The PIN/UV Auth Protocol 2 instantiation offers a different implementation for several of the interface's functions, and is usually the preferred instantiation to use if supported by both the client and the authenticator.

Each instantiation has a state, which maintains the ECDH key pair, the derived shared secret and, for the authenticator, the pinUvAuthToken. As a consequence, each authenticator generates and maintains one pinUvAuthToken and one ECDH key pair per supported instantiation.

We revisit the PIN/UV Auth Protocol and its instantiations in Chapter 5.

The pinUvAuthToken

The pinUvAuthToken (or pinToken in CTAP 2.0) is a central element of CTAP2. At its core, it is a random 16 or 32 byte cryptographic key (if using PIN/UV Auth Protocol 2, then it can only be 32 bytes long) used to produce HMAC authentication tags. Since CTAP 2.1, the pinUvAuthToken also maintains several state variables, such as the relying party with which it is associated, a set of permissions, and a flag that states if it is still in use or not.

A detailed explanation of how pinUvAuthToken permissions work can be consulted in Appendix B but, in a nutshell, when a client requests a pinUvAuthToken from the authenticator, it must state for what type of operations said pinUvAuthToken will be used. For example, if the client wants to send a registration request to the authenticator through the authenticatorMakeCredential command, it must first obtain a pinUvAuthToken with the mc permission (for *make credential*). To authenticate a authenticatorGetAssertion command, it must instead request the ga permission (for *get assertion*).

For the mc and ga permissions specifically, the client must also provide the RP ID with which the pinUvAuthToken and its permissions will be associated. Then, this pinUvAuthToken can only be used for creating credentials or getting assertions for that specific relying party.

Authenticator commands

The CTAP2 protocol specifies an API of commands to interact with an authenticator. A command here may be seen as a function, in the sense that it can receive input parameters, triggers a series of operations that are executed inside the authenticator, and can output return values. A client communicates with an authenticator by issuing commands, passing the necessary parameters as input, and obtaining the authenticator's response.

In Figure 3.3, each right arrow represents a command, with the name of the command below and the most important parameters above the arrow. Left arrows correspond to the values returned by the authenticator in response to the received commands.

In total, CTAP2 specifies 11 commands, but in this work we are mostly interested in four: authenticatorGetInfo, authenticatorClientPIN, authenticatorGetAssertion and authenticatorMakeCredential.

The authenticatorGetInfo command is used to obtain information from the authenticator, such as the FIDO protocol versions and PIN/UV Auth Protocol instantiations it supports, but also about the support for discoverable credentials, biometric authentication or pinUvAuthToken permissions. The client uses this command to obtain critical information on what operations can be requested from the authenticator. For example, if it only supports PIN/UV Auth Protocol 1, then the client must not request to obtain the pinUvAuthToken using PIN/UV Auth Protocol 2.

As mentioned before, a client invokes the authenticatorMakeCredential and authenticatorGetAssertion commands to request the creation of a new credential or an assertion signature, respectively. These are complex commands, and a detailed account of their behaviour can be consulted in Appendix B. For this description, however, the most important aspect is that they must be authenticated using the pinUvAuthToken. We explain how this is performed in Subsection 3.4.3.

Finally, the authenticatorClientPIN command is used to perform a myriad of different operations, such as obtaining an ECDH public key from the authenticator, setting or changing the authenticator's stored PIN (hash), and obtaining a pinUvAuthToken. Each operation can be requested by providing a specific subCommand parameter as an input to authenticatorClientPIN. This can be seen in Figure 3.3, where the client first sends a authenticatorClientPIN command with the getKeyAgreement subcommand (the subCommand parameter is set to 0x02) and afterwards with the getPinToken (0x05) subcommand. If, for example, the client wanted to set a PIN on the authenticator instead, to be used in future runs to obtain a pinUvAuthToken, it would use the authenticatorClientPIN command with the setPIN (0x03) subcommand (see Appendix B for details on the setPIN and changePIN subcommands).

3.4.3. CTAP2 flow description

With the key CTAP2 concepts covered, we can now describe the flow of execution of a typical CTAP2 execution between a client and an authenticator, as shown in Figure 3.3.

Initialization (authenticator power up)

The initialization step shown in the diagram corresponds to the action of powering up the authenticator. For USB authenticators specifically, which are the focus of this work, this is simply the process of plugging in the authenticator into one of the computer's USB ports, which immediately starts two procedures (per PIN/UV Auth Protocol instantiation): the generation of an Elliptic Curve Diffie-Hellman (ECDH) key pair, and the generation of the pinUvAuthToken. The ECDH key pair is used later for performing the key exchange necessary to establish the authenticated channel, but it is still generated on power up, before any interaction with the client occurs.

Establishing the authenticated channel

After receiving the necessary information for credential generation or for obtaining an assertion signature, the client must perform the appropriate request to the authenticator through the authenticatorMakeCredential or authenticatorGetAssertion commands. However, because these commands need to be authenticated, it first needs to obtain the pinUvAuthToken from the authenticator. Therefore, the establishment of an authenticated channel in CTAP2 is simply the process of securely sending the pinUvAuthToken from the authenticator to the client. As mentioned above, this is done through the authenticatorClientPIN command.

The procedure usually starts with the client requesting information about the authenticator through the aforementioned authenticatorGetInfo command. Depending on the received response, it will then choose the parameters to send to the subsequent commands. One of the most important is the PIN/UV Auth Protocol instantiation that will be used throughout the entire procedure, which the client chooses from the list of supported instantiations received from the authenticator. Additionally, it also generates its own ECDH key pair, and collects the user's PIN (through a prompt in the browser's interface, for example), even though this information is not yet sent in the first authenticatorClientPIN command[8].

To start the ECDH key exchange, the client sends the authenticatorClientPIN command with the getKeyAgreement subcommand to the authenticator, with the selected instantiation puvProtocol as a parameter. The authenticator, upon receiving the request, returns its ECDH public key pk_T, previously generated on power up, for the PIN/UV Auth Protocol instantiation requested by the client.

When the client receives the authenticator's public key, it derives the shared secret key using its private key sk_C (through the PIN/UV Auth Protocol function encapsulate) and uses it to encrypt the hashed user PIN, producing a ciphertext encPinHash. Then, it sends its own ECDH public key pk_C and encPinHash to the authenticator through another authenticatorClientPIN method, with the getPinToken subcommand[9].

Note that this command corresponds to the transmission of the client's public key and of the user's hashed PIN. Therefore, the authenticator first derives the shared secret using

[8]The specification requires the PIN to be sampled before the shared secret is obtained, to avoid having to generate a new one if the user is using an NFC authenticator and removes it while typing the PIN.

[9]There are, in total, 3 subcommands for obtaining a pinUvAuthToken. We use getPinToken here for simplicity, but there are also getPinUvAuthTokenUsingPinWithPermissions, and getPinUvAuthTokenUsingUvWithPermissions. A detailed description of each subcommand can be consulted in Appendix B.

its private key sk_T and then decrypts encPinHash, obtaining the hashed PIN. This is then compared to the hashed PIN that is stored in the authenticator, which for this flow we assume was previously set using the setPIN subcommand. If the verification succeeds, the authenticator encrypts the pinUvAuthToken with the same shared secret key used to encrypt the hashed PIN, producing a new ciphertext encPinToken and sending it to the client. If it fails, the authenticator returns an error, and the process aborts.

At this point, the client decrypts the received ciphertext and obtains the pinUvAuthToken, and the authenticated channel is established.

Sending an authenticated command

The last step of the CTAP2 flow is sending a authenticatorMakeCredential or authenticatorGetAssertion command to the authenticator. The command itself is not relevant here, because the authentication procedure is the same for every command that must be authenticated.

In CTAP2, authenticating a command is the process of calculating an HMAC tag on the clientDataHash, using the pinUvAuthToken as the key. The resulting tag is called the pinUvAuthParam, and is sent as a parameter of the command. This can be seen in Figure 3.3, colored in orange, where the authenticatorMakeCredential command is sent to the authenticator with the pinUvAuthParam as a parameter, alongside the usual clientDataHash, rpID and userID.

The authenticator, upon receiving this information, generates its own HMAC tag on the received clientDataHash and compares it with pinUvAuthParam. If the tags are different, it means the authentication failed, and an error is returned. If they are the same, then authentication was successful and the procedure continues. Then, at some point before producing the credential, the authenticator requests the user for proof of user presence, and, if it is provided, returns the response to the client.

Note that this procedure is considered to have user verification, because the pinUvAuthParam was produced using a pinUvAuthToken that was obtained by sending the user's (hashed) PIN to the authenticator.

3.5. Summary

In this Chapter, we introduced the FIDO2 protocol. We started by briefly introducing its historical context, flow of execution and some core concepts. Then, we introduced WebAuthn and CTAP2, the two (sub)protocols that make up FIDO2, and provided a brief description over the most important aspects of each protocol. We focused these descriptions on the use cases that are more relevant for our work, with more detailed explanations in Appendices A and B.

In Chapter 4, we highlight two design choices in CTAP2 that make it vulnerable to specific types of attacks. We then show that these attacks can be used in practice to negatively impact FIDO2's expected security guarantees, by providing our proof-of-concept implementations and deployment strategy.

4. Practical attacks on FIDO2

The FIDO2 protocol aims to strengthen the security of users online, with both WebAuthn and CTAP2 employing mechanisms to achieve this objective. In WebAuthn, the client communicates with the server using a secure channel, and validates information it receives from the server and the authenticator to ensure it conforms with the protocol's requirements. In CTAP2, an authenticator generally creates a credential or produces an assertion signature over a relying party's challenge *only* if the client issuing the command is authorized by its owner, the user, using a PIN or some form of biometric authentication to obtain the pinUvAuthToken[1]. This process requires setting up an encrypted (but notably not authenticated) channel for the client to transmit the hashed PIN to the authenticator (if this is the chosen verification mechanism) and for the authenticator to transmit the pinUvAuthToken back to the client. This is achieved by encrypting the hashed PIN *and* the pinUvAuthToken with a shared secret encryption key, derived from an Elliptic Curve Diffie-Hellman key exchange.

However, some FIDO2 design choices, specifically in CTAP2, can expose the protocol to certain types of attacks that can break some of its expected security guarantees. We explain what these design choices are in Section 4.1, and briefly explain how they might be exploitable by an attacker that can interfere with messages exchanged by clients and authenticators. In Sections 4.2 and 4.3 we introduce concrete attacks that leverage these design choices from CTAP2, detailing how they work and describing how they may be used to launch larger attacks on FIDO2. In Section 4.5 we introduce our proof of concept implementation for each attack, and explain our deployment strategy. Finally, in Section 4.6 we end with a summary of our results, pointing out some limitations with our attacks, and with a brief reflection on what this means for the practicality of attacking FIDO2.

At the moment of writing, all design choices we mention are present in every version of CTAP2, which means our attacks can be performed independently of versions and PIN/UV Auth Protocol instances. Nevertheless, there are some differences on how they work depending on the CTAP2 version, which we mention whenever relevant. Also, we use the term pinUvAuthToken instead of pinToken because this is the term used from CTAP

[1] Assuming the relying party enforces or prefers user verification and the authenticator is capable of providing it.

2.1 onwards, but they are usually interchangeable in the context of our attacks. If we use the term pinToken specifically, then we are refering *only* to CTAP 2.0.

4.1. Design choices in CTAP2

4.1.1. Unauthenticated Elliptic Curve Diffie-Hellman exchange

In the WebAuthn protocol, the information transmitted from the client to the server, and vice-versa, is protected against tampering and eavesdropping through TLS. This is possible because the TLS handshake, which performs a key exchange to derive the symmetric encryption keys, is authenticated with the server's certificate. This effectively eliminates the possibility of man-in-the-middle attacks, which are attacks where an adversary interferes with the communication between parties by relaying or modifying the information being exchanged without being detected.

This resistance to active attackers does not translate to CTAP2, because the ECDH key exchange performed to obtain the pinUvAuthToken is unauthenticated. An attacker that is capable of positioning itself between the communications of a client and an authenticator can, in theory, perform a man-in-the-middle attack by replacing the ECDH public keys exchanged between the two parties with keys of its own. This enables the attacker to derive two shared secret keys, one with each party, and then encrypt or decrypt any ciphertext sent through this channel. Each party is unaware of the attack, because neither ECDH public key is authenticated. We expand on this type of attack, and how it can be leveraged to perform an *impersonation attack* on FIDO2 in Section 4.2.

4.1.2. Unauthenticated response from the authenticator

When an authenticator receives a command, it can verify that it came from a previously authorized client that obtained the pinUvAuthToken, by calling the PIN/UV Auth Protocol's verify function to validate the pinUvAuthParam parameter. In fact, since CTAP 2.1, where every pinUvAuthToken given to a client is always unique, there can only be one client that could have sent that command, because it is the only party that has a valid pinUvAuthToken (unless an attacker obtains it in some way, for example through a man-in-the-middle attack).

However, the authenticator's response to the client is *not* authenticated. The client validates the response syntactically and semantically, but not cryptographically. This is true in every version of CTAP2, and can lead to a certain type of attacks where an adversary intercepts the last message (an authenticatorMakeCredential command, for example) from

a client to a user's authenticator and sends it to its own authenticator instead. The response produced by the attacker's authenticator is then returned to the client, which has no form of knowing that it did not come from the user's authenticator. We expand on this type of attack, and how it can be leveraged to perform a *rogue key attack* on FIDO2 in Section 4.3.

4.2. Impersonation Attack

The unauthenticated ECDH key exchange theoretically allows an adversary to perform a man-in-the-middle attack and obtain the pinUvAuthToken from the authenticator. This is an attack at the CTAP2 level, and gives the adversary the ability to send authenticated commands to the authenticator. This on its own is already significant, but is not sufficient to attack a user using FIDO2 to register or authenticate with a relying party online. However, it can be used as a building block for an *impersonation attack*, where an adversary leverages the pinUvAuthToken to authenticate with a specific relying party on behalf of a user (i.e., impersonate the user) unnoticed.

We describe this impersonation attack in two parts: the execution of the man-in-the-middle attack to obtain the pinUvAuthToken, and how an attacker can leverage the obtained pinUvAuthToken to perform a WebAuthn authentication ceremony with a relying party unnoticed, using the user's authenticator. We consider cases where the client uses a PIN, although this attack can be performed with other forms of user verification as long as they still employ the same unauthenticated key exchange.

4.2.1. Performing a man-in-the-middle attack on CTAP2

Fig. 4.1 contains a simplified diagram of the man-in-the-middle attack we implemented, the first part of our impersonation attack.

As detailed in Chapter 3 and Appendix B, to start a key exchange with the authenticator and obtain the shared secret key the client must invoke the authenticatorClientPIN command with the getKeyAgreement subcommand (cmd in the figure), which receives no additional parameters other than the PIN/UV Auth Protocol instance[2]. It does not send its own ECDH public key to the authenticator with this subcommand, so the attacker does not need to interfere with this request. The authenticator's response, however, does include its ECDH public key pk_t. When the attacker receives this response, it saves the received

[2]The man-in-the-middle attack is not conditioned by which PIN/UV Auth Protocol instance is chosen, because both instances as of CTAP 2.2 perform the same unauthenticated key exchange.

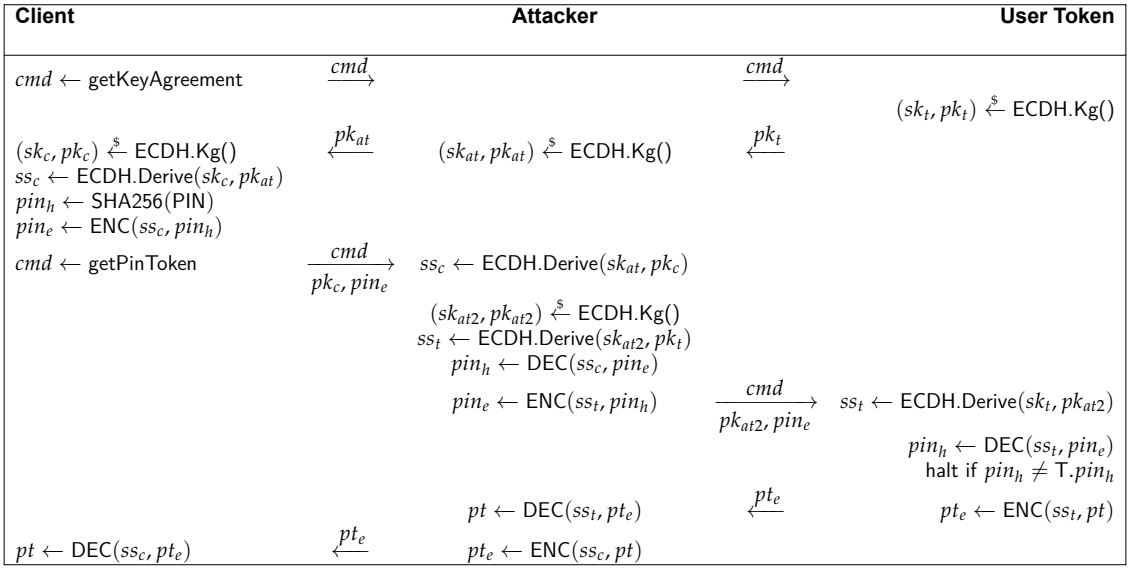


Figure 4.1: Simplified diagram of the man-in-the-middle attack, as the first step of the impersonation attack. Right arrows are authenticatorClientPIN commands, and left arrows the respective responses. Functions ENC and DEC correspond to the PIN/UV Auth Protocol functions encrypt and decrypt.

public key pk_t (for later usage), generates a new ECDH key pair (sk_{at}, pk_{at}), saves the private key sk_{at} and replaces the authenticator's public key pk_t in the response with its own pk_{at}, before forwarding it to the client. On its side, the client derives a shared secret ss_c with the attacker's public key pk_{at} instead of the intended pk_t, collects the user's PIN, hashes it using (the leftmost 16 bytes of) SHA-256 to produce a PIN hash pin_h and encrypts it with ss_c to produce a ciphertext pin_e. Next, it invokes a new authenticatorClientPIN command with a subcommand to request the pinUvAuthToken, which contains the client public key pk_c and the encrypted PIN hash pin_e. We specify the getPinToken command in Figure 4.1 for simplicity, but for the purposes of this attack the client could also invoke getPinUvAuthTokenUsingPinWithPermissions[3], and the attack would work in the same way. The attacker intercepts the getPinToken subcommand and modifies the information to be sent to the authenticator. It derives the same shared secret ss_c, using its own ECDH private key sk_{at} and the client's ECDH public key pk_c, and generates a new ECDH key pair (sk_{at2}, pk_{at2}). The encrypted PIN hash pin_e is decrypted with the shared secret ss_c, and then re-encrypted with the new shared secret ss_t, derived from the ECDH private key sk_{at2} and the previously saved authenticator ECDH public key pk_t. Finally, the attacker sends the new encrypted PIN hash pin_e and the new ECDH public key pk_{at2} to the authenticator.

[3] Or getPinUvAuthTokenUsingUvWithPermissions, assuming the authenticator supports a form of user verification other than a PIN. In this case, the client does not send pin_e in the command, and thus the attacker does not need to perform this decrypt/re-encrypt step.

This is not detected by the authenticator, because the client's public key pk_c is not authenticated, and the authenticator does not know before hand to expect pk_c. It will derive the shared secret ss_t, verify the received PIN hash pin_h with the saved PIN hash (T.pin_h), and respond with the encrypted pinUvAuthToken (pt_e). To complete the flow and go undetected, the attacker decrypts pt_e with ss_t, at which point it obtains the pinUvAuthToken pt, re-encrypts with ss_c, and sends the new ciphertext back to the client.

4.2.2. Impersonating the user on a relying party

In the second part of the impersonation attack, showcased in Fig. 4.2, we explain how an attacker can leverage the pinUvAuthToken to authenticate with a target relying party using the user's authenticator, without needing to know the PIN or perform user verification on that authenticator, and with a chance of being undetected by the user.

During an authentication session, the client obtains the pinUvAuthToken with the user's consent, and uses it to authenticate an authenticatorGetAssertion command. In normal circumstances, only this client (or clients if in CTAP 2.0, since multiple clients can obtain the same pinToken) should have that ability. However, if a man-in-the-middle attack is performed, this no longer holds, because an attacker who also possesses the pinUvAuthToken can issue its own authenticated authenticatorGetAssertion command, which the user's authenticator will accept, completely bypassing the intended user verification. However, this is not enough on its own, because any assertion that can be accepted by a relying party must also involve proof of user presence, performed by the user during the authentication ceremony, and proven to the relying party with the assertion signature over the authData structure with the UP bit set to 1. Providing proof of user presence is *not* optional.

This poses a challenge, because it means the attack can only be performed if the user physically interacts with the authenticator, and a responsible user might suspect that something is wrong if its authenticator suddenly requests proof of user presence outside the context of an authentication ceremony (or any other operation that also requests user presence). The solution to this challenge, then, is to perform the attack when the user is already expecting to provide proof of user presence, such as when it attempts to log in with a specific relying party.

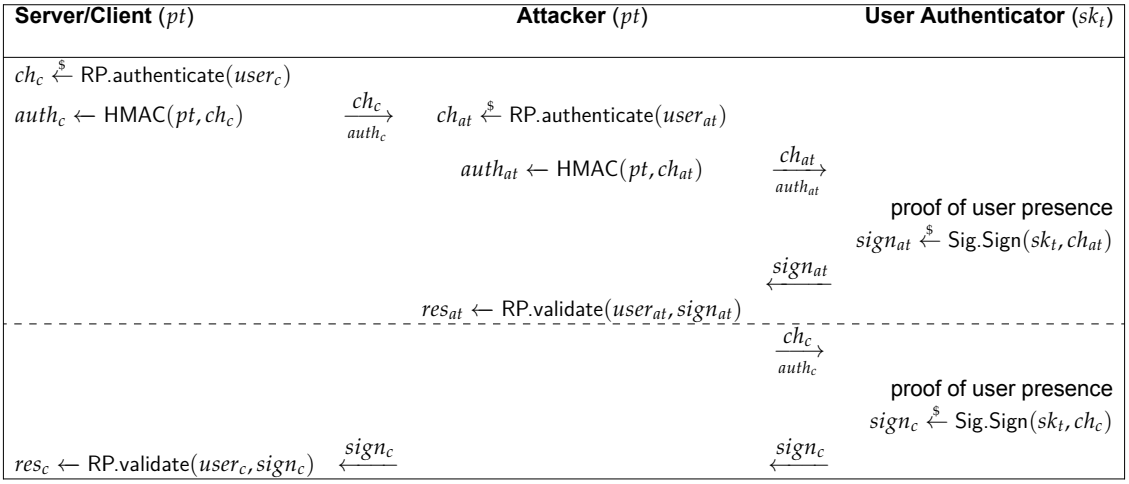


Figure 4.2: The impersonation attack. The attacker authenticates the user with a target relying party using the previously obtained pinUvAuthToken (pt). Right arrows are (authenticated) authenticatorGetAssertion commands, and left arrows the respective responses. RP.authenticate($user$) represents the start of a WebAuthn authentication ceremony, and outputs a challenge. RP.validate($user, sign$) represents the relying party validation of the assertion signature, and produces a boolean result.

Hijacking an authentication ceremony

The impersonation attack we describe takes advantage of a user's legitimate request to start an authentication ceremony by starting a new parallel authentication ceremony with a target relying party. This can be the same relying party with which the user is authenticating or a different one entirely. A user might have more than one account in the same service, or have another account on a different service, to which the attacker is trying to gain access.

The attack comprises the following steps: (i) intercepting a authenticatorGetAssertion command with the legitimate challenge ch_c; (ii) starting a new authentication request with the target relying party (RP.authenticate); (iii) sending the new authenticatorGetAssertion command to the user's authenticator, with the new challenge ch_{at}; (iv) receiving the authenticator's response with the (malicious) assertion signature $sign_{at}$; and (v) sending the response to the target relying party (RP.validate).

Note that the user will confirm presence between steps (iii) and (iv) because it believes that the authenticator is answering the honest client request when, in fact, it is answering the attacker's request. Also note that, in order to attempt to complete the original assertion, and thus allow the user to authenticate with the original relying party to improve the chances that this attack goes unnoticed, the attacker can still send the original challenge ch_c to the authenticator to produce an assertion signature $sign_c$, which the client will send back to the relying party (below the dashed line in Figure 4.2). This requires the user to

prove presence again, which might increase suspicion. Most users might, however, simply dismiss this double user presence test as a malfunction or a bad button press, akin to failing a fingerprint scan in a smartphone. Nevertheless, even if the user is suspicious, it is the first proof of user presence that actually enables the attacker to perform the attack, so by the time the user realizes something is wrong the attacker has already obtained access to its target account.

If providing proof of user presence twice is not desirable, then the initial approach where the user's legitimate authentication attempt fails (i.e., the part below the dotted line in Figure 4.2 is not performed) is still successful. It might assume that the login failed due to some other factors, and simply try again. In the meantime, the attacker already obtained access to the target account.

We stress once more why this attack must be performed in this way. Taking user interaction into consideration when executing the impersonation attack is crucial. If user presence confirmation was not required, then one could launch this attack anytime after getting the pinUvAuthToken, as long as the user's authenticator was still plugged in and the pinUvAuthToken hadn't changed in the meantime.

The impact of CTAP2 permissions

In CTAP 2.0, the step in the impersonation attack where the attacker allows the user to complete its legitimate authentication ceremony can be performed successfully. This is not true for the newer versions of CTAP2, because of the existence of permissions.

From CTAP 2.1 onwards, it is no longer possible to reuse the same pinUvAuthToken to authenticate two commands in a row, because whenever a command that requires user presence is executed, the pinUvAuthToken associated with the command has every permission except lbw permanently revoked (see Subsection B.2.4 of Appendix B). In these scenarios, the optional step where the attacker sends the original authenticatorGetAssertion command to the user's authenticator in an attempt to still allow the user to login no longer works. Of course, this does not affect the first authenticatorGetAssertion command sent by the attacker, since this corresponds to the first usage of the pinUvAuthToken, and so the attack itself can still be carried out, even if it results in the user always failing its login attempt.

Another aspect worth mentioning concerns the RP ID that is associated with the pinUvAuthToken. The getPinUvAuthTokenUsingPinWithPermissions and getPinUvAuthTokenUsingUvWithPermissions subcommands receive the rpId parameter with the RP ID to be associated with the pinUvAuthToken and its permissions (see Subsection B.2.4 of Appendix B). If the user starts an authentication ceremony with a given relying party, this relying party might become associated with the pinUvAuthToken if the client invokes one of these two subcommands, which in turn means only authenticatorGetAssertion commands for that specific RP ID are accepted. This does not mean the attack is not possible, but it does restrict it to impersonations within the same relying party.

This limitation can be solved when performing the man-in-the-middle attack, because the attacker can change the rpId parameter in one of these subcommands to the rpId of the relying party with which it wishes to authenticate. This can guarantee that the obtained pinUvAuthToken will be associated with the target relying party and, therefore, that the attack is still successful. The user's login attempt will fail if the pinUvAuthToken is associated with a different relying party, but it would have failed regardless because of the revoked permissions mentioned above. Another possible strategy, which we have not tested, might be to simply downgrade the client's authenticatorClientPIN command to use the getPinToken subcommand instead, which does not have the rpId parameter. In this situation, the pinUvAuthToken is given default mc and ga permissions, and the RP ID associated with it is only set *during* the authenticatorGetAssertion command.

4.2.3. Obtaining the PIN hash

During this man-in-the-middle attack, an attacker can also obtain and store the PIN hash pin_h received from the client. This is particularly noteworthy, first because it might enable the attacker to try to obtain the PIN later by performing *offline* brute force or dictionary attacks, but also because once the decrypted PIN hash is in its possession, an attacker may use it to obtain a new pinUvAuthToken from the authenticator later, completely undetected. It can perform a new ECDH key exchange with the authenticator, obtain a new shared secret, encrypt the previously obtained PIN hash and send it in a authenticatorClientPIN command to obtain the pinUvAuthToken. This interaction does not require user presence, which in this type of authenticatorClientPIN command (i.e., that requires a PIN instead of biometric authentication) is only requested by the authenticator when responding to authenticatorMakeCredential or authenticatorGetAssertion. Nevertheless, limitations on the timing of this attack are still in place: an attacker can only use the PIN hash to obtain a

pinUvAuthToken from the authenticator when it is plugged in, and an impersonation can only take place if the user approves of a relying party request by proving user presence. In other words, an attacker may be able to obtain another pinUvAuthToken, but if it wishes to fool the user into approving a relying party request then it still must intercept a request the user is already expecting.

The impact on pinUvAuthToken permissions An attacker that performs this kind of attack can also overcome the limitations that come from CTAP2 permissions, because it becomes possible for it to always obtain a new pinUvAuthToken, for whatever relying party it wishes, and with whatever permissions it chooses.

PIN vs PIN hash Note that, in this attack, the attacker never learns the actual PIN, but also that in the context of obtaining a pinUvAuthToken, the distinction between the PIN and the PIN hash is irrelevant, since the authenticator only ever stores the latter. Nevertheless, an attacker learning the PIN may have consequences on a broader level than just FIDO2. The user's PIN might be the same as (or a variant of) other passwords it might employ in different online services, and so by discovering it the attacker could put those accounts at risk too.

4.3. Rogue Key Attack

The rogue key attack takes advantage of the fact that, during registration, neither the relying party nor the client can usually detect if a response to a authenticatorMakeCredential command came from the intended authenticator, i.e., the authenticator the user intends to use to perform the registration, and to which the client sends the authenticatorMakeCredential command.

During an authentication ceremony, the relying party expects the issued challenge to be signed using a credential that was previously registered. When credentials are bound to the authenticator on which they were generated, only that authenticator can correctly sign the challenge. Therefore, if the signature is valid, the relying party knows that the authentication ceremony was performed using the intended user's authenticator (i.e., the authenticator that performed the registration). However, this is not the case in registrations. During a registration run, the relying party does not know in advance the exact authenticator the user will be using. It can potentially know what kind of model was used when it receives the attestation signature, and may even employ some policies that restrict the usage of certain models of authenticators *a priori*, but generally it does not know

which authenticator will be used exactly[4]. Therefore, we cannot rely on the relying party's ability to ensure that the authenticator that responds to a credential creation request is the intended user's authenticator.

On the client side, as mentioned in Section 4.1, the client has no guarantee that a response it receives comes from the authenticator to which it sent an authenticated command.

Indeed, an attacker is theoretically able to hijack a registration run because (i) registration requests are sent without encryption to the authenticator, which exposes the clientDataHash to be signed by the authenticator; and (ii) responses sent from the authenticator back to the client are not bound to the authenticated channel established with the pinUvAuthToken, nor are they encrypted. The attacker can hijack the registration session by replacing the credential public key returned by the user's authenticator with its own credential public key, possibly (but not necessarily) generated by another hardware authenticator. Because the response is not encrypted or authenticated, the client has no reason not to accept the attacker's credential public key. Therefore, we also cannot rely on the client to guarantee the response comes from the intended authenticator.

4.3.1. Attack description

Figure 4.3 illustrates our rogue key attack. Similarly to the impersonation attack, an attacker must be capable of observing and intercepting messages exchanged between client and authenticator, as well as injecting new ones. When compared to the impersonation attack, however, the required adversary power is much weaker. It does not need to extract the pinUvAuthToken, and therefore does not interfere with the key exchange. Instead, it just observes the authenticatorMakeCredential request sent to the authenticator, which reveals the clientDataHash that contains the Webauthn challenge issued by the relying party, and for which it is expecting an answer.

At this point, the attacker can block the authenticator response (or simply not deliver the request) and inject a malicious response that allows it to register its own credential public key, which we call the *rogue key*. The rogue key can be generated offline without a hardware authenticator as long as the attacker somehow has access to a valid attestation key that the server will accept, or if the server doesn't require attestation with an attestation key. Otherwise, it must be generated on the fly by an authenticator the attacker controls. Note that if this is the case, and the attacker uses an untampered authenticator,

[4]We exclude enterprise attestation for this attack (see Appendix A).

Server/Client (pt_c)		**Attacker** (pt_{at})		**Authenticator**
$ch \leftarrow$ RP.register$(user)$				**Attacker** (ak_{at})
$auth_c \leftarrow$ HMAC(pt_c, ch)	$\xrightarrow[auth_c]{ch}$	$auth_{at} \leftarrow$ HMAC(pt_{at}, ch)	$\xrightarrow[auth_{at}]{ch}$	proof of user presence
$res_{at} \leftarrow$ RP.validate$(user, sign_{at})$	$\xleftarrow{sign_{at}}$	$\xleftarrow{sign_{at}}$	$\xleftarrow{sign_{at}}$	$sign_{at} \xleftarrow{\$}$ Sig.Sign(ak_{at}, ch)
				User (ak_t)
			$\xrightarrow[auth_c]{ch}$	proof of user presence
			$\xleftarrow{sign_c}$	$sign_c \xleftarrow{\$}$ Sig.Sign(ak_t, ch)

Figure 4.3: The rogue key attack. Right arrows are (authenticated) authenticatorMakeCredential commands, and left arrows are the respective responses. RP.register and RP.validate denote requesting and validating a credential registration with the relying party. RP.register$(user)$ produces a challenge and RP.validate$(user, sign)$ produces a boolean result.

it will need to confirm user presence, because the authenticator will include in the attestation signature if a test of user presence was performed in the UP flag from the authData structure.

Figure 4.3 shows the full sequence when the rogue key is generated on the fly. In our implementation of the attack, we emulate it by placing the attacker's authenticator in the same machine as the user's, but clearly a network request could be placed to an adversarially controlled machine to the same effect if needed. We now describe each step in the attack in more detail.

The first step is to intercept the authenticatorMakeCredential command. As was pointed out, this command is authenticated but not encrypted, which means that one can recover all the information needed to construct a well formed response, including the challenge ch (which represents the clientDataHash). At this point, if using an adversarially controlled hardware authenticator, the attacker simply follows the protocol to issue an authenticatorMakeCredential command using the parameters provided in the legitimate request, including the challenge ch. The attacker's authenticator then produces an attestation signature $sign_{at}$ on the challenge ch using its own attestation private key ak_{at}. This signature is returned to the attacker and forwarded to the client, which will not be able to detect that the new credential does not come from the user's authenticator, and completes the registration flow with the relying party, also unaware of what happened.

Moreover, to make sure the user does not detect the attack, one may indeed send the original authenticatorMakeCredential request to the user's authenticator, who will be prompted to confirm presence. The authenticator's response $sign_c$ is then discarded.

Note that if the attacker is operating remotely, and thus its authenticator is on another machine, and if the original authenticatorMakeCredential command is sent to the user's authenticator, this process looks completely normal to the user, who will think the registration was successful with its own authenticator. Only when trying to perform an authentication run will the user find out that its authenticator was indeed not the one that performed the registration ceremony.

4.4. Double registration attack

The rogue key attack allows the adversary to register its own authenticator *instead* of the user's. This means the user will not be able to login on the account it supposedly just created or registered, which might raise suspicion that an attack was carried out. On the other hand, if the attacker was able to register the user's authenticator alongside its own, the user would not suspect that anything strange had happened because it would still be able to login to its account. As it turns out, this double registration attack can indeed be carried out, by combining properties from the rogue key and the impersonation attacks.

Initially, the attacker performs the man-in-the-middle attack as detailed in Section 4.2, obtaining the pinUvAuthToken before the authenticatorMakeCredential command is sent by the client. Then, it redirects the command to register its own authenticator instead of the user's, just like in the rogue key attack. Unlike with the rogue key attack, however, the attacker must now also interact with the relying party and request a new key registration for the same user[5]. This request will result in a new authenticatorMakeCredential command, which the attacker can issue to the user's authenticator because it has previously obtained the pinUvAuthToken. The only action needed from the user in this case is to provide user presence[6]. If the whole attack is successful, registration of the rogue key may go undetected by the user, as future authentications will succeed.

Note, however, that the restrictions regarding pinUvAuthToken permissions from the impersonation attack still apply.

[5]Many services allow the registration of more than one credential for the same user as a form of backup, in case a user loses one of its authenticators.

[6]If the user is using a CTAP 2.1 authenticator with biometric authentication, like a fingerprint sensor, it might not even need to provide user presence at that time, because it might have already been provided when obtaining the pinUvAuthToken. In that case, the user does not have to do anything to authorize the registration process.

4.5. Implementation and deployment strategy

So far, we have explained how an attacker can *theoretically* take advantage of two design choices from CTAP2 (and on the overall FIDO2 protocol) to impersonate a user online or register its own authenticator in a user's account. However, this raises a natural question. Are these attacks relevant for practical applications? In other words, is the adversarial model where an attacker can eavesdrop *and* manipulate communications between clients and authenticators realistic?

In this Section, we answer these questions in the affirmative, by introducing our proof-of-concept implementation and deployment strategy for these attacks. Note that, while these attacks are different, the threat model is the same: the attacker must position itself between the client and authenticator's communications, and must be capable of intercepting, changing and forwarding messages in both directions. Therefore, in this Section we only describe the implementation logic related to the eavesdropping and manipulation of USB level messaging, as well as how the malicious code can be deployed on the user's machine, and refer to our implementation[7] of the attacks for more details.

We implemented our attacks for USB FIDO2 authenticators only (arguably the most used type of authenticator on laptops and desktops), but they can naturally be extended to other transport mechanisms. Our goal is to show that it is realistic to assume a motivated attacker *can* exploit the aforementioned design choices of FIDO2. With this in mind, we prioritized making our implementation of the attacks easily deployable, with no privileged permissions required, and without modifying the client (browser) used to perform the registration or authentication ceremonies. We use one of many possible strategies (detailed below) to achieve this, using a minimally intrusive piece of malware: the simple goal for this malware is to ensure that, at some point, the user is interacting with a browser window in a context where the browser's access to the system library offering USB communications is performed via a malicious wrapper that the attacker controls. Since messages are manipulated at the USB level, this approach works with any other browser implementation that follows the same USB communication pattern. This is mostly fixed by CTAP2 (CTAPHID), although the exact sequence of system calls may vary.

4.5.1. Software stack

Our implementations were developed and tested on Linux, using C and Python. We mostly relied on three libraries to perform various operations on the authenticator and the

[7] `https://github.com/esquivel71/FIDO2-Attacks`

data it exchanges with the client application. First, we used libfido2,[8] an open source library from Yubico that provides an API for performing operations on the authenticator. We needed to (minimally) adapt certain functions to our purposes. Second, we used the libcbor library to serialize and deserialize CBOR messages (recall that every CTAP2 command and response encodes its parameters in a CBOR map), as well as to manipulate those structures in memory.[9] The third library we use is libcrypto, for multiple cryptographic operations such as AES encryption and decryption and Elliptic-Curve Diffie-Hellman operations. We note that all of this code could be simply replaced with an optimized dedicated implementation to obtain malware with a minimal code footprint.

The malicious wrapper that intercepts the communications between the client and the authenticator and performs the specific attack is implemented as a shared library, using a technique called *function hooking*, which we explain next.

4.5.2. Hooking system calls

Function hooking consists of creating a wrapper with additional code around a preexisting function to extend its functionality at runtime. When set, the wrapper can intercept the function call and then execute that additional code before, after or instead of executing the function. This can be achieved without having to change system code, making it a great tool to use for our attacks. We briefly explain how.

On Linux, applications like libfido2 and Google Chrome will communicate with a USB authenticator through the *read* and *write* system calls. The application first obtains a file descriptor *fd* for the authenticator (through the *open* system call) and then sends information (i.e., issues a command) to that descriptor using *write* and receives information (i.e., the response) through *read*. Our attack is carried out by hooking these two system calls, by creating two functions with identical signatures (the names, return types and parameters exactly match the function signature of *read* and *write* in the glibc library) and implementing the attack logic inside them.

Creating a basic function hook for these functions is a simple procedure. It starts by creating a new C file with two functions: *read* and *write*. Our implementation of these attacks relies on the fact that these functions will be called *before* the actual *read* and *write* system calls, so they must have the exact same signature. Figure 4.4 shows a very simple implementation of these hooks. We can see the function signature for each of

[8] https://github.com/Yubico/libfido2.
[9] Documentation available from https://libcbor.readthedocs.io.

the system calls we want to hook, and also how we can load the original *read* and *write* functions into memory, by first declaring the variable that will contain the pointer to the original function, and then using the *dlsym* function to find the correct function (symbol) to load into memory. We use the handler rtld_next to inform *dlsym* to find the next occurrence of the symbol, specified by the second parameter. Loading the symbols for the original *read* and *write* functions is essential, because this is how they can be called inside our own definitions to maintain the functionality expected from the application that called *read* or *write*. Not performing this step results in messages from the client or the authenticator not reaching its intended recipient, which might not be desirable. Most of the time, our malware needs to be capable of not only intercepting messages, but also modifying and relaying them to the original recipient, or to a malicious authenticator controlled by the attacker.

```c
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>

/* The function signature must be identical to the signature of the write system call in glibc */
ssize_t write(int fd, const void *buf, size_t count)
{

    /* Declare a variable to contain the pointer to the write function defined in glibc and attribute that pointer */
    int (*real_write)(int fd, const void *buf, size_t count);
    real_write = dlsym(RTLD_NEXT, "write");

    printf("This message will be shown when an application calls the write system call!\n");

    /* This will call the actual write system call as defined in glibc */
    /* Without this our hook will not be able to maintain the functionality of the original write system call */
    return real_write(fd, buf, count);

}

/* The function signature must be identical to the signature of the read system call in glibc */
ssize_t read(int fd, void *buf, size_t count)
{
    /* Declare a variable to contain the pointer to the read function defined in glibc and attribute that pointer */
    int (*real_read)(int fd, void *buf, size_t count);
    real_read = dlsym(RTLD_NEXT, "read");

    printf("This message will be shown when an application calls the read system call!\n");

    /* This will call the actual read system call as defined in glibc */
    return real_read(fd, buf, count);
}
```

Figure 4.4: Simple example of function hooks on the *read* and *write* system calls. Notice the functions signatures that exactly match the signatures of *read* and *write* in glibc.

After developing our hooks, which contain the logic of the attack that is being implemented, the code is built into a shared library, which contains the symbols for our *read* and *write* functions. The goal now is to ensure the client application calls our function hooks instead of the real *read* and *write* from the glibc library. We do this using the ld_preload environment variable.

4.5.3. Intercepting system calls

On Linux, if the path for a given shared library attack.so is present in the ld_preload variable, for a given process, the dynamic linker will attempt to locate and load the symbols that process needs from attack.so *prior* to searching any other libraries. We can use this mechanism to create an environment in which processes, like a browser, are given access to our implementation of these functions instead of the system's implementations. One important aspect to note is that in no point do we need to obtain root access to the user's system, which makes our attacks plausible in a much wider range of real life scenarios.[10]

The simplest form of launching the attack, after the shared library is compiled, is to export its path to ld_preload in the current terminal session, and then launch the client application in that session. The new process inherits the contents of ld_preload and will call our definitions of *read* and *write*. We note, however, that hooking these two system calls will most likely not only affect the process being targeted (the client application), but also other processes launched from the same terminal session. Additionally, functionalities from the client application (most likely a browser) that are not related to FIDO2 may also use *read* and *write*, and may stop working if the function hooks cannot filter out calls that are not meant for the authenticator. For that reason, in our implementation, we always check if a call to *read* or *write* is associated to a FIDO2 device. If not, we just return the result from calling the real *read* or *write* system call accordingly, with no perceivable cost to performance.

4.5.4. Implementation details

Our approach has the aforementioned advantage of being generic with respect to clients and authenticators because it operates at the USB interface level. However, note that intercepting messages at the USB interface level is technically more intricate than using some higher-level API (e.g., a CTAP2 library itself). For example, the fact that each message packet has a fixed size of 64 bytes means that multiple read/write operations have to be performed for a given operation, and that state must be kept by the wrapper to keep track of the higher-level transaction that these messages are implementing. However, since we need to wait for the client to call our implementations of *read* and *write*, we do not keep state information in memory, keeping it instead in the file system. Moreover, when intercepting a message that needs modification, the function hook needs to buffer

[10] For more information on ld_preload, as well as on a system-wide alternative see https://www.baeldung.com/linux/ld_preload-trick-what-is.

it completely, to decode the CBOR-encoded information, modify it, encode it again and relay to the intended recipient. This means that it must sometimes feed false responses to the client, so that it thinks that the previous message was successfully delivered to the authenticator, and issues a new one with the continuation packet. We now describe the process in a bit more detail.

When a client application invokes the *write* system call, our function hook is executed instead, where it first checks if the file descriptor points to a FIDO2 device and, if so, then checks for the existence of a specific *helper* file. This is the file we use to maintain the aforementioned state between system calls, which is only created when an initialization packet is intercepted. If it does not exist, we create it with the appropriate information to maintain state, including, but not limited to, the total length of the CBOR data (which we can calculate from the initialization packet), the CTAP2 command being requested, and an array which will contain the entire CBOR data, i.e., the CTAP2 parameters for the command being requested. We call this field `cbor_payload` in our implementation.

If the file already exists, then we have already intercepted the first packet, and now expect only continuation packets. We load the *helper* file to memory and update, among others, the `cbor_payload` field by appending the CBOR data from the intercepted packet. When the `cbor_payload` length matches the CBOR data total length that was specified in the initialization packet, we know we have received the last packet. At this stage, we have the entire CBOR data sent by the client, and thus can deserialize it, modify any CTAP2 parameters necessary to make the attack work, serialize the new modified CBOR structure, and send it to the authenticator.

Intercepting a *read* system call is almost identical to intercepting a *write* system call. After sending every packet to the authenticator, the client application will try to read the response through the *read* system call. Depending on the response's size, the authenticator may also send it via multiple packets. However, because the response messages we want to intercept are substantially smaller than the requests, and thus have fewer packets, we opted to replace the required parameters directly in each packet. This has the benefit of avoiding sending false *read* responses to the client (because the client might only call *read* to receive a continuation packet after receiving the previous one), which might result in behaviour that is harder to predict.

4.5.5. Weaponization

As previously mentioned, exporting the path of our malicious shared library to ld_preload before launching the client application through the terminal can position our function hooks to intercept the client and authenticator communications and perform the attacks. While this serves as a proof of concept, it doesn't yet demonstrate how easily the attacks can be deployed. A regular user will probably not launch its browser through the terminal right after exporting a malicious library to ld_preload, and even if it does, the attacker still has to somehow download the malicious library into the user's computer. To show that our attacks are relevant in practice, and can be performed by a motivated attacker to obtain access to a user's account, we had to develop a better strategy to deploy them.

Next, we describe an attack vector that can be leveraged to install the malicious library seamlessly, such that the next time the user opens its browser, the malware will quietly wait for it to start communicating with the authenticator, at which point it starts performing the chosen attack.

The LibreOffice macro

It is known that information-stealers, a particular type of malware specialized in stealing credentials, often hook themselves to browser libraries to intercept information typed by the user [34, 35], which means our methodology to deploy our attacks does not deviate too much from relatively common techniques. Nevertheless, the majority of these pieces of malware exploit Windows operating systems, and so we explore how to leverage these attacks to create Linux malware as a proof of concept. Other possibilities of weaponization and attack motivations are discussed in Section 4.6. The attacks presented here require changing the environment associated with user applications, which implies that the attacker must find a way to get the first foothold on the victim's machine. Malicious attachments have always been a prime way to propagate malware, which is why we chose this type of attack vector to install and prepare our malicious shared library.

Following a similar approach that is commonly used in Windows campaigns, we tested the possibility of using a malicious LibreOffice Calc spreadsheet file as the first stage of our malware. LibreOffice is an open-source office suite that is able to open and edit files from Microsoft Office. Many Linux users have it installed in their workstations (with many distributions even bundling it by default), since Microsoft Office does not support Linux.

LibreOffice Calc supports the execution of macros, which are functions that run inside the document and can execute arbitrary commands on the host machine. Macros will inherit the privileges of the user that launched the application, which we were able to exploit to automatically deploy our attack on the user's machine. The use of LibreOffice macros comes with one caveat though, which is that they can only be executed without user interaction if they are signed by a known entity, the file is in a specific location, or the user allowed the execution of unsigned macros. Regardless of these countermeasures, attackers may find ways to bypass or facilitate the execution of these macros. Microsoft Office has a similar set of protections against execution of untrusted macros, and still attackers are able to trick users to execute their malicious files by exploring social engineering techniques or even exploiting vulnerabilities in these countermeasures. For the purpose of our proof-of-concept attack vector, we assumed that the user has LibreOffice configured to allow unsigned macros.

We created a macro inside a LibreOffice spreadsheet that downloads two files: the malicious library from a predefined address, which it stores somewhere in the user's computer (in the `/tmp` folder, for example); and a `.desktop` file which contains the instructions to export the path of the library to the `ld_preload` variable of the user's client application (for example, the Google Chrome browser) when it is launched. The `.desktop` file is related to desktop environments used in Linux computers, which we explain next.

Leveraging the desktop environment to load the malicious library

To install the malicious shared library, we exploit the fact that desktop environments allow the customization of their settings by each user without requiring user confirmation. Desktop environments are responsible for providing a user friendly way to open applications without using the terminal. Normally, these environments have a menu that lists all installed programs, and a lookup functionality to easily find and execute them. The entries in this list are created during the program's installation, which usually requires a privileged user. However, unprivileged users may customize these entries by overriding or creating new ones that will only be seen on their own desktop environment session. This feature can be abused to override a programs's desktop environment entry, which is usually a `.desktop` file with instructions on how to launch it.

Our macro downloads the malicious `.desktop` file for the target application that must use our library, replacing the current legitimate `.desktop` file. Note that this is possible because

the macro inherits the permissions of the current user. The malicious file contains additional instructions to export the path of the already downloaded library to the ld_preload environment variable just before launching the program. These instructions will be executed every time the user opens that program through its desktop environment.

This technique was tested in three of the most common desktop environments on Linux: Gnome, KDE and Cinnamon. In all cases, we were able to download the library and a modified Google Chrome .desktop file, and launch the attacks just by opening the Google Chrome browser through the desktop environment's list of applications.

4.5.6. Testing

We primarily developed and tested our attacks and methods of deployment on Ubuntu 20.04.4 LTS version, using Google Chrome on version 103.0.5060.53 (Official Build 64-bit, installed via SNAP) as the main client, although we also successfully performed tests on Chromium and Brave. We targeted Chrome because it was by far the most popular browser in the world, with approximately 66% market share in desktop computers[11]. Additionally, other Chromium-based browsers like Microsoft Edge, Opera and Brave accounted for an additional 17% of market share, bringing the total market share of all Chromium-based browsers to around 83% (although the real market share on Linux might be lower, given how much more prevalent Firefox is in this system compared to others like Windows or MacOS).

To test the registrations and authentications performed by the attacks, we used the demo relying party implementation `https://webauthn.io`, which provides a simple implementation of the Webauthn specification and also exposes a Web API. On the authenticator side, we have managed to consistently perform our attacks on the following models of commercial authenticators: Solokey USB-A (CTAP 2.0), TrustKey G310 (CTAP 2.0), YubiKey Security Key NFC (CTAP 2.0), and YubiKey Bio (CTAP 2.1).

4.6. Results

We now present a summary of our results on the implementation and deployment of our attacks, concluding with a brief discussion on how they suggest that strengthing FIDO2 against these threats is necessary for improving its security guarantees.

[11] Source: `https://gs.statcounter.com/browser-market-share/desktop/worldwide`. Data was collected in the beginning of 2023. As of this writing, in the beginning of 2026, the market share is approximately 76%, with other Chromium browsers accounting for 12.5%.

4.6.1. Attacks on FIDO2

We have successfully performed all the attacks mentioned in this Chapter[12].

In the impersonation attack, when the user tries to login in the webauthn.io website, we can successfully perform the man-in-the-middle attack every time, capture the pinUvAuthToken, and then use it to establish a parallel authentication session for a different username also registered with the same authenticator in the webauthn.io website. This simulates the case where an attacker tries to usurp the authenticator and maliciously use it to gain access to another one of the user's accounts on that website.

In the rogue key attack, we have consistently been able to register the malicious authenticator instead of the intended user authenticator by hijacking the authenticatorMakeCredential command sent by the client and providing the response from the malicious authenticator. We have successfully used each model mentioned above as the malicious authenticator in the attack, having been able to successfully register it in the webauthn.io website instead of the intended authenticator. We have also successfully implemented our double registration attack, that combines properties from the other two attacks.

Additionally, we were successful in recovering the PIN from the hash recovered during the man-in-the-middle attack using a very simple Python script. The PIN that we recovered was only 4 bytes long, which took approximately 4 seconds to break with an unoptimised brute force algorithm. Nevertheless, this suggests that a resourceful attacker with the right password cracking techniques and capable hardware may be able to obtain the PIN given sufficient time (recall that these attacks on the PIN hash are *offline*).

4.6.2. Deployment

All of our attacks were successfully deployed using the LibreOffice macro technique to download the malicious library and load it into the ld_preload variable associated with the user's browser. Every time the user launches its browser, our malware is able to intercept every FIDO2 message exchanged between the browser and the authenticator.

Even though the malware we developed and the attacks we described target the Linux operating system, these findings should be adaptable to other operating systems. In this work, we leveraged the ld_preload environment variable, but other operating systems have

[12]We have communicated our results in private communications to relevant stakeholders, including the manufacturers of the hardware devices mentioned in Subsection 4.5.6, the Chromium development team, Canonical, DuoLabs and the FIDO Alliance. More details on these communications can be consulted in the original publication [36].

similar features. For instance, macOS has the dyld_insert_libraries environment variable that has the same function as ld_preload. In the case of Windows, we have a registry key, AppInit_DLLs, that has a similar feature set. However, this option is more limited than the others since it requires administrative rights and is disabled on Secure Boot.

Regardless, there are other techniques that may be exploited to achieve the same results. Dynamic code instrumentation enables the manipulation of running processes by injecting code in its address space. This allows the dynamic interception of function calls and the change of variables. Threat actors may leverage these techniques to implement the attacks we demonstrated without relying on the operating system features we listed earlier and its inherited limitations. As we mentioned before, this type of behaviour has been observed in the past [34, 35, 37].

4.6.3. Limitations

We quickly recall some limitations to our attacks.

In the impersonation attack, we usurp the user presence confirmation for the forged request, which means that the user will need to confirm presence a second time to complete its initially intended authentication. We believe that in most cases this would be interpreted by users as a quirk in the legitimate authentication process. Another limitation is that we used the webauthn.io website as the relying party for all of our tests. While we believe that this does not affect the validity of the attack, it is still worth keeping in mind that additional logic might be needed to correctly adapt the attack to complete an authentication session with a different relying party, such as modifying the relying party identifier during the man-in-the-middle attack when working with CTAP 2.1 authenticators to associate the RP ID to the pinUvAuthToken.

One limitation of the rogue key attack is that, for demonstration purposes, we are plugging in the attacker's authenticator in the same machine as the user's rather than accessing it over the internet in a different machine. This makes demonstration cumbersome, but it is without loss of generality. Indeed, we believe that a most realistic deployment of this attack would be for the wrapper library to connect remotely to an attacker-controlled computer where it can use its own authenticator undetected, or to come with a hardwired attestation private key (possibly corrupted or leaked from some authenticator in the adversary's possession), which would allow the library to then launch the attack without any

external interaction. This is also the case if the relying party does not enforce attestation, which is a good argument as to why attestation is important during a registration ceremony.

4.6.4. Discussion

Our attacks show that it is practical to attack FIDO2 through CTAP2, exploiting the insufficient authentication between clients and authenticators. Indeed, the entire procedure of obtaining a pinUvAuthToken to establish an authenticated channel is unauthenticated, paving the way for an attacker to freely manipulate messages exchanged between a client and an authenticator beyond just the ECDH key exchange. This can enable the aforementioned impersonation attack and double registration attacks, as well as attacks on the user PIN. At the same time, once the unauthenticated channel is established, it is not bi-directional, which means that authenticator responses are not bound to the same authenticated channel as client-issued commands. This opens the way for the aforementioned rogue key attack and double registration attack.

However, our implementation and deployment of these attacks is all software-based, where we have to first install a malicious library on a user's computer. As mentioned in Chapter 1, the authors from [18] introduce a way for launching USB message injection attacks on a target computer. The malicious USB devices they created are capable of first observing downstream traffic from the host to a legitimate plugged-in USB device and then, when required, of injecting the last message the host is expecting to receive, which it will detect as coming from the legitimate device. The authors show this malicious device can be programmed to inject key strokes by passing itself as a keyboard, and to alter files being transfered from a legitimate USB device to the host.

Note that this is precisely what is needed to perform our rogue key attack, but without requiring the previous installation of malware on the user's computer. Indeed, an attacker could program such a malicious device to observe traffic between the client and the authenticator. Then, when it detects the client's last CTAP2 message containing an authenticatorMakeCredential command, it can inject the expected CTAP2 response (that is produced by a malicious authenticator in our implementation), which the client will accept as coming from the user's authenticator.

Additionally, in Chapter 5, we describe an attack that enables the attacker to perform offline dictionary attacks on the PIN without having to obtain the hashed PIN directly. This

attack does not require a full man-in-the-middle attack on the ECDH key exchange to work, relying instead on the attacker's ability to hijack the transmission of the encrypted pinUvAuthToken, sent by the authenticator to the client, and replacing it with a mauled ciphertext. The requirements to perform this attack also perfectly match with these USB injection methods, because an attacker can program the malicious device to observe the communications until it detects a getPinToken subcommand, and then inject the malicious USB message containing the mauled ciphertext.

In conclusion, whether executed via software or hardware, our attacks demonstrate that the CTAP2 design choices identified in this Chapter introduce exploitable vulnerabilities, and highlight the necessity of refining the protocol's architecture to effectively mitigate these threats.

In the next Chapter, we discuss FIDO2 in a formal setting, overviewing the protocol's history of formal security analyses, providing our own improved security analysis, and proposing a modification to FIDO2 that rules out its vulnerability to rogue key attacks.

5. FIDO2 security against rogue key attacks

In Chapter 4, we looked into FIDO2's security from a practical angle, explaining how some design choices in CTAP2 can expose the protocol to certain types of attacks. Indeed, the unauthenticated ECDH key exchange and the lack of proper verification of authenticator responses from the client open the way for an *active* adversary to interfere with communications, freely manipulating the exchanged messages. However, while it is true that an adversary with this power can attack the protocol, this doesn't paint the full picture on what security guarantees FIDO2 offers for *passive* attackers. Is the protocol secure if an attacker does not interfere with messages exchanged between the client and the authenticator? What about a partially active adversary, restricted to only be able to manipulate specific messages? What can be done to improve FIDO2 so that it can resist against more powerful adversaries?

In this Chapter, we switch focus to the theoretical side of FIDO2, where we formally define and analyse its authentication security guarantees. We place a greater focus on CTAP2, which is the protocol we analysed more throughly, and for which we propose modifications that make FIDO2 resistant to rogue key attacks. These were motivated by our results from Chapter 4, where we demonstrated that these are realistic attack scenarios on the protocol.

We start in Section 5.1, where we first present an overview of the previous work conducted in this field, explaining the existing security analyses of FIDO2 by underlining their main contributions and shortcomings. Then, in Section 5.2, we explain our novel results. In Section 5.3, we analyse CTAP2, fixing some oversights from prior work, and then propose an addition to the protocol that provably strengthens its security. In Section 5.4, we introduce the syntax for PIA protocols and the security model for analysing WebAuthn, based prior work. Finally, in Section 5.5 we introduce our model for analysing the composition of WebAuthn with our proposed modified CTAP2 protocol, and show how it strengthens FIDO2 to resist rogue key attacks when compared to the original FIDO2.

5.1. Background

The design rationale and threat model for FIDO2 are described informally by the FIDO Alliance in [38]. A formal view of FIDO2's authentication properties has been established by a sequence of research papers, with some focusing exclusively on the WebAuthn [7,

39] security and privacy, and others focusing on analysing the overall FIDO2 as composed by WebAuthn and CTAP2 [1, 2]. While these works offered comprehensive studies of FIDO2's provable security, there are still some gaps in the results they provided.

Barbosa et al. [1] defined the syntax and authentication security of FIDO2 by introducing two primitives called *Passwordless Authentication (PIA)* and *PIN-Based Access Control for Authenticators (PACA)*. WebAuthn is cast as an instance of PIA, and CTAP2 as an instance of PACA. This terminology is also adopted by Bindel et al. [2], where the authors refined the syntax and security models to more closely reflect the structure (and various versions) of the FIDO2 specification. Later, Bindel et al. [39] continued the work of [2], focusing only on WebAuthn, but considered various attestation types and extended the analysis to privacy guarantees.

More detailed background is summarized next.

5.1.1. First FIDO2 security analysis

The first full security analysis of FIDO2 was performed by Barbosa et al. in [1], where the authors proved the *passwordless authentication* security of WebAuthn (version 1) as an instance of PIA and the *unforgeability with trusted binding* (UF-t) security of CTAP 2.0 as an instance of PACA.

PIA security

Informally, the *passwordless authentication* security of a PIA protocol as defined by the authors captures the guarantee that, if a server (relying party) accepts an authentication response from an authenticator, then it must have performed a previous registration run with the same authenticator. Furthermore, if a server accepts a registration response with a given attestation public key, then the response must have been generated by an authenticator that holds the corresponding attestation private key.

The PIA trust model considered here by the authors allows active adversaries against the *passwordless authentication* experiment both in registration and authentication, but with the assumption that each authenticator has a *unique* attestation private key that is known *a priori* by the relying party. In these circumstances, the authors were able to prove the *passwordless authentication* security of WebAuthn, but also point out that if Basic attestation is used instead, PIA security only guarantees that if a server accepts a registration response with a given attestation public key, then the response must have been produced

by *one* of the authenticators sharing the corresponding private key, although it may not be the one the user is attempting to register.

PACA security

In the PACA security model, the authors consider four security experiments: *unforgeability* (UF), *strong unforgeability* (SUF), *unforgeability with trusted binding* (UF-t), and *strong unforgeability with trusted binding* (SUF-t). The adversary is considered to win if it can forge a command that is accepted by the authenticator (i.e., produce a valid authenticated command without trivially obtaining the pinToken from a client). The differences between experiments are related to the restrictions on what the adversary can do when interacting with the protocol. Independently of what security experiment is considered, the trust model considered by the authors assumes *trust-on-first-use* (TOFU) during the setup of an authenticator, i.e., up until the PIN is set by the user. This means no adversary can actively interfere with this part of the protocol.

The UF and SUF experiments relate to the relationship between authenticators and clients, which the authors call *partnership*. We formally introduce the notion of partnership in Section 5.3, but intuitively a client and an authenticator are considered partners if they have established a secure channel, i.e., if the client has obtained a valid pinToken (or pinUvAuthToken) from the authenticator. In CTAP 2.0, which was the only version of CTAP2 at the time, the pinToken is not regenerated every time a client requested it via the getPinToken subcommand. This means multiple clients can end up sharing the same pinToken, and therefore an authenticator can have multiple partners at the same time (i.e., there can be multiple active authenticated channels with the same authenticator). From CTAP 2.1 onwards, the pinUvAuthToken is regenerated every time a client requests one (even across different PIN/UV Auth Protocol instances), and thus an authenticator can have only one partner at a time. In the UF security experiment, the adversary cannot win by delivering an authenticated command to an authenticator if it obtained the pinToken from any client that is partnered with that authenticator before issuing the command. In the SUF experiment, the adversary cannot win if it obtains the pinUvAuthToken from the authenticator's unique partner, but can still win if it obtains a pinUvAuthToken from any other client in the experiment (this captures the fact that only *one* pinUvAuthToken is active at any given time).

The difference between UF and UF-t (and SUF and SUF-t) relates to the adversary's ability to perform active attacks during the process of obtaining the pinToken, which the authors

call Bind. When the security experiment has *trusted binding* (UF-t or SUF-t), the adversary cannot actively interact with a client during the entire exchange performed when obtaining a pinToken, but can still attempt to perform active attacks against the authenticator (such as attempting to guess the PIN). When there is no *trusted binding*, then the adversary is free to actively engage with the client as well.

The authors proved that CTAP 2.0, as a PACA protocol, can only achieve UF-t security, which is consistent with the man-in-the-middle attack described in Chapter 4. However, they proposed some changes to CTAP2, and were able to prove that, with those changes, the protocol achieved SUF security. We expand on these results in Chapter 6.

Composed security

The authentication security of FIDO2 as a whole is captured considering FIDO2 as composed model of PIA+PACA. The trust model is that of both PIA and PACA, with the additional assumption of a trusted server-client communication channel, which captures the TLS communication between a relying party and the client. The authors prove the *mutual authentication* security of FIDO2 as a composition of WebAuthn + CTAP2, which they divide into two separate security definitions: *user authentication* and *server authentication*. Informally, the former captures the server's guarantee that, for an attacker to impersonate a user to the server, then it must compromise an access channel to the authenticator and have it accept an authorized command from the server through that channel. The latter captures the user's guarantee that, if the intended server accepts, then it is partnered with the correct user token.

The authors then prove that the only way to break these guarantees is to either break PIA or PACA.

5.1.2. An updated analysis of FIDO2

Based on the work from [1], the authors in [2] extended and refined the security models of PIA and PACA, which they called ePIA and ePACA, to capture the new versions of WebAuthn and CTAP2, detailing them by describing the security experiment oracles, winning conditions and the syntax for both WebAuthn (version 2) and CTAP 2.1 in pseudocode. In this work, we closely followed their definitions, adapting them to include our trust models and protocol changes.

ePIA security

On the ePIA side (*Extended Passwordless Authentication*), the authors provided a new analysis on the *passwordless authentication* security of WebAuthn (version 2) with None attestation, which deviates from the assumption of a unique attestation private key from [1], at the cost of changing the trust model to assume only passive adversaries during registration. Furthermore, the authors provided a more fine-grained model of WebAuthn, which additionally considers aspects like algorithm negotiations, support for the token binding protocol [40] (which links a WebAuthn session to the TLS connection[1]) and user actions like user presence and user verification requirements from the relying party.

Additionally, they slightly modify the definition of partnership between servers and authenticators, which we discuss in Section 5.4.

ePACA security

On the ePACA side (*Extended PIN-Based Access Control for Authenticators*), the authors changed the *trusted binding* assumption from [1], introducing a new security definition called SUF-t'. In this trust model, an adversary can launch active attacks against clients, but only when delivering the final message a token sends to a client when responding to a getPinToken (or getPinUvAuthTokenUsingPinWithPermissions) subcommand, which contains the encrypted pinUvAuthToken. Note that, while this does not allow the adversary to perform the man-in-the-middle attack against the ECDH key exchange, because it is still not allowed to provide its own ECDH public key to a client, it nevertheless grants the adversary additional power to interact with the protocol. The authors provided a proof that CTAP 2.1 satifies SUF-t' security, on which we expand in Section 5.3.

Composed security

The composed analysis of ePIA+ePACA is analogous to the analysis from [1], where breaking the guarantees of FIDO2 requires breaking either ePIA or ePACA.

5.1.3. An extended analysis of WebAuthn

The authors in [39] followed on the work from [2] on the provable security of WebAuthn, expanding on the syntax and security model to include multiple attestation types and a new analysis on the privacy guarantees of WebAuthn.

[1] The usage of token binding is no longer expected in WebAuthn version 3, as mentioned in the specification [28].

The authors proved the *passwordless authentication* security of WebAuthn as an instance of an ePIAA (*Extended Passwordless Authentication Protocol with Attestation*) protocol for every attestation type. Unlike in [2], they do not assume passive adversaries during registration in the *passwordless authentication* experiment, and point out that with this trust model WebAuthn cannot satisfy this security notion with attestation types None or Self. Therefore, they defined a *weak* version of *passwordless authentication*, which does assume passive adversaries as in [2], and proved WebAuthn satifies this security notion with the aforementioned attestation types.

In this work, the authors reuse the same definition of partnership from [2], which differs from [1] and the work we present in this thesis. We expand on this security model, and how it differs from our model, in Section 5.4.

5.1.4. Provable security and rogue key attacks

Performing a rogue key attack on FIDO2 is possible during registration because neither the relying party nor the client can cryptographically verify the origin of an authenticator response. In most cases, the relying party has no *a priori* information about the authenticator, and the authenticated channel established between a client and an authenticator is not bi-directional.

The aforementioned papers on the provable security of FIDO2 do not capture rogue key attacks. In [1], the authors rule out this type of attack by assuming unique attestation keys for each authenticator, which are known *a priori* to the relying party. However, they do recognize that if Basic attestation is considered, then it is not possible to guarantee that the authenticator that registered with the relying party is the authenticator the user intended. At most, a server *could* be capable of requiring authenticators from the same batch, and expect the usage of a specific attestation key *a priori*. However, even in this scenario, an attacker would still be able to launch a rogue key attack, provided it used an authenticator from the same batch as the user's.

The trust model in [2] considers the registration phase to be authenticated, both in PIA and the composed model, which also does not capture rogue key attacks[2]. However, this may not be realistic in practice, as our proof of concept implementation of the rogue key attack demonstrated in Chapter 4.

[2] Although, as the authors point out, this captures the trust model in the official WebAuthn specification [9], in Section 13.4.4

Finally, the work in [39] considered only the WebAuthn protocol, which means that the security of client-token communications is not considered.

5.2. Our results

The work we describe in this Chapter closes the gaps identified in prior work. In [1], the results on the CTAP2 authentication security apply only to CTAP 2.0, which only achieves UF-t security. In [2], the authors analyse CTAP 2.1, introducing an updated *trusted binding* definition, which strengthens the model by allowing more powerful adversaries against ePACA protocols. The authors also point out some mistakes in the proofs from [1] but, as we will discuss later, also have some oversights in their proof of CTAP 2.1. Furthermore, neither analysis captures rogue key attacks, due to the trust model not allowing active adversaries in registration or due to assumptions on the relying party's ability to discern between different authenticators using their unique attestation keys.

Regarding CTAP2 specifically, neither work models the authenticator's response to a command and its reception by the client, which is essential in capturing rogue key attacks at the CTAP2 level. This is to be expected, since an authenticator's response in CTAP2 is not validated at a cryptographic level by the client. Nevertheless, in order to strengthen CTAP2 against rogue key attacks, it is essential to extend the security analysis to capture the handling of authenticator responses.

Our results build on prior work in two fundamental ways. First, we provide a new detailed proof for CTAP 2.1, fixing the oversights found in [2], and clarify how FIDO2 authentication can be proven secure with its current specification. Second, we extend the PACA class of protocols to account for rogue key attacks, and prove that our modified CTAP2 protocol, cast as an instance of this new extended PACA protocol, is resistant against this type of attack. Concretely, we present the following new results:

- We fix the security definitions under which the current version of FIDO2 can be proved to provide passwordless authentication. Our model is similar to that of [1, 2] and guarantees the following: (i) a server *registers* only credentials generated by valid *physical* authenticators (when considering Basic attestation), (ii) if the server successfully *authenticates* a registered credential, then the authenticator that generated that credential must be involved in the authentication, and (iii) if an authenticator is interacting with an honest client, then it will only answer registration/authentication requests if the

client unlocks this authenticator using the correct user PIN[3]. We give a detailed proof of FIDO2 authentication security in this model that addresses the shortcomings we identified in prior work. Our proof applies to the attestation types most commonly used by USB authenticators: None, Self and Basic.

- We propose a stronger variant of the composed authentication model that captures the rogue key attacks described in Chapter 4 and the USB injection attacks demonstrated in [18]. Intuitively, this stronger model further guarantees that (iv) if a server successfully *registers* or *authenticates* a credential via an honest client, then the user must have *authorized* that specific client to unlock the authenticator that generated that credential. Crucially, this model does *not* assume TOFU: the client (browser) plays a role in ensuring the server (and to the user) that, if the server is talking to a client through an authenticated channel (e.g., a TLS connection), then only an authenticator unlocked by this specific client can successfully register a credential.

- We propose a simple fix to CTAP2 that appends a message authentication code (MAC) to the authenticator responses before they are sent to the client, where they are validated before being delivered to the server. We then provide a detailed proof that, with this fix, FIDO2 is provably resistant to rogue key attacks and meets our stronger security. Our results apply to attestation types None, Self and Basic, with a trust model that no longer requires passive adversaries during registration, a departure from prior work.

- We present a counterexample to the security claim for CTAP2 in [2] in the form of an attack that breaks the protocol in their security model for authentication. The attack, which allows an adversary to perform offline dictionary attacks on the PIN by replacing the encrypted pinUvAuthToken sent to the client with a mauled ciphertext, highlights a subtle oversight in their security proof which effectively shows that the new *trusted binding* setting makes the security model too strong, and that FIDO2 can only be proven secure in a slightly weaker model.

In the next Section, we first present our analysis of CTAP 2.1, and then introduce our proposed fix that makes this protocol resistant to rogue key attacks.

[3]Note that, as in prior work and due to the lack of authenticator-to-client authentication, FIDO2 guarantees only that the *same* authenticator is used in registration and authentication. However, it does not ensure that the authenticator used in authentication is bound to the (user-controlled) client through which the authentication run is carried out, even when assuming *trust on first use*, i.e., the attacker was passive during registration.

5.3. CTAP 2.1 and CTAP 2.1+ security analysis

The authors from [1, 2] prove the authentication security of CTAP2 by modeling it as an instance of a PACA protocol (we simplify the naming of ePACA to PACA whenever we refer to both protocols). The security models defined in the two publications are built on the basic PACA syntax, which defines certain subprotocols and state variables that describe high level processes, and do not take into account specific implementations of these subprotocols. When proving the security of CTAP2 as an instance of PACA, the authors then define the concrete protocol syntax, including its functions and variables, and how they combine to implement the PACA subprotocols.

We also follow this approach, by first introducing the syntax of a PACA protocol and of our extension called mPACA, followed by the respective security models, and then proving the SUF-t security of CTAP 2.1 as a PACA protocol, and of our fixed protocol, which we call CTAP 2.1+, as an mPACA protocol. As previously mentioned, our pseudocode security model and protocol syntax closely follows that of [2].

CTAP2 scenarios captured in the model The detailed CTAP2 description in Appendix B describes all commands for obtaining a pinUvAuthToken, as well as in what scenarios user presence and user verification is required. Our model does not account for all possible configurations of settings (as mentioned in Chapter 3, we consider only specific usage scenarios) in which CTAP 2.1 may be executed, following instead the approach from [1, 2] in assuming *specific* common usage scenarios of CTAP 2.1 when considering USB authenticators. Concretely, we model cases where: (i) the pinUvAuthToken is obtained using a PIN[4], (ii) a pinUvAuthToken always has the correct permission for the command being issued (since we don't consider the permissions of a pinUvAuthToken), (iii) user presence is always required for an authenticator to accept a command, and (iv) user verification is always required, modeled by the fact that a pinUvAuthToken is necessary for validating commands and that it can only be obtained using the PIN. Furthermore, we don't consider every command CTAP 2.1 supports. A client-issued command, in our model, is an abstract message that is authenticated by the client and validated by the authenticator. When considering in composition with WebAuthn, this approach aims to model *only* the basic usage of authenticatorMakeCredential and authenticatorGetAssertion commands. For

[4] Although we don't model cases where user verification is provided by biometric authentication, as was pointed out in [2], the model can be adapted to have the client not send the encrypted hashed PIN c_{ph} to the authenticator, modeling the biometric authentication in the authenticator as an authenticated transmission. We leave this as future work.

example, we don't consider scenarios where authenticatorGetAssertion may be used multiple times for "pre-flight" operations, which do not require proof of user presence (see Appendix B). We leave the analysis of other CTAP2 features and commands for future work.

A remark on CTAP 2.2 Our work was conducted when the latest released version of CTAP2 was CTAP 2.1. However, as previously mentioned, CTAP 2.2 does not introduce any new features or changes that influence our model and results. Notably, it also does not solve any of the attack scenarios described in Chapter 4. As a result, although we refer to CTAP 2.1 in this work, the same results apply to CTAP 2.2.

5.3.1. Syntax for (m)PACA protocols

PACA syntax

A PACA protocol is an interactive protocol between 3 parties: a client C, a token T and a user U. It comprises five subprotocols: Reboot, Setup, Bind, Auth-C and Validate-T. Below, when we say a protocol takes as input C and/or T, we mean their states, which are updated during the protocol execution.

Reboot: inputs a token T and initializes (or refreshes) its volatile storage, to prepare for the execution of the other PACA subprotocols. This subprotocol powers up the token, and always requires *user interaction* (e.g., a USB token being plugged out and then plugged in to a machine). The token's static storage is not affected.

Setup: inputs a token T, client C, and user U who participates by providing a PIN to C. During this subprotocol execution, C securely transmits the PIN to T, and then T saves information about this PIN in its static storage, such that later it can be used by U to authorize clients to access T. This subprotocol is executed *once* per token.

Bind: inputs a token T, client C, and user U who again provides C with a PIN. This subprotocol aims to create an authenticated channel from C to T. It can be expressed as multiple runs of client-side and token-side processing Bind-C and Bind-T. Bind-C inputs C, U and a message m and outputs a message m'. Bind-T inputs T and a message m and outputs a message m'. Depending on the stage of execution of Bind-C or Bind-T, m and m' will represent different types of information. If the protocol terminates successfully, the client and the token share a volatile binding state bs, establishing the authenticated channel.

Auth-C: inputs a client C, command M, and outputs (M, t), where t is a tag that authorizes command M. This subprotocol authorizes commands that C sends to token T, using $C's$ binding state obtained from Bind.

Validate-T: inputs a token T, command M, tag t and user decision bit d. This subprotocol validates command M given tag t, $T's$ binding state and a user decision bit d, and outputs a bit b as the result of the validation.

Correctness imposes that if a token T is Setup with a PIN, and derives a binding state bs with a client C through Bind (using the PIN), then a command authorized by C through Auth-C using bs is accepted by T in Validate-T (i.e., $b = 1$) if and only if $d = 1$ (i.e., the user approves the command).

mPACA syntax

To capture our authentication fix of CTAP2, we now introduce the syntax of an mPACA protocol, where the "m" stands for "mutually authenticated". An mPACA protocol extends the syntax of PACA with two additional subprotocols:

Auth-T: inputs a token T, response M, and outputs (M, t), where t is a tag that authorizes response M. This subprotocol authorizes responses that T sends to client C, using $T's$ binding state obtained from Bind.

Validate-C: inputs a client C, response M and tag t. This subprotocol validates response M given tag t and $C's$ binding state, and outputs a bit b' as the result of the validation.

Correctness is extended from PACA to further impose that if a token T authorizes a response through Auth-T, then client C will accept the response through Validate-C (i.e., $b' = 1$).

5.3.2. (m)PACA Authentication Model

We closely followed [2] to define our authentication security model for PACA protocols, and extended it to capture mPACA security. As presented in Figures 5.1, 5.2, the authentication security (SUF-t) of a PACA protocol PACA is defined with a security experiment $\mathsf{Expt}^{\mathsf{SUF\text{-}t}}_{\mathsf{PACA}}$, executed between a challenger and an adversary $\mathcal{A}$.

Trust model The model assumes a fully authenticated channel for all communications between clients and tokens during Setup. It also assumes that no active attacks can be

carried out against any client during the whole execution of Bind, while fully active attacks on tokens are allowed (i.e., an adversary can start Bind with a token). As previously stated, this is the *trusted binding* assumption made in [1], which differs from the model in [2] that allows for stronger adversaries that may carry out active attacks against clients at the end of Bind. We later expand on the impossibility of achieving this stronger security in Section 5.3.4.

Session oracles and states We consider two types of session oracles π_T^i and π_C^j to specify the i^{th} and j^{th} instance of token T and client C, respectively. An (m)PACA protocol implements certain states for client and tokens. Client session oracles are completely independent from each other and maintain no global state for any given C. Session oracles of token T each share a global state st_T, which contains the associated user identifier $\mathsf{st}_T.\mathsf{user}$, some information about the PIN, and some initialization data $\mathsf{st}_T.\mathsf{initialData}$ which includes static configuration data like the supported protocol versions, and other protocol-specific states like a public counter that limits the maximum number of failed PIN tries. π_T^i and π_C^j have a binding state bs, session identifier sid and execution state $\mathsf{st}_{\mathsf{exe}} \in \{\perp, \mathsf{waiting}, \mathsf{bindStart}, \mathsf{bindDone}\}$. If a session (oracle) is not yet initialized, we simply write $\pi_T^i = \perp$ or $\pi_C^j = \perp$. When describing CTAP2 as an (m)PACA instance, following prior work, we use session oracles to simplify presentation and model the fact that an incoming message is processed in the context of a specific session. A client or token session simply represents one of possibly multiple executions of (m)PACA for a given client or token. Some information is shared between executions, like the token's stored information about the PIN, while other only pertains to one specific execution, like the binding state.

Experiment-specific boolean variables Each session (oracle) has an isValid variable, stating if this session is available for interaction with the adversary. It defaults to true when the session oracle is created. Additionally, each token session has a $\pi_T^i.\mathsf{pinCorr}$ variable determining if the PIN associated with token T has been corrupted, and each client session has a $\pi_C^j.\mathsf{compromised}$ variable determining if its internal state has been compromised.

Experiment oracles We showcase the $\mathsf{Expt}_{\mathsf{PACA}}^{\mathsf{SUF\text{-}t}}$ oracles in Figure 5.2, which closely follow the pseudocode description in [2]. NewT and NewU generate new tokens and new users, respectively. CorruptUser permanently corrupts a user U (by flagging it as corrupt) and returns its PIN to $\mathcal{A}$. The Setup oracle performs a full Setup subprotocol run between

two sessions π_C^j and π_T^i with user U's PIN, and returns the trace of communications to $\mathcal{A}$[5]. Likewise, Execute performs a full Bind subprotocol run between two sessions π_C^j and π_T^i, using the PIN that was stored in T, invalidating all of T's previous sessions; the full trace of communications is then given to $\mathcal{A}$. Compromise returns a client session's binding state π_C^j.bs and permanently marks this session as compromised. Reboot on a token session π_T^i calls the Reboot subprotocol on T and marks all of T's sessions as invalid. Auth-C authorizes a message M on a client session π_C^j using its binding state and outputs the same message and a tag t. Validate-T inputs a message M, tag t and user decision bit d on a token session π_T^i and outputs a boolean response. Finally, Send-Bind-T allows $\mathcal{A}$ to send a message m to a token session π_T^i to initiate, continue or complete Bind; in the latter case, it also invalidates all of T's previous sessions, including π_T^i if the query caused T to reboot.

Users with multiple tokens In this model, a user U does not represent a *human* user with multiple tokens and PINs. Instead, U is simply an abstraction that represents a single PIN, which can be used to run Setup with multiple tokens. This captures the somewhat unusual scenario in which a *human* user has multiple tokens with the same PIN. The most natural scenario where a *human* user has multiple tokens with different independent PINs is captured in our model by having different users setting up different tokens, with each user U setting up only one token.

Session partnership We say sessions π_C^j and π_T^i are *partners* if, and only if, they completed Bind and agree on the session identifier sid. The concrete instantiation of sid depends on the concrete protocol to be analyzed. For our analysis of CTAP2, we take the sid to be the full trace of Bind.

Winning conditions For a PACA protocol PACA, we say that an adversary $\mathcal{A}$ against the security experiment $\text{Expt}_{\text{PACA}}^{\text{SUF-t}}$ wins if it gets a message-tag pair (M, t) accepted by a token session π_T^i through the Validate-T oracle and one of the following statements holds: (i) the user decision bit $d \neq 1$, (ii) two token sessions complete Bind with the same sid, (iii) two client sessions complete Bind with the same sid, or (iv) T's PIN was not corrupted and *either* π_T^i has no partner *or* its partner was not compromised and did not output (M, t). This is captured by the winning condition Token-Win-SUF-t in Figure 5.1, which is checked

[5] Like in [2], we did not model the token reset or PIN change operations. Indeed, a user can set up as many tokens as it wishes with the same PIN, and tokens with different PINs are modeled by different users setting up different tokens.

$\text{Expt}^{\text{SUF-t}}_{\text{PACA}}(\mathcal{A})$:
1: $\mathcal{L}_{\text{authC}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: win-SUF-t $\leftarrow 0$
4: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
5: **return** win-SUF-t

tokenBindPartner (T, i):
1: **if** $\exists (C, j)$ s.t. $\pi^i_T.\text{sid} = \pi^j_C.\text{sid}$ **then**
2: **return** (C, j)
3: **return** $(\perp, \perp)$

Token-Win-SUF-t (T, i, M, t, d):
1: **if** $d \neq$ accepted **then return** 1
2:
3: **if** $\exists (C_1, j_1), (C_2, j_2)$ $s.t.$ $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{j_2}_{C_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{j_1}_{C_1}.\text{sid} = \pi^{j_2}_{C_2}.\text{sid}$ **then return** 1
4:
5: **if** $\exists (T_1, i_1), (T_2, i_2)$ $s.t.$ $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{i_2}_{T_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{i_1}_{T_1}.\text{sid} = \pi^{i_2}_{T_2}.\text{sid}$ **then return** 1
6:
7: $(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
8: **if** $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
9: **if** $(C, j) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
10: **if** $\pi^i_T.\text{pinCorr} = \text{false}$ **then**
11: **return** 1
12: **return** 0

$\text{Expt}^{\text{SUF-t}}_{\text{mPACA}}(\mathcal{A})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: win-SUF-t $\leftarrow 0$
4: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
5: **return** win-SUF-t

clientBindPartner (C, j):
1: **if** $\exists (T, i)$ s.t. $\pi^j_C.\text{sid} = \pi^i_T.\text{sid}$ **then**
2: **return** (T, i)
3: **return** $(\perp, \perp)$

Client-Win-SUF-t (C, j, M, t):
1: **if** $\exists (C_1, j_1), (C_2, j_2)$ $s.t.$ $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{j_2}_{C_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{j_1}_{C_1}.\text{sid} = \pi^{j_2}_{C_2}.\text{sid}$ **then return** 1
2:
3: **if** $\exists (T_1, i_1), (T_2, i_2)$ $s.t.$ $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{i_2}_{T_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{i_1}_{T_1}.\text{sid} = \pi^{i_2}_{T_2}.\text{sid}$ **then return** 1
4:
5: $(T, i) \leftarrow \text{clientBindPartner}(C, j)$
6: **if** $(T, i, M, t) \notin \mathcal{L}_{\text{authT}}$ **then**
7: **if** $\pi^j_C.\text{compromised} = \text{false}$ **then**
8: **if** $\pi^i_T.\text{pinCorr} = \text{false}$ **then**
9: **return** 1
10: **return** 0

Figure 5.1: (m)PACA authentication security experiments and winning conditions. Code in red is defined only for the mPACA model. $\mathcal{O}$ denotes all oracles available to $\mathcal{A}$, as shown in Figure 5.2. The winning conditions are checked in Validate-T and Validate-C whenever a message-tag pair (M, t) is accepted.

whenever the adversary queries Validate-T and the validation of (M, t) is successful (i.e., when a token session π^i_T accepts a command).

Security for mPACA protocols The mPACA authentication model extends the PACA authentication model with two additional oracles Auth-T and Validate-C, as shown in Figure 5.2, that provide the adversary with the power to obtain token-authorized responses and check the validity of response-tag pairs (M, t). The winning conditions for the mPACA security experiment $\text{Expt}^{\text{SUF-t}}_{\text{mPACA}}$ (shown in Figure 5.1) are also extended to reflect the additional attack vectors: an additional winning condition Client-Win-SUF-t is checked whenever the adversary queries Validate-C with a valid pair (M, t).

Advantage measures For protocol $\text{prot} \in \{\text{PACA}, \text{mPACA}\}$, the authentication advantage of $\mathcal{A}$ is defined as:

$$\text{Adv}^{\text{SUF-t}}_{\text{prot}}(\mathcal{A}) = \Pr[\text{Expt}^{\text{SUF-t}}_{\text{prot}}(\mathcal{A}) = 1] \,.$$

NewT (T, initialData):
1: **if** $\text{st}_T \neq \perp$ **then**
2: **return** $\perp$
3: st_T.initialData $\leftarrow$ initialData
4: Reboot(st_T)
5: **return**

CorruptUser (U):
1: **if** $\mathcal{L}_{\text{valid}}[U] = \perp$ **then**
2: **return** $\perp$
3: $\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ true
4: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
5: **return** pin

Reboot (T):
1: **if** $\text{st}_T = \perp$ **then**
2: **return** $\perp$
3: **for all** i *s.t.* $\pi_T^i \neq \perp$ **do**
4: π_T^i.isValid $\leftarrow$ false
5: Reboot(st_T)
6: **return**

Send-Bind-T (T, i, m):
1: **if** $\text{st}_T = \perp$ **then return** $\perp$
2: **if** $\pi_T^i = \perp$ **then**
3: $\pi_T^i \leftarrow \text{st}_T$
4: **if** $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone **or** π_T^i.isValid $=$ false **then**
5: **return** $\perp$
6: π_T^i.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
7: **if** $\pi_T^i.\text{st}_{\text{exe}} =$ waiting **then**
8: $m_T \leftarrow$ Bind-T(π_T^i, m)
9: $c_{pl} \,||\, \text{calledReboot} \leftarrow m_T$
10: **if** calledReboot $=$ true **then**
11: **for all** i' *s.t.* $\pi_T^{i'} \neq \perp$ **do**
12: $\pi_T^{i'}$.isValid $\leftarrow$ false
13: **else if** $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone **then**
14: **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
15: $\pi_T^{i'}$.isValid $\leftarrow$ false
16: **else**
17: $m_T \leftarrow$ Bind-T(π_T^i, m)
18: **return** m_T

Auth-C (C, j, M):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **or** π_C^j.isValid $=$ false **then**
2: **return** $\perp$
3: $(M, t) \xleftarrow{\$}$ auth-C(π_C^j, M)
4: $\mathcal{L}_{\text{authC}} \xleftarrow{\cup} \{(C, j, M, t)\}$
5: **return** (M, t)

Auth-T (T, i, M):
1: **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq$ bindDone **or** π_T^i.isValid $=$ false **then**
2: **return** $\perp$
3: $(M, t) \xleftarrow{\$}$ auth-T(π_T^i, M)
4: $\mathcal{L}_{\text{authT}} \xleftarrow{\cup} \{(T, i, M, t)\}$
5: **return** (M, t)

NewU (U):
1: **if** $\mathcal{L}_{\text{valid}}[U] = \perp$ **then**
2: pin $\xleftarrow{\$} \mathcal{P}$
3: $\mathcal{L}_{\text{valid}}[U] \leftarrow$ pin
4: $\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ false
5: **return**

Compromise (C, j):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **then**
2: **return** $\perp$
3: π_C^j.compromised $=$ True
4: **return** π_C^j.bs

Setup (T, i, C, j, U):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
2: **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then**
3: **return** $\perp$
4: $\pi_T^i \leftarrow \text{st}_T$
5: trans $\xleftarrow{\$}$ Setup(π_T^i, π_C^j, pin)
6: π_T^i.isValid, π_C^j.isValid $\leftarrow$ false
7: st_T.user $\leftarrow U$
8: **return** trans

Execute (T, i, C, j):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
2: **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then**
3: **return** $\perp$
4: $\pi_T^i \leftarrow \text{st}_T$
5: trans, $m_C, m_T \leftarrow \perp$
6: **while** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **do**
7: $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
8: $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
9: trans $\leftarrow$ trans $|| \, m_T \, || \, m_C$
10: **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
11: $\pi_T^{i'}$.isValid $\leftarrow$ false
12: **return** trans

Validate-T (T, i, M, t, d):
1: **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq$ bindDone **or** π_T^i.isValid $=$ false **then**
2: **return** $\perp$
3: status $\leftarrow$ validate-T(π_T^i, M, t, d)
4: **if** status $=$ accepted **then**
5: win-SUF-t $\leftarrow$ Token-Win-SUF-t(T, i, M, t, d)
6: **return** status

Validate-C (C, j, M, t):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **or** π_C^j.isValid $=$ false **then**
2: **return** $\perp$
3: status $\leftarrow$ validate-C(π_C^j, M, t)
4: **if** status $=$ accepted **then**
5: win-SUF-t $\leftarrow$ Client-Win-SUF-t(C, j, M, t)
6: **return** status

Figure 5.2: Oracles for the (m)PACA security experiments. Changes from [2] are in blue. Code in red is defined only for the mPACA model.

Differences from [2] Our security model modifies the Execute and Send-Bind-T oracles from [2] back to the modeling approach of [1], in order to exclude active attacks on clients. In [2], the Execute oracle only runs half of Bind passively, and the adversary must use Send-Bind-T and Send-Bind-C to complete the Bind process. Our Execute oracle captures a full Bind execution between two sessions π_C^j and π_T^i, whereas Send-Bind-T allows the

adversary to arbitrarily attempt to bind with a token session π_T^i. We remove the Send-Bind-C entirely, which removes the adversary's ability to behave actively when *completing* Bind on the client side.

Another minor difference is that we modify the Execute oracle's input, which no longer includes the user U. Therefore, Execute always uses the configured PIN information stored in the involved token. Not doing so, as in [2], implies that the adversary could always convince another user to interact with a token, and trivially win the game if the PINs coincide. This adds to the advantage of the adversary a non-negligible term that depends on the number of passive protocol executions, which is non-standard in the modeling of low-entropy secrets such as passwords and PINs.

5.3.3. CTAP 2.1 as a PACA protocol

We now introduce the additional experiment variables specific to CTAP 2.1, its high level flow of execution (as a PACA protocol), and its complete syntax.

Session and protocol variables

As in [2], we specify here the relevant variables for tokens and clients specific for CTAP 2.1. All variables defined for PACA are naturally inherited in CTAP 2.1. A token's internal state st_T is composed of: (i) a token version st_T.version , (ii) a list of available PIN/UV protocols st_T.puvProtocolList, (iii) the currently selected PIN/UV protocol st_T.puvProtocol, (iv) the counter for the maximum amount of PIN failed attempts st_T.pinRetries, (v) the counter for the number of consecutive PIN failed attempts $\mathsf{st}_T.m$ and (vi) the stored hashed PIN st_T.pinHash. Both tokens and clients (i.e. all session oracles π_C^j and π_T^i) share: (i) the binding state bs which is set as the pinUvAuthToken and (ii) the session identifier sid, which is defined as the full trace of Bind. Client sessions have: (i) the selected PIN/UV protocol π_C^j.puvProtocol, and (ii) the ephemeral session key derived from ECDH π_C^j.K.

Additionally, we added a new π_T^i.canValidate variable to token sessions, which was not present in prior work. It is set to true in the protocol function obtainPinUvAuthToken-T, after the generation of the pinUvAuthToken. The role of the flag is to determine if validate-T can verify the received tag or not, for a given token session π_T^i. After the first successful validation, it is permanently set to false. This modification to the protocol description aims at more closely following the CTAP 2.1 protocol specifications. Recall that any command sent to a token which requires user presence (modeled by the bit d) and verifies correctly causes the token to revoke permissions from all currently in use binding states, effectively

removing the possibility of validating more commands using the same binding state after the first successful validation. In our model, we always set this variable to false after the first successful validation because this validation requires d to be 1, i.e., user presence is always performed.

High-level flow and syntax of CTAP 2.1

The instantiation of CTAP 2.1 as a PACA protocol is simply the implementation of the PACA subprotocols with the concrete CTAP 2.1 operations. We show how these subprotocols are instanced in Figure 5.3, and provide the pseudocode functions shown in the diagram in Figure 5.4. Both are based on the figures from [2].

The Reboot subprotocol is performed via the authPowerUp-T function, which inputs the state of the token st_T and freshly samples a new ECDH key pair and pinUvAuthToken for each supported protocol. It also resets the consecutive pin attempts counter $\mathsf{st}_T.m$. The variable st_T.initialData contains the token version and supported PIN/UV Auth Protocol list, and is used only the first time authPowerUp-T is called (for a token T) to set st_T.version and st_T.puvProtocolList.

During the Setup phase, the authenticator T first outputs its info, which contains the list of supported PIN/UV Auth Protocol (max. 2), and sends it to the client C. C then runs obtainSharedSecret-C-start, which selects the protocol it is going to use and sends that information to T. This step models the authenticatorClientPIN command with the getKeyAgreement subcommand. T then runs obtainSharedSecret-T, sets its puvProtocol chosen by C and outputs its ECDH public key pk_T back to C, the response to the getKeyAgreement subcommand. At this stage, C runs two different functions. First, it executes obtainSharedSecret-C-end, which runs the puvProtocol function encapsulate. This function derives the shared secret K from the client's ECDH public key c and the received public key pk_T from the token, and outputs c. The client then runs setPIN-C, which inputs the user pin, encrypts it with K to create ciphertext c_p, and then authenticates c_p to create t_p. In the end, C sends c, c_p, t_p to the token, modeling authenticatorClientPIN with the setPIN subcommand. Finally, after receiving this data, the token executes setPIN-T, which runs puvProtocol function decapsulate to derive the same shared secret K, and uses it to verify and decrypt c_p. The resulting pin is then hashed with SHA-256, and its 16 most significant bytes (the pinHash) stored in the token's static storage st_T. The token's retry counter st_T.pinRetries is also initialized to pinRetriesMax (which is at most 8).

Client C (U, pin)		**Authenticator** T
Reboot		
		authPowerUp-T(st_T)
Setup		
	$\xleftarrow{\text{info}}$	info $\leftarrow$ getInfo-T(π_T^i)
puvProtocol $\xleftarrow{\$}$ obtainSharedSecret-C-start(π_C^j, info)		
	$\xrightarrow{\text{puvProtocol}}$	$pk_T \xleftarrow{\$}$ obtainSharedSecret-T(π_T^i, puvProtocol)
$c \xleftarrow{\$}$ obtainSharedSecret-C-end(π_C^j, pk_T)	$\xleftarrow{pk_T}$	
$(c_p, t_p) \xleftarrow{\$}$ setPIN-C(π_C^j, pin)	$\xrightarrow{c, c_p, t_p}$	status $\leftarrow$ setPIN-T(π_T^i, puvProtocol, c, c_p, t_p)
Bind		
Bind-C		Bind-T
if $\pi_C^j.\text{st}_{\text{exe}} = \perp$:	$\xleftarrow{\text{info}}$	info $\leftarrow$ getInfo-T(π_T^i)
puvProtocol $\xleftarrow{\$}$ obtainSharedSecret-C-start(π_C^j, info)		
	$\xrightarrow{\text{puvProtocol}}$	if $\pi_T^i.\text{st}_{\text{exe}} = \perp$:
if $\pi_C^j.\text{st}_{\text{exe}} =$ waiting :	$\xleftarrow{pk_T}$	$pk_T \leftarrow$ obtainSharedSecret-T(π_T^i, puvProtocol)
$c \xleftarrow{\$}$ obtainSharedSecret-C-end(π_C^j, pk_T)		
$c_{ph} \xleftarrow{\$}$ obtainPinUvAuthToken-C-start(π_C^j, pin)		
	$\xrightarrow{c, c_{ph}}$	if $\pi_T^i.\text{st}_{\text{exe}} =$ waiting :
		$c_{pt} \xleftarrow{\$}$ obtainPinUvAuthToken-T(π_T^i, puvProtocol, c, c_{ph})
if $\pi_C^j.\text{st}_{\text{exe}} =$ bindStart :	$\xleftarrow{c_{pt}}$	
obtainPinUvAuthToken-C-end(π_C^j, c_{pt})		
Auth-C		Validate-T
if $\pi_C^j.\text{st}_{\text{exe}} =$ bindDone :	$\xrightarrow{M, t}$	if $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone :
$(M, t) \xleftarrow{\$}$ auth-C(π_C^j, M)		status $\leftarrow$ validate-T(π_T^i, M, t, d)
Validate-C		Auth-T
if $\pi_C^j.\text{st}_{\text{exe}} =$ bindDone :	$\xleftarrow{R, t}$	if $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone :
status $\leftarrow$ validate-C(π_C^j, R, t)		$(R, t) \xleftarrow{\$}$ auth-T(π_T^i, R)

Figure 5.3: CTAP 2.1 as a PACA protocol, in black. The red-coloured block corresponds to our fixed CTAP 2.1+ protocol, modeled as an mPACA protocol. Complete descriptions of the algorithms are shown in Figure 5.4. We refer to the token's response as R here, to clarify that it is different from the command M sent by the client. However, in the syntax and model, we always use M for both client-to-token and token-to-client messages.

The Bind phase is essentially identical to Setup until the execution of obtainSharedSecret-C-end on the client's side. Afterwards, C runs obtainPinUvAuthToken-C-start, which encrypts the hash of the user pin, and sends its ECDH public key c and the resulting ciphertext c_{ph} to token T. This step models the authenticatorClientPIN command with the getPinToken subcommand[6]. The token executes obtainPinUvAuthToken-T, which, as long as st_T.pinRetries is not 0, runs decapsulate to obtain K, decrements st_T.pinRetries, decrypts c_{ph} and then verifies if the result matches the saved st_T.pinHash from Setup. If it does not, then it regenerates the token's ECDH public key for the currently in use puvProtocol, decrements the token's consecutive tries counter $\text{st}_T.m$ and, if $\text{st}_T.m$ reaches 0, forces a token reboot. If verification succeeds, then it samples a new pinUvAuthToken for every

[6]Or the getPinUvAuthTokenUsingPinWithPermissions subcommand, since in our model they are not differentiated.

authPowerUp-T(st_T):
1: **if** st_T.version $=\perp \wedge$ st_T.puvProtocolList $=\perp$ **then**
2: $(version, puvProtocolList) \leftarrow st_T$.initialData
3: st_T.version $\leftarrow$ version
4: st_T.puvProtocolList $\leftarrow$ puvProtocolList
5: **for all** puvProtocol $\in st_T$.puvProtocolList **do**
6: st_T.puvProtocol.initialize()
7: $st_T.m \leftarrow 3$

obtainSharedSecret-C-start(π_C^j, info):
1: Parse(version, puvProtocolList) $\leftarrow$ info
2: **if** version $= 2.0$ **then return** $\perp$
3: select puvProtocol $\leftarrow$ puvProtocolList
4: π_C^j.puvProtocol $\leftarrow$ puvProtocol
5: π_C^j.puvProtocol.initialize()
6: $\pi_C^j.st_{exe} \leftarrow$ waiting
7: π_C^j.sid $\leftarrow \pi_C^j$.sid || info || puvProtocol
8: **return** puvProtocol

obtainPinUvAuthToken-C-start(π_C^j, pin):
1: pinHash $\leftarrow$ H(pin)
2: $c_{ph} \xleftarrow{\$} \pi_C^j$.puvProtocol.encrypt($\pi_C^j$.K, pinHash)
3: $\pi_C^j.st_{exe} \leftarrow$ bindStart
4: π_C^j.sid $\leftarrow \pi_C^j$.sid || c_{ph}
5: **return** c_{ph}

obtainPinUvAuthToken-C-end(π_C^j, c_{pt}):
1: π_C^j.bs $\leftarrow \pi_C^j$.puvProtocol.decrypt(π_C^j.K, c_{pt})
2: $\pi_C^j.st_{exe} \leftarrow$ bindDone
3: π_C^j.sid $\leftarrow \pi_C^j$.sid || c_{pt} || false
4: **return**

setPIN-C(π_C^j, pin):
1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: $c_p \xleftarrow{\$} \pi_C^j$.puvProtocol.encrypt($\pi_C^j$.K, pin)
3: $t_p \xleftarrow{\$} \pi_C^j$.puvProtocol.authenticate(π_C^j.K, c_p)
4: **return** (c_p, t_p)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p):
1: **if** puvProtocol $\notin st_T$.puvProtocolList $\vee$ st_T.pinHash $\neq\perp$ **then**
2: **return** $\perp$
3: $K \leftarrow st_T$.puvProtocol.decapsulate(c)
4: **if** $K =\perp \vee st_T$.puvProtocol.verify(K, c_p, t_p) = false **then**
5: **return** $\perp$
6: pin $\leftarrow st_T$.puvProtocol.decrypt(K, c_p)
7: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
8: st_T.pinHash $\leftarrow$ H(pin)
9: st_T.pinRetries $\leftarrow$ pinRetriesMax
10: **return** accepted

auth-C (π_C^j, M):
1: $t \xleftarrow{\$} \pi_C^j$.puvProtocol.authenticate(π_C^j.bs, M)
2: **return** (M, t)

— CTAP 2.1+ below

auth-C (π_C^j, M):
1: $(K_{authC}, _) \leftarrow \pi_C^j$.puvProtocol.expand($\pi_C^j$.bs)
2: $t \xleftarrow{\$} \pi_C^j$.puvProtocol.authenticate(K_{authC}, M)
3: **return** (M, t)

validate-C (π_C^j, M, t):
1: $(_, K_{authT}) \leftarrow \pi_C^j$.puvProtocol.expand($\pi_C^j$.bs)
2: **if** π_C^j.puvProtocol.verify(K_{authT}, M, t) = true **then return** accepted
3: **return** rejected

getInfo-T(π_T^i):
1: info $\leftarrow (st_T$.version, st_T.puvProtocolList)
2: π_T^i.sid $\leftarrow \pi_T^i$.sid || info

obtainSharedSecret-T(π_T^i, puvProtocol):
1: **if** puvProtocol $\notin st_T$.puvProtocolList **then return** $\perp$
2: $pk_T \leftarrow st_T$.puvProtocol.getPublicKey()
3: $\pi_T^i.st_{exe} \leftarrow$ waiting
4: π_T^i.sid $\leftarrow \pi_T^i$.sid || puvProtocol || pk_T
5: **return** pk_T

obtainSharedSecret-C-end(π_C^j, pk_T):
1: $(c, K) \leftarrow \pi_C^j$.puvProtocol.encapsulate(pk_T)
2: π_C^j.K $\leftarrow K$
3: π_C^j.sid $\leftarrow \pi_C^j$.sid || pk_T || puvProtocol || c
4: **return** c

obtainPinUvAuthToken-T(π_T^i, puvProtocol, c, c_{ph}):
1: **if** puvProtocol $\notin st_T$.puvProtocolList $\vee$ st_T.pinRetries $= 0$ **then**
2: **return** $(\perp$, false)
3: $K \leftarrow st_T$.puvProtocol.decapsulate(c)
4: **if** $K =\perp$ **then return** $(\perp$, false)
5: st_T.pinRetries $\leftarrow$ pinRetries $- 1$
6: pinHash $\leftarrow st_T$.puvProtocol.decrypt(K, c_{ph})
7: **if** pinHash $\neq st_T$.pinHash **then**
8: st_T.puvProtocol.regenerate()
9: $st_T.m \leftarrow st_T.m - 1$
10: **if** $st_T.m = 0$ **then**
11: authPowerUp-T(st_T)
12: **return** $(\perp$, true)
13: **else**
14: **return** $(\perp$, false)
15: $st_T.m \leftarrow 3$
16: st_T.pinRetries $\leftarrow$ pinRetriesMax
17: **for all** puvProtocol' $\in st_T$.puvProtocolList **do**
18: st_T.puvProtocol'.resetPuvToken()
19: π_T^i.bs $\leftarrow \pi_T^i$.puvProtocol.pt
20: $c_{pt} \xleftarrow{\$} st_T$.puvProtocol.encrypt($K, \pi_T^i$.bs)
21: $\pi_T^i.st_{exe} \leftarrow$ bindDone
22: π_T^i.canValidate $\leftarrow$ true
23: π_T^i.sid $\leftarrow \pi_T^i$.sid || puvProtocol || c || c_{ph} || c_{pt} || false
24: **return** $(c_{pt}$, false)

validate-T (π_T^i, M, t, d):
1: **if** st_T.puvProtocol.verify(π_T^i.bs, M, t) = true **then**
2: **return** d
3: **return** rejected

— CTAP 2.1+ below

auth-T (π_T^i, M):
1: $(_, K_{authT}) \leftarrow st_T$.puvProtocol.expand($\pi_T^i$.bs)
2: $t \xleftarrow{\$} st_T$.puvProtocol.authenticate(K_{authT}, M)
3: **return** (M, t)

validate-T (π_T^i, M, t, d):
1: $(K_{authC}, _) \leftarrow st_T$.puvProtocol.expand($\pi_T^i$.bs)
2: **if** π_T^i.canValidate = true **then**
3: **if** st_T.puvProtocol.verify(K_{authC}, M, t) = true **then**
4: π_T^i.canValidate $\leftarrow$ false
5: **return** d
6: **return** rejected

Figure 5.4: CTAP 2.1 and CTAP 2.1+ protocol functions. The CTAP 2.1 version of auth-C and validate-T (as presented in [2]) is marked in orange. Code below corresponds to CTAP 2.1+, where we highlight in red the changes to the current version of the protocol: deriving two MAC keys instead of one and using them for bi-directional authentication. We also highlight the addition of canValidate in blue.

supported PIN/UV Auth Protocol, sets the correct pinUvAuthToken as its binding state, encrypts it with K and sends the resulting ciphertext c_{pt} to the client. Finally, C runs obtainPinUvAuthToken-C-end, which decrypts c_{pt} and sets the result as its binding state.

The Auth-C subprotocol phase is performed after a client C and token T have finished Bind. C can use its binding state π_C^j.bs as the key to authenticate a command M by running auth-C (the CTAP 2.1 function defined in Figure 5.4), which outputs M and a tag t. The token performs the Validate-T phase by validating (M, t) with the same binding state π_T^i.bs using the validate-T function. As aforementioned, this syntax models any command sent by the client that requires proof of user presence (i.e., $d = 1$) and user verification through the use of a pinUvAuthToken.

We describe the details of both PIN/UV Auth Protocol instances next.

Low-level CTAP 2.1 description

As introduced in Appendix B, CTAP 2.1 specifies an abstract PIN/UV Auth Protocol, which provides an interface for a set of lower level cryptographic functions, and provides two distinct instantiations, referred to as PIN/UV Auth Protocol 1 and PIN/UV Auth Protocol 2 (seen in Figures 5.6 and 5.5 respectively). We start by first describing PIN/UV Auth Protocol 2, which is the instance on which we focused our security analysis of CTAP 2.1, and on top of which we based our proposed CTAP 2.1+, and then describe PIN/UV Auth Protocol 1 for completeness.

PIN/UV Auth Protocol 2 The initialize function, which is called by a token on reboot or a client when starting a new Bind run, generates a fresh ECDH key pair over the NIST P-256 curve and a fresh 2λ-bit pinUvAuthToken (with $\lambda = 128$). Both actions can also be executed separately via regenerate and resetPuvToken, respectively. The generated ECDH public key can then be obtained (but not regenerated) via getPublicKey. The encrypt and decrypt functions both input a 4λ-bit key K and a message m or ciphertext c, then take the last 2λ bits as K_2, and use K_2 to respectively encrypt m or decrypt c, using the underlying encryption scheme SKE, instantiated as AES-256 in CBC-mode with random IV. The authenticate function inputs a key K, which can be 2λ or 4λ bits long (depending on whether the pinUvAuthToken is the key or if a session key from Setup is used instead) and a message m, and uses the first 2λ bits as a key to authenticate m, producing a tag t using the underlying MAC. The verify function is almost identical, but also inputs a tag t and simply outputs if t is a valid tag for m or not. In both cases, the underlying MAC is HMAC-SHA-256 using a 256-bit key. The encapsulate and decapsulate functions perform

initialize ():
1: regenerate()
2: resetPuvToken()

expand (pt):
1: $pt_e \leftarrow \mathcal{H}_3(pt)$
2: Parse$(K_C, K_T) \leftarrow pt_e$, s.t. $|K_C| = 2\lambda$
3: **return** (K_C, K_T)

encrypt (K, m):
1: Parse$(K_1, K_2) \leftarrow K$, s.t. $|K_1| = 2\lambda$
2: $c \leftarrow$ SKE.Enc(K_2, m)
3: **return** c

encapsulate (pk'):
1: $Z \leftarrow$ XCoordinateOf$(sk \cdot pk')$
2: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
3: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
4: $K \leftarrow (K_1, K_2)$
5: $c \leftarrow pk$
6: **return** (c, K)

authenticate (K', m):
1: Parse$(K'_1, K'_2) \leftarrow K'$, s.t. $|K'_1| = 2\lambda$
2: $t \leftarrow$ MAC(K'_1, m)
3: **return** t

getPublicKey ():
1: **return** pk

regenerate ():
1: $(pk, sk) \xleftarrow{\$}$ ECDH.KG()

resetPuvToken ():
1: $pt \xleftarrow{\$} \{0,1\}^{2\lambda}$

decrypt (K, c):
1: Parse$(K_1, K_2) \leftarrow K$, s.t. $|K_1| = 2\lambda$
2: $m \leftarrow$ SKE.Dec(K_2, c)
3: **return** m

decapsulate (c):
1: $Z \leftarrow$ XCoordinateOf$(sk \cdot c)$
2: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
3: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
4: $K \leftarrow (K_1, K_2)$
5: **return** K

verify (K', m, t):
1: Parse$(K'_1, K'_2) \leftarrow K'$, s.t. $|K'_1| = 2\lambda$
2: $t' \leftarrow$ MAC(K'_1, m)
3: **return** $[[t = t']]$

Figure 5.5: The PIN/UV Auth Protocol 2. All original functions are presented as shown in [2], with our proposed addition of expand for CTAP 2.1+ in red.

encrypt (K, m):
1: $c \leftarrow$ SKE.Enc(K, m)
2: **return** c

encapsulate (pk'):
1: $Z \leftarrow$ XCoordinateOf$(sk \cdot pk')$
2: $K \leftarrow \mathcal{H}_1(Z)$
3: $c \leftarrow pk$
4: **return** (c, K)

authenticate (K', m):
1: $t \leftarrow$ MAC(K', m)
2: **return** t

decrypt (K, c):
1: $m \leftarrow$ SKE.Dec(K, c)
2: **return** m

resetPuvToken ():
1: $pt \xleftarrow{\$} \{0,1\}^{\mu\lambda}$

decapsulate (c):
1: $Z \leftarrow$ XCoordinateOf$(sk \cdot c)$
2: $K \leftarrow \mathcal{H}_1(Z)$
3: **return** K

verify (K', m, t):
1: $t' \leftarrow$ MAC(K', m)
2: **return** $[[t = t']]$

Figure 5.6: The PIN/UV Auth Protocol 1. Only functions that are different from PIN/UV Auth Protocol 2 are shown.

the ECDH key exchange and then produce K_1 and K_2 by passing the ECDH result through a KDF (modeled by $\mathcal{H}_2$), instantiated as HKDF-SHA-256. In encapsulate, which is only executed by a client, both $K = (K_1, K_2)$ and the ECDH public key c are output, while in decapsulate, which is only executed by the token, only K is returned.

PIN/UV Auth Protocol 1 This instance is identical to PIN/UV Auth Protocol 2 in the initialize, getPublicKey and regenerate functions. The resetPuvToken function is almost identical, but can sample a fresh pinUvAuthToken that is λ-bit *or* 2λ-bit in length. The encrypt and

decrypt functions receive a 2λ-bit key K and use it directly to encrypt or decrypt the input m or c. SKE is instantiated as AES-256 in CBC-mode with ***zero*** IV. The authenticate and verify functions use a key K (which can be the same key used for encrypt or decrypt) to authenticate a message m, producing a λ-bit tag t. In practice, t is the result of truncating the leftmost λ bits from the HMAC-SHA-256 2λ-bit output. Finally, encapsulate and decapsulate perform the same ECDH key exchange, but pass the ECDH shared secret through SHA-256 (modeled by $\mathcal{H}_1$) to produce a single, 2λ-bit symmetric key K.

5.3.4. CTAP 2.1 authentication security

The CTAP 2.1 authentication security proof in [2] undoubtedly gives a valuable and comprehensive security analysis of CTAP 2.1 that broadens the coverage and corrects some aspects in the initial proof given in [1]. Nevertheless, it still has some shortcomings that we fix in our proof. In the following, we first discuss two shortcomings with the proof and their implications, especially regarding the new *trusted binding* setting introduced in [2], and then show our security results.

Proof Shortcomings

Active attacks against clients during Bind The first shortcoming, and the most impactful, is related to the updated *trusted binding* setting the authors introduced in their security analysis, where an adversary can actively interact with client sessions via the Send-Bind-C oracle to deliver the final message of Bind (the encrypted pinUvAuthToken c_{pt}). As it turns out, this new setting is too strong, making it impossible to prove CTAP 2.1 secure in this model.

The problem with the proof lies in a step of the CTAP 2.1 proof [2, Appendix I, Game 16] that justifies removing PIN hashes from CBC encryptions during Bind. There, it is argued that CBC security guarantees that an attacker delivering a mauled ciphertext back to the client in the final step of the Bind subprotocol has no information on the resulting decrypted pinUvAuthToken. We show that this is not true by presenting a simple attack, inspired by CBC padding oracle attacks [41].

CTAP 2.1 with PIN/UV Auth Protocol 2 uses AES-256 in CBC mode with random IV to encrypt the 128-bit pinHash that is sent from a client to a token during Bind and, immediately after that, to encrypt the 256-bit pinUvAuthToken back to the client. Crucially, both encryptions use the same symmetric key (this is true for both PIN/UV Auth Protocol versions). Consider the ciphertext $c_{ph} = \mathsf{IV} \,||\, \mathsf{AES}(\mathsf{IV} \oplus \mathsf{pinHash})$, generated by a client session π_C^j in

obtainPinUvAuthToken-C-start. If the hash of the PIN checks out, π_T^i will accept and output a ciphertext c_{pt}: this is a CBC encryption with the same key that decrypted c_{ph}, and contains a fresh IV and two blocks encoding a pinUvAuthToken to be decrypted by π_C^j. Now, $\mathcal{A}$ can create a new ciphertext $\tilde{c}_{pt} = \text{IV} \,||\, \text{AES}(\text{IV} \oplus \text{pinHash}) \,||\, \text{AES}(\text{IV} \oplus \text{pinHash})$ and deliver $\tilde{c}_{pt}$ to π_C^j. Since AES in CBC mode is not an authenticated encryption scheme, π_C^j always decrypts $\tilde{c}_{pt}$, recovering a mauled pinUvAuthToken of the form $\pi_C^j.\text{bs} = \text{pinHash} \,||\, \text{IV} \oplus \text{pinHash} \oplus \text{AES}(\text{IV} \oplus \text{pinHash})$. Formally, and this is where the proof in [2] is incorrect, this means that the recovered pinUvAuthToken can actually depend on the user's PIN. In practice, this means that when the client issues a command authenticated with this mauled pinUvAuthToken, producing a tag t, the adversary can use the MAC verification algorithm to perform an *offline* dictionary attack, by hashing values in the PIN space and checking if they produce an identical tag on the same command issued by the client. This is possible because the pinHash is the only part of the mauled pinUvAuthToken the adversary does not know. This will allow the adversary to win the game with probability close to 1, and so there is no hope of proving CTAP 2.1 secure in such a model.

Note that, while we describe this attack with PIN/UV Auth Protocol 2, the situation is identical with PIN/UV Auth Protocol 1, except that CBC uses zero IV and the pinUvAuthToken *may* be 128-bit in length. In that case, $\tilde{c}_{pt}$ has only one block, and is in fact identical to c_{ph}, which π_C^j decrypts into the pinHash and sets as the binding state.

Also note that this captures the scenario where an attacker may be incapable of performing a man-in-the-middle attack, but still has the ability to inject the last message sent by the authenticator during Bind. As pointed out in Chapter 4, this is an excelent use case for the aforementioned USB injection attacks, because the encrypted pinUvAuthToken is the last message sent by the authenticator during Bind, and therefore an attacker equipped with such a malicious USB device could program it to hijack the legitimate ciphertext and replace it with the mauled ciphertext.

IND-1CPA is not enough The second shortcoming is related to a minor oversight we identified in Game 12 of the same proof, that applies only to unlikely cases where tokens do not refresh their ECDH public keys after Setup. We explain this next.

The reduction to the security of the symmetric encryption scheme used to communicate with tokens has been improved in [2] compared to [1]. In particular, it was shown that, during Bind, a stronger (plaintext-checking) assumption was needed to deal with active attacks on the token. However, this was considered to be unnecessary during Setup,

which is not the case if ECDH public keys are *reused* by the token. Indeed, while it is true that a query to Setup always results in a new ECDH key exchange between π_C^j and π_T^i, it cannot be said, even when excluding collisions between ECDH public keys and symmetric keys, that the symmetric key used to encrypt the pin sent to π_T^i will not be used again later in the experiment. Since every Setup trace is given to $\mathcal{A}$, the adversary can attempt to start a Bind run with another token session $\pi_T^{i'}$ of the same token, using a public key that was previously used by π_C^j during Setup. If T did not regenerate its public key, then it will derive the same symmetric key that was used in Setup, and therefore might encrypt a pinUvAuthToken with the same key used to encrypt the pin in Setup. Therefore, the encryption scheme that encrypts the user pin during Setup must be IND-1$PA-LHPC secure (see Chapter 2), rather than only IND-1CPA. This is a minor change to the proof, which we handle, and would not be necessary if the token was guaranteed to reset its state (via Reboot) after Setup or simply regenerate its ECDH public key before or after every key exchange.

Security results and proof

We now state our authentication security result for CTAP 2.1, cast as a PACA protocol, and instantiated with PIN/UV Auth Protocol 2. In Appendix C, we present how its security can be proven when instantiated with PIN/UV Auth Protocol 1, by adapting our proof for Theorem 5.1 shown below.

We assume that every user pin is sampled according to some distribution $\mathscr{D}$ with min-entropy $h_{\mathscr{D}}$ over the set $\mathcal{P}$ of all valid pins. We also assume that HKDF-SHA-256 [42] is modeled as a random oracle $\mathcal{H}_2$. Then, for other CTAP 2.1 building blocks, hash function H outputs the leftmost 128 bits of the SHA-256 digest, q is the prime order of the underlying elliptic-curve Diffie-Hellman (ECDH) group (for NIST curve P-256) [43], the underlying symmetric encryption scheme SKE denotes AES-256 in CBC mode with random IV [44], and the message authentication code MAC denotes HMAC-SHA-256 with 256-bit keys [45]. A query to **Send-Bind-T** on a session π_T^i is considered to be *active* if $\mathcal{A}$ delivers an ECDH public key c and ciphertext c_{ph} to π_T^i and (c, c_{ph}) was not output by any client session that was involved in a previous **Execute** query with a token session using the same token ECDH public key. The following theorem states the SUF-t security of CTAP 2.1.

Theorem 5.1. *For every efficient adversary $\mathcal{A}$ that makes at most q_S, q_E, q_{Send}, q_{NT} and q_R queries to Setup, Execute, Send-Bind-T, NewT and Reboot, and at most q_{Send}^{act} active queries*

to Send-Bind-T, there exist efficient adversaries $\mathcal{B}_1$, $\mathcal{B}_4$, $\mathcal{B}_5$, $\mathcal{B}_6$ and $\mathcal{B}_{10}$ such that:

$$\begin{aligned}
Adv^{SUF\text{-}t}_{CTAP\ 2.1}(\mathcal{A}) \leq\ & q^{act}_{Send}/2^{h_{\mathscr{D}}} + (q_S + q_E + q_{Send})\ Adv^{sCDH}_{ECDH}(\mathcal{B}_1) \\
& + (q_S + q_E + q_{NT} + q_R + 2q_{Send})^2/(2q) + Adv^{coll}_{H}(\mathcal{B}_4) \\
& + q_S\ Adv^{IND\text{-}1\$PA\text{-}LHPC}_{SKE}(\mathcal{B}_5) + q_E\ Adv^{IND\text{-}1\$PA\text{-}LPC}_{SKE}(\mathcal{B}_6) \\
& + (q_E + q_{Send})^2/(2^{2\lambda+1}) + (q_E + q_{Send})\ Adv^{SUF\text{-}CMA}_{MAC}(\mathcal{B}_{10})\ .
\end{aligned}$$

We now show our proof sketch for Theorem 5.1, and refer to Appendix E for a complete, detailed pseudocode proof[7]. All security definitions of the underlying primitives used in the theorem can be consulted in Chapter 2.

Proof. (Sketch.) Let Pr_i denote the probability that Game i outputs 1.

Game 0 This is the original experiment. Therefore,

$$\mathsf{Pr}_0 = \mathsf{Adv}^{\mathsf{SUF\text{-}t}}_{\mathsf{CTAP}}(\mathcal{A})\ .$$

Game 1 In this game, we replace every symmetric session key K_1 and K_2 calculated in encapsulate and decapsulate from Figure 5.5 as $K_1 \leftarrow \mathcal{H}_2(Z, \cdot)$ and $K_2 \leftarrow \mathcal{H}_2(Z, \cdot)$, where Z is the ECDH shared secret derived by a client session π^j_C and token session π^i_T in a **Setup** or **Execute** query, with independent random values $\tilde{K_1}$ and $\tilde{K_2}$. We define bad as the event that, at some point during the game, π^j_C and π^i_T are involved in a **Setup** or **Execute** and $\mathcal{A}$ queries $\mathcal{H}_2$ with Z, and bound the probability of bad by constructing an adversary $\mathcal{B}_1$ against the sCDH security of the underlying ECDH group, such that if $\mathcal{A}$ causes bad, $\mathcal{B}_1$ wins the sCDH game. Since token public keys are only given to $\mathcal{A}$ in **Setup**, **Execute** and **Send-Bind-T** oracles, we have $|\mathsf{Pr}_0 - \mathsf{Pr}_1| \leq (q_S + q_E + q_{Send})\ \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1)$.

Game 2 In this game, in every **Setup** query, the pin is no longer stored in the token state st_T, instead being stored in a new strucuture accessible only by the challenger. Since this model never allows $\mathcal{A}$ to corrupt any token, Games 1 and 2 are functionally identical. Therefore, $\mathsf{Pr}_1 = \mathsf{Pr}_2$.

[7] The pseudocode proof in the Appendix refers to CTAP 2.1+. However, due to the the protocols being identical until Bind is finished, the only difference in the proofs comes after Game 9, with the CTAP 2.1 proof only requiring one additional game and the CTAP 2.1+ proof requiring three.

Game 3 In this game, $\mathcal{A}$ immediately loses if the challenger samples two identical ECDH public keys from a client or token. Given that client ECDH public keys are sampled only in **Setup** and **Execute** queries, and token ECDH public keys are sampled in **NewT**, **Reboot** and **Send-Bind-T**[8] queries, we have $|\mathsf{Pr}_2 - \mathsf{Pr}_3| \leq (q_\mathsf{S} + q_\mathsf{E} + q_\mathsf{NT} + q_\mathsf{R} + 2q_{\mathsf{Send}})^2 \ / \ (2q)$.

Game 4 In this game, $\mathcal{A}$ loses if a collision occurs on some output of H when called by the challenger. Since H is assumed to be collision resistant, there exists an adversary $\mathcal{B}_4$ such that $|\mathsf{Pr}_3 - \mathsf{Pr}_4| \leq \mathsf{Adv}^{\mathsf{coll}}_{\mathsf{H}}(\mathcal{B}_4)$.

Game 5 This game replaces every encrypted pin in **Setup** queries with an encrypted constant pin 0000. Additionally, if the ECDH client and token public keys used in **Setup** are used again in **Send-Bind-T**, the token encrypts and outputs a random pinUvAuthToken $\tilde{pt}$ while setting the real pinUvAuthToken as its binding state. We bound the advantage of $\mathcal{A}$ against Game 5 by reduction to the IND-1$PA-LHPC security of the underlying encryption scheme SKE. Therefore, we construct an adversary $\mathcal{B}_5$ against the IND-1$PA-LHPC security of SKE such that $|\mathsf{Pr}_4 - \mathsf{Pr}_5| \leq q_\mathsf{S} \ \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5)$.

Game 6 This game replaces every encrypted hashed pin $\mathsf{H}(\mathsf{pin})$ and pinUvAuthToken pt in **Execute** queries with an encrypted hashed constant $\mathsf{H}(0000)$ and a random pinUvAuthToken $\tilde{pt}$ respectively. We bound the advantage of $\mathcal{A}$ against Game 6 by reduction to the IND-1$PA-LPC security of the underlying encryption scheme SKE. Therefore, we construct an adversary $\mathcal{B}_6$ against the IND-1$PA-LPC security of SKE such that $|\mathsf{Pr}_5 - \mathsf{Pr}_6| \leq q_\mathsf{E} \ \mathsf{Adv}^{\text{IND-1\$PA-LPC}}_{\mathsf{SKE}}(\mathcal{B}_6)$.

Game 7 In this game, user pins are no longer sampled in **NewU**, being sampled instead only when relevant to answer to some query from $\mathcal{A}$, which happens only in **Corrupt** and **Send-Bind-T** queries. Since every pin is independently sampled from $\mathscr{D}$, it is independent from anything that happens during the experiment, and thus Games 6 and 7 are functionally identical. Therefore, $\mathsf{Pr}_6 = \mathsf{Pr}_7$.

Game 8 In this game, the challenger rejects any attempt from $\mathcal{A}$ to actively guess the correct pin, unless the pin for the target token has been corrupted. By considering $q^{\mathsf{act}}_{\mathsf{Send}}$ the maximum number of active queries to **Send-Bind-T**, where $\mathcal{A}$ actively attempts at

[8] Every time a pinHash verification fails in obtainPinUvAuthToken-T (which is called in **Send-Bind-T**), the regenerate function is called. However, if this verification forces a reboot, due to the consecutive retries counter $\mathsf{st}_T.m$ reaching 0, there might be a new ECDH public key regeneration, which is why the upper bound accounts for 2 queries to **Send-Bind-T**.

guessing the pin, and $1/2^{h_{\mathscr{D}}}$ as the maximum probability that $\mathcal{A}$ can guess the pin in any query, we have $|\mathsf{Pr}_7 - \mathsf{Pr}_8| \leq q_{\mathsf{Send}}^{\mathsf{act}}/2^{h_{\mathscr{D}}}$.

Game 9 In this game, $\mathcal{A}$ loses if two pinUvAuthTokens that are sent to the adversary in either an **Execute** or **Send-Bind-T** query collide. Since, at most, one pinUvAuthToken is generated and given to $\mathcal{A}$ in each **Execute** or **Send-Bind-T** query, we have $|\mathsf{Pr}_8 - \mathsf{Pr}_9| \leq (q_{\mathsf{E}} + q_{\mathsf{Send}})^2 \ / \ 2^{2\lambda+1}$.

Game 10 In this game, when $\mathcal{A}$ queries **Validate-T** with message M and a tag t on a token session π_T^i that finished Bind passively (which happens *only* in **Execute** and passive **Send-Bind-T** queries), such that (M, t) would constitute a valid forgery, the challenger still rejects (recall that $\mathcal{A}$ cannot win against any token session π_T^i that was actively attacked in **Send-Bind-T** queries, since after Game 8 this means T's pin must have been corrupted). We consider this bad event and bound the advantage of $\mathcal{A}$ by reducing to the SUF-CMA security of the underlying MAC, by constructing an efficient adversary $\mathcal{B}_{10}$ against the SUF-CMA security of MAC such that if bad happens, $\mathcal{B}_{10}$ wins the SUF-CMA game. Therefore, we have $|\mathsf{Pr}_9 - \mathsf{Pr}_{10}| \leq (q_{\mathsf{E}} + q_{\mathsf{Send}})\ \mathsf{Adv}_{\mathsf{MAC}}^{\text{SUF-CMA}}(\mathcal{B}_{10})$.

Final analysis After the modifications in Game 10, $\mathcal{A}$ can no longer win. Indeed, for $\mathcal{A}$ to win, it must satisfy at least one of the four conditions in Token-Win-SUF-t. The first condition is always false, since $d = 1$ must always be true for any **Validate-T** query to accept a tag t. The second and third conditions are also always false. Collisions between sid values in client sessions have been ruled out in Game 3, by avoiding client public key collisions, while collisions between sid values in token sessions have been ruled out in Game 9, by removing collisions between any pinUvAuthToken that is encrypted and sent to $\mathcal{A}$. Therefore, it can never be the case that two different client or token sessions have the same sid. Finally, $\mathcal{A}$ can never win via the last condition, because in Game 10 every valid forgery attempt against any token session via a **Validate-T** query is rejected by the challenger. Therefore, it is always true that $\mathsf{Pr}_{10} = 0$. Note that the running times of all adversaries are close to that of $\mathcal{A}$. □

Remark In Game 8, we bound $\mathcal{A}$'s advantage as a function of $q_{\mathsf{Send}}^{\mathsf{act}}$, i.e., how many active attacks $\mathcal{A}$ makes against *any* token. This captures the worst case scenario where $\mathcal{A}$ can always reset pinRetries by executing a passive Bind for *any* token (i.e., the adversary may have access to any user from any token to input the correct pin). However, it is also reasonable to consider the case where $\mathcal{A}$ has access to an arbitrary number of tokens but

not to their users, which limits the amount of active attacks on any token to pinRetriesMax. Considering $q_{\mathsf{NT}}^{\mathsf{act}}$ as the number of tokens created that are actively attacked by $\mathcal{A}$ during the experiment, we could write the bound as $|\mathsf{Pr}_7 - \mathsf{Pr}_8| \leq \mathsf{pinRetriesMax} \cdot q_{\mathsf{NT}}^{\mathsf{act}} / 2^{h_{\mathscr{D}}}$.

5.3.5. CTAP 2.1+ as an mPACA protocol

Just as mPACA is an extension of a PACA protocol, introducing two new subprotocols Auth-T and Validate-C for bi-directional authentication in a client-token communication channel, CTAP 2.1+ is an extension of CTAP 2.1 that adds two new functions auth-T and validate-C (see Figure 5.4) for authenticating and validating a token response, respectively.

Session and protocol variables

CTAP 2.1+ has the same session and protocol variables as in CTAP 2.1.

High-level flow and syntax of CTAP 2.1+

The CTAP 2.1+ protocol is identical to CTAP 2.1 until Bind is finished. The modifications (marked in red) can be seen in Figures 5.3, 5.4 and 5.5. After a client C and token T have finished Bind, C runs auth-C, which expands the binding state π_C^j.bs to produce a new authentication key K_{authC}, and uses K_{authC} to produce a tag t on message M. Afterwards, the token validates (M, t) with the same binding state by running validate-T, which recovers K_{authC} to perform the verification of (M, t). After this verification, the token runs auth-T, which expands the binding state π_T^i.bs into a new authentication key K_{authT} (different from K_{authC}) and uses it to produce a new tag t on its response R. Finally, the client receives (R, t) and runs validate-C, which obtains K_{authT} to perform its verification. Notice that the reason we expand the binding state during the command and response authentication is to minimize the modifications needed to achieve mutual authentication in CTAP 2.1+ as much as possible. This expansion allows us to ensure that every part of CTAP 2.1 up until the authentication of commands and responses is exactly the same, including the pinUvAuthToken transmission from a token to the client. This results in a reduction of the amount of overhead necessary to implement our modifications in the protocol.

Low-level CTAP 2.1+ description

For CTAP 2.1+, we propose a PIN/UV Auth Protocol which is identical to the CTAP 2.1 PIN/UV Auth Protocol 2, but with the addition of a new function expand (see Figure 5.5).

This function inputs a 2λ-bit pinUvAuthToken pt (binding state), produces a 4λ-bit expanded pinUvAuthToken pt_e with a key derivation function modeled by $\mathcal{H}_3$, and then parses pt_e into two 2λ-bit keys, to be used for authenticating client-to-token and token-to-client messages. In practice, $\mathcal{H}_3$ could be instantiated with HKDF-SHA-256 called twice with two labels distinct from those in CTAP 2.1, similar to how (K_1, K_2) are derived, or simply instantiated with SHA-512.

Overhead over CTAP 2.1

One of the core goals for this extension was to reduce the necessary changes to CTAP 2.1's specification and codebase to a minimum. As a result, and as mentioned above, the entire Bind process is completely unaltered (as is Setup and Reboot). All changes occur in the auth-C and validate-T functions, which see only minor changes to expand the pinUvAuthToken and use one of the resulting keys, and on the two new functions auth-T and validate-C, which mirror auth-C and validate-T. Furthermore, while the pinUvAuthToken expansion is new, it can easily be instantiated with HKDF-SHA-256, which is already used to expand the shared secret from the ECDH key exchange.

As a result, CTAP 2.1+ can be made to work seamlessly with CTAP 2.1 clients and tokens, because it has backwards compatibility. Tokens inform the client of their supported CTAP2 versions with the authenticatorGetInfo command. If a token does not support CTAP 2.1+, the client just sends the normal CTAP 2.1 authenticated command, and does not expect an authenticated response. If the client does not support it, it will ignore the token's supported CTAP 2.1+ version, and send the authenticated command as in CTAP 2.1. The token validates the command with the pinUvAuthToken (i.e., without the expansion), and does not produce an authenticated response.

The CTAP 2.1+ additions also have minimal overhead in bandwidth and computational cost, and do not require adding any new cryptographic primitives to the existing CTAP 2.1 codebase, which further facilitates their integration in existing FIDO2 commercial implementations. We expand on this subject in Chapter 7.

5.3.6. CTAP 2.1+ authentication security

The previous proof shows CTAP 2.1 satifies SUF-t security in the PACA model. However, it does not capture rogue key attacks. Simply put, barring collisions between sid values, an adversary wins against the SUF-t experiment in the PACA model by forging (or guessing) a command tag that is accepted by a token session π_T^i. The PACA security model does

not account for what happens after a token session accepts. It models neither a token session's response nor its reception by the partner client session. Our mPACA model fixes this by introducing a new winning condition similar to the one from Token-Win-SUF-t: an adversary wins if it can make a client session π_C^j accept a response and tag that did not come from its partner token session π_T^i, provided π_C^j was not compromised and π_T^i's pin was not corrupted.

It is clear that CTAP 2.1 cannot be proven secure in our stronger mPACA security model. An adversary can always win via this condition with probability 1 by delivering any message to a client session, because CTAP 2.1 does not perform any client-side verification on a token's response (this is modeled with the Validate-C oracle always outputing 1).

CTAP 2.1+ security proof

We were able to prove that CTAP 2.1+ offers resistance to rogue key attacks, by showing that it satisfies SUF-t security in our stronger mPACA model. The following theorem states our result for CTAP 2.1+. Following the same logic from the CTAP 2.1 proof, we model the KDF used in the expand function as a random oracle $\mathcal{H}_3$.

Theorem 5.2. *For every efficient adversary $\mathcal{A}$ that makes at most q_S, q_E, q_{Send}, q_{NT} and q_R queries to Setup, Execute, Send-Bind-T, NewT and Reboot, at most q_{Send}^{act} active queries to Send-Bind-T, and at most $q_{\mathcal{H}_3}$ queries to $\mathcal{H}_3$, there exist efficient adversaries $\mathcal{B}_1$, $\mathcal{B}_4$, $\mathcal{B}_5$, $\mathcal{B}_6$, $\mathcal{B}_{11}$ and $\mathcal{B}_{12}$ such that:*

$$\begin{aligned}
Adv^{SUF\text{-}t}_{CTAP\ 2.1+}(\mathcal{A}) \leq\ & q_{Send}^{act}/2^{h_{\mathscr{D}}} + (q_S + q_E + q_{Send})\,Adv^{sCDH}_{ECDH}(\mathcal{B}_1) \\
& + (q_S + q_E + q_{NT} + q_R + 2q_{Send})^2/(2q) + Adv^{coll}_{H}(\mathcal{B}_4) \\
& + q_S\ Adv^{IND\text{-}1\$PA\text{-}LHPC}_{SKE}(\mathcal{B}_5) + q_E\ Adv^{IND\text{-}1\$PA\text{-}LPC}_{SKE}(\mathcal{B}_6) \\
& + (q_E + q_{Send})^2/(2^{2\lambda+1}) + q_{\mathcal{H}_3}(q_E + q_{Send})/2^{2\lambda} \\
& + (q_E + q_{Send})\ Adv^{SUF\text{-}CMA}_{MAC}(\mathcal{B}_{11}) + q_E\ Adv^{SUF\text{-}CMA}_{MAC}(\mathcal{B}_{12})\ .
\end{aligned}$$

The security proof for CTAP 2.1+ is identical to the proof of CTAP 2.1 until Game 9. We summarize the modifications and extra steps next, and refer to the pseudocode proof in Appendix E for more details.

Proof. (Sketch.) Let $\Pr_i$ denote the probability that Game i outputs 1. Games 0 through 9 are identical to the CTAP 2.1 proof.

Game 10 In this game, $\mathcal{A}$ loses if it queries random oracle $\mathcal{H}_3$ with any pinUvAuthToken generated throughout the experiment and set as the binding state of a token or client session, but for which the adversary has no information. These pinUvAuthTokens correspond to sessions where the adversary behaved passively, which includes all of the Execute queries, and possibly some of the Send-Bind-T queries. The two games are identical until bad, and so we have $|\mathsf{Pr}_9 - \mathsf{Pr}_{10}| \leq q_{\mathcal{H}_3}\ (q_{\mathsf{E}} + q_{\mathsf{Send}})/2^{2\lambda}$. After this game, the MAC keys used by these sessions to authenticate commands are information theoretically hidden from the adversary.

Game 11 In this game, when $\mathcal{A}$ queries Validate-T with message M and a tag t on a token session π_T^i that finished Bind passively (which happens *only* in Execute and passive Send-Bind-T queries), such that (M, t) would constitute a valid forgery, the challenger still rejects (recall that $\mathcal{A}$ cannot win against any token session π_T^i that was actively attacked in Send-Bind-T queries, since after Game 8 this means T's pin must have been corrupted). We consider this bad event and bound the advantage of $\mathcal{A}$ by reducing to the SUF-CMA security of the underlying MAC, by constructing an efficient adversary $\mathcal{B}_{11}$ against the SUF-CMA security of MAC such that if bad happens, $\mathcal{B}_{11}$ wins the SUF-CMA game. Therefore, we have $|\mathsf{Pr}_{10} - \mathsf{Pr}_{11}| \leq (q_{\mathsf{E}} + q_{\mathsf{Send}})\ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_{11})$.

Game 12 In this game, when $\mathcal{A}$ queries Validate-C with message M and a tag t and a client session π_C^j that finished Bind (which happens only in Execute queries), such that (M, t) would constitute a valid forgery, the challenger still rejects. Much like in Game 11, we bound the advantage of $\mathcal{A}$ against Game 12 via reduction to the SUF-CMA security of MAC. Therefore, there exists an efficient adversary $\mathcal{B}_{12}$ such that $|\mathsf{Pr}_{11} - \mathsf{Pr}_{12}| \leq q_{\mathsf{E}}\ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_{12})$.

Final analysis After the modifications in Game 12, $\mathcal{A}$ can no longer win. Indeed, for $\mathcal{A}$ to win, it must satisfy at least one of the four conditions in Token-Win-SUF-t or one of the three conditions in Client-Win-SUF-t. The reasoning for $\mathcal{A}$ not being able to win via Token-Win-SUF-t is identical to the reasoning used for the previous CTAP 2.1 security proof. Additionally, $\mathcal{A}$ cannot win Game 12 via the conditions from Client-Win-SUF-t. Indeed, the first and second conditions are identical to the second and third conditions from Token-Win-SUF-t, and are therefore always false, since no collisions between client session sid values and token session sid values can occur after Games 3 and 9 respectively. $\mathcal{A}$ can also never win via the third condition in Client-Win-SUF-t, because in Game 12 every valid forgery attempt against any client session via Validate-C query is rejected

by the challenger. Therefore, it is always true that $\Pr_{12} = 0$. Note that the running times of all adversaries are close to that of $\mathcal{A}$. This concludes the proof. □

In the next Section, we introduce the syntax for PIA protocols, a simplified syntax for WebAuthn as a PIA protocol, and the PIA security model we considered for our analysis of the composed PIA+mPACA protocol.

5.4. PIA and WebAuthn

As mentioned in the beginning of the Chapter, we do not claim new security results for WebAuthn, and therefore do not provide a new security analysis for the *passwordless authentication* security of WebAuthn as we did for CTAP 2.1. Nevertheless, it is still important to formally introduce what a PIA protocol is, how PIAinstantiates WebAuthn, and the security models on which WebAuthn has been proven secure in prior work.

Furthermore, we also introduce our model which is adapted from [39], simplified to only capture attestation types None, Self and Basic. This is relevant for completeness, but also for the security analysis of the composed PIA+mPACA protocol we present in Section 5.5, which combines both the mPACA model from Section 5.3 and the PIA model we introduce here.

5.4.1. Syntax for PIA protocols

The PIA syntax we define here is very similar to the ePIA protocol defined in [2] and the ePIAA protocol defined in [39]. We simply call our PIA primitive a PIA protocol. When we say an algorithm inputs a server, a client, or a token, we mean all of its state. The outputs of the algorithm make it explicit which parts of the state may be changed that are relevant for the security game.

A PIA protocol PIA consists of two subprotocols: Register and Authenticate.

Register: a two-pass challenge-response protocol run among a token T, a client C, and a server S, which is run at most once per tuple (T, S). At the end of Register, both T and S hold registration contexts, which are relevant for subsequent authentications. Register can be decomposed into the following algorithms:

$m_{\mathsf{rch}} \xleftarrow{\$} \mathsf{rChal}(S, \mathsf{tb}, \mathsf{UV})$: inputs a server S, a token binding state tb, and a user verification condition $\mathsf{UV} \in \{\mathsf{T}, \mathsf{F}\}$, and outputs a challenge message m_{rch}. It does not change the state of the server S.

$(m_{\mathsf{rcl}}, m_{\mathsf{rcom}}) \xleftarrow{\$} \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$: run by the stateless client; it inputs the intended server identity id_S, a challenge message m_{rch}, and a token binding state tb, and outputs a client message m_{rcl} and a command message m_{rcom}.

$(m_{\mathsf{rrsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{rRsp}(T, m_{\mathsf{rcom}})$: inputs a token T and a command message m_{rcom} and outputs a response message m_{rrsp}, the token-side registration context rc_T, a credential identifier cid, a session identifier sid, and agreed contents agCon from the perspective of the token T.

$(b, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{rVrfy}(S, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}}, \mathsf{gpars})$: inputs a server S, a client message m_{rcl}, a response message m_{rrsp}, and attestation group parameters gpars, and outputs a bit $b \in \{0,1\}$ to indicate whether the registration request was accepted. It also outputs the server-side context rc_S, a credential identifier cid, a session identifier sid, and agreed contents agCon from the perspective of the server S.

Authenticate: a two-pass challenge-response protocol run among a token T, a client C, and a server S after a successful run of Register, in which both T and S generated their registration contexts. At the end of Authenticate, S either accepts or rejects the authentication attempt. Similarly to Register, Authenticate can be decomposed into four algorithms:

$m_{\mathsf{ach}} \xleftarrow{\$} \mathsf{aChal}(S, \mathsf{tb}, \mathsf{UV})$: inputs a server S, a token binding state tb, and a user verification condition $\mathsf{UV} \in \{\mathsf{T}, \mathsf{F}\}$, and outputs a challenge message m_{ach}. This algorithm does not change the state of the server S.

$(m_{\mathsf{acl}}, m_{\mathsf{acom}}) \xleftarrow{\$} \mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$: run by the stateless client; it inputs the intended server identity id_S, a challenge message m_{ach}, and a token binding state tb, and outputs a client message m_{acl} and a command message m_{acom}.

$(m_{\mathsf{arsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{aRsp}(T, m_{\mathsf{acom}})$: inputs a token T, and a command message m_{acom}, and outputs a response message m_{arsp}, the updated token-side registration context rc_T, a credential identifier cid, a session identifier sid, and agreed contents agCon from the perspective of the token T.

$(b, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{aVrfy}(S, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$: inputs a server S, a client message m_{acl}, and a response message m_{arsp}, and outputs a bit $b \in \{0,1\}$ indicating whether the authentication request was accepted. It also outputs the updated server-side registration context rc_S, a credential identifier cid, a session identifier sid, and agreed contents agCon from the perspective of the server S.

5.4.2. WebAuthn as a PIA protocol

We now introduce the high level flow of execution of WebAuthn as a PIA protocol (see Figure 5.7) and its simplified syntax (see Figure 5.8), which closely follows the descriptions from [2, 39].

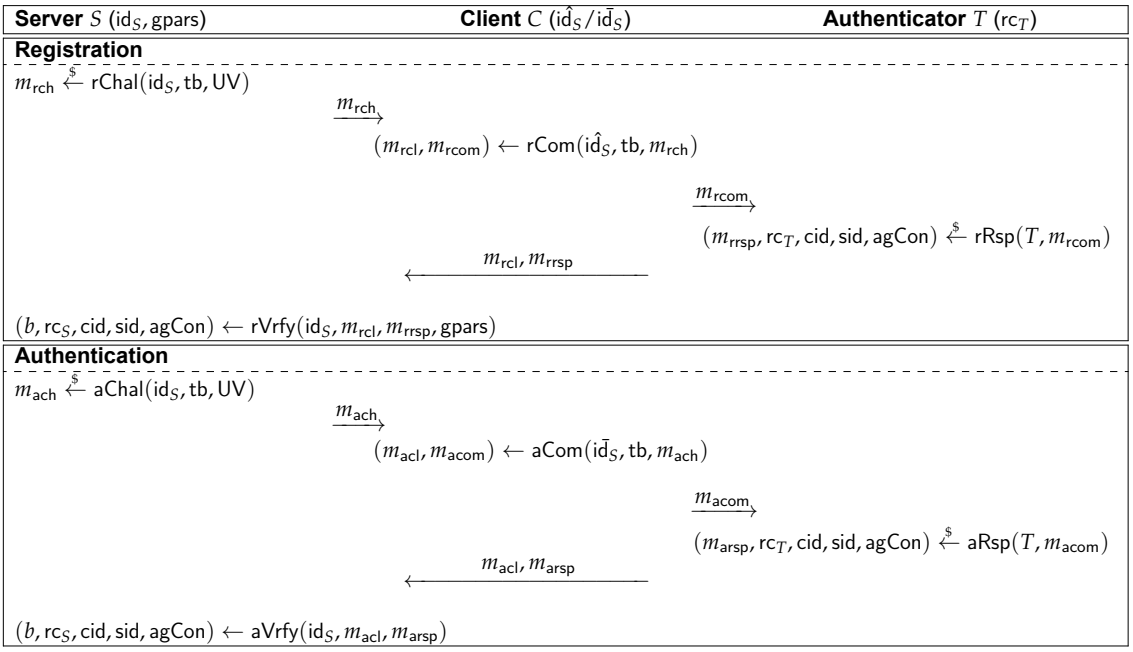


Figure 5.7: The WebAuthn protocol. For attestation type Basic, attestation material is generated by the group initialization algorithm $(\mathsf{gpars}, \mathsf{rc}) \xleftarrow{\$} \mathsf{GInit}$, which for our purposes can be thought of as a signature key generation algorithm where gpars contains the attestation verification key vk. The token's registration context rc_T is initialized as rc, which contains the attestation private key ak. The server inputs its identity id_S and attestation info gpars. For attestation types None and Self, both rc_T and gpars are empty. The full description of the algorithms in this diagram can be consulted in Figure 5.8.

On registration, the server runs rChal, which inputs the server identity id_S, token binding state tb and user verification condition UV (indicating whether the user should be verified with PIN or biometrics), and samples a new random challenge ch_S and user identifier uid_S. All these variables, except the token binding state, compose the challenge message m_{rch}, which is output and delivered to the client C. This challenge message is a very simplified representation of the data showcased in Figure 3.2 of Chapter 3, with its delivery to the client representing the invocation of navigator.credentials.create. C then runs rCom, which inputs the server domain $\hat{\mathsf{id}}_s$, the token binding state tb and the challenge message m_{rch}, and first checks if the received server id_S identity in m_{rch} matches $\hat{\mathsf{id}}_s$. If it does, then it combines the server challenge ch and the token binding state tb into a client message m_{rcl}, which is then hashed. The digest h (which represents the clientDataHash parameter), along with the server identity id_S, the user identifier uid and the user verification condition UV, are grouped into a command message m_{rcom}, which is output and sent to

the authenticator T (this represents the client calling the authenticatorMakeCredential command). The authenticator runs rRsp, which inputs m_{rcom} and generates a new assertion key pair (pk, sk), samples a new credential identifier cid and sets the signature counter n to zero. Next, it combines the received id_S, the signature counter n, the cid, the new public key pk and the received UV into a message m and, depending on the attestation type required by the relying party (Basic in Figure 5.7), produces a new attestation signature σ_{att} on m and h (the hash of m_{rcl}) using the authenticator's private attestation ak. The authenticator then sets, in its registration context rc_T, and for the server identity id_S, the received user identifier uid, the sampled cid, the assertion secret key sk and the signature counter n, and also sets the id_S, h, cid, n, pk, UV and attestation type used as the agreed content agCon (data that must be the same both from the perspective of the server and the token). Finally, it sets the hash of id_S, the cid and the signature counter as the session identifier sid, and returns the token response message m_{rrsp}, composed by m and (optionally) σ_{att} back to the client, which in turn forwards m_{rrsp} and its own m_{rcl} to the server. To conclude the registration, the server runs rVrfy, which inputs id_S, m_{rcl}, m_{rrsp} and public parameters gpars, and then checks if the information that came from the token and client is correct, including the server identity id_S, the token binding state tb_S, the sampled challenge ch_S, the user verification condition UV_S, the signature counter n (which must be zero) and, depending on the attestation type, the attestation signature σ_{att}. If everything is correct, it sets the agreed content agCon and the session identifier sid identically to the token, sets the user identifier uid_S, the token assertion public key pk and the public signature counter n in its registration context rc_S, for the specified cid, and finishes by returning successfully.

On authentication, the procedure is very similar. The server starts with aChal, which is identical to rChal except that it does not sample a new uid_S, which is therefore excluded from the output server message m_{ach}. The same is true of the aCom algorithm, which differs only in the output message m_{acom} that no longer contains uid. The token, upon receiving m_{acom}, runs aRsp, which first increments the signature counter n stored in its registration context. Next, the hashed server identity id_S, along with n, the user verification condition UV and the hashed m_{acl} client message h are signed using the assertion secret key sk generated in the previous registration run to produce a signature σ. Then, it sets the id_S, h, n and UV as the agreed content agCon and the hashed id_S, the cid (sampled and stored in the previous registration run), h and n as the session identifier sid. Finally, it sets the cid, the hashed id_S, n, UV, the assertion signature σ and the user identifier uid as the response message m_{arsp}, which is returned to the client, and then forwarded, along

rChal($\mathsf{id}_S, \mathsf{tb}, \mathsf{UV}$):
1: $\mathsf{ch}_S \xleftarrow{\$} \{0,1\}^{\geq\lambda}, \mathsf{tb}_S \leftarrow \mathsf{tb}$
2: $\mathsf{UV}_S \leftarrow \mathsf{UV}, \mathsf{uid}_S \xleftarrow{\$} \{0,1\}^{\leq 4\lambda}$
3: $m_{\mathsf{rch}} \leftarrow (\mathsf{id}_S, \mathsf{ch}_S, \mathsf{uid}_S, \mathsf{UV}_S)$
4: **return** m_{rch}

rCom($\hat{\mathsf{id}_S}, \mathsf{tb}, m_{\mathsf{rch}}$):
1: $(\mathsf{id}_S, \mathsf{ch}, \mathsf{uid}, \mathsf{UV}) \leftarrow m_{\mathsf{rch}}$
2: **if** $\mathsf{id}_S \neq \hat{\mathsf{id}_S}$ **then**
3: abort
4: $m_{\mathsf{rcl}} \leftarrow (\mathsf{ch}, \mathsf{tb})$
5: $h \leftarrow \mathsf{H}(m_{\mathsf{rcl}})$
6: $m_{\mathsf{rcom}} \leftarrow (\mathsf{id}_S, \mathsf{uid}, h, \mathsf{UV})$
7: **return** $(m_{\mathsf{rcl}}, m_{\mathsf{rcom}})$

rRsp(T, m_{rcom}):
1: $(\mathsf{id}_S, \mathsf{uid}, h, \mathsf{UV}) \leftarrow m_{\mathsf{rcom}}$
2: $(pk, sk) \xleftarrow{\$} \mathsf{KG}(), \mathsf{cid} \xleftarrow{\$} \{0,1\}^{\geq\lambda}, n \leftarrow 0$
3: $m \leftarrow (\mathsf{H}(\mathsf{id}_S), n, \mathsf{cid}, pk, \mathsf{UV})$
4: $\sigma_{\mathsf{att}} \xleftarrow{\$} \mathsf{Sign}(\mathsf{rc}_T.\mathsf{ak}, (m, h))$
5: $m_{\mathsf{rrsp}} \leftarrow (m, \sigma_{\mathsf{att}})$
6: $\mathsf{rc}_T[\mathsf{id}_S] \leftarrow (\mathsf{uid}, \mathsf{cid}, sk, n)$
7: $\mathsf{agCon} \leftarrow (\mathsf{id}_S, h, \mathsf{cid}, n, pk, \mathsf{UV}, \mathsf{basic})$
8: $\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}_S), \mathsf{cid}, n)$
9: **return** $(m_{\mathsf{rrsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon})$

rVrfy($\mathsf{id}_S, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}}, \mathsf{gpars}$):
1: $(\mathsf{ch}, \mathsf{tb}) \leftarrow m_{\mathsf{rcl}}, (m, \sigma_{\mathsf{att}}) \leftarrow m_{\mathsf{rrsp}}$
2: $(h, n, \mathsf{cid}, pk, \mathsf{UV}) \leftarrow m$
3: **if** $\mathsf{ch}_S \neq \mathsf{ch} \vee \mathsf{tb}_S \neq \mathsf{tb} \vee h \neq \mathsf{H}(\mathsf{id}_S) \vee \mathsf{UV}_S \neq \mathsf{UV} \vee n \neq 0 \vee \mathsf{Ver}(\mathsf{gpars.vk}, (m, \mathsf{H}(\mathsf{id}_S)), \sigma_{\mathsf{att}}) = 0$ **then**
4: **return** $(0, \mathsf{rc}_S, \perp, \perp, \perp)$
5: $\mathsf{agCon} \leftarrow (\mathsf{id}_S, \mathsf{H}(m_{\mathsf{rcl}}), \mathsf{cid}, n, pk, \mathsf{UV}, \mathsf{basic})$
6: $\mathsf{rc}_S[\mathsf{cid}] \leftarrow (\mathsf{uid}_S, pk, n)$
7: $\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}_S), \mathsf{cid}, n)$
8: **return** $(1, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon})$

aChal($\mathsf{id}_S, \mathsf{tb}, \mathsf{UV}$):
1: $\mathsf{ch}_S \xleftarrow{\$} \{0,1\}^{\geq\lambda}, \mathsf{tb}_S \leftarrow \mathsf{tb}$
2: $\mathsf{UV}_S \leftarrow \mathsf{UV}$
3: $m_{\mathsf{ach}} \leftarrow (\mathsf{id}_S, \mathsf{ch}_S, \mathsf{UV}_S)$
4: **return** m_{ach}

aCom($\bar{\mathsf{id}_S}, \mathsf{tb}, m_{\mathsf{ach}}$):
1: $(\mathsf{id}_S, \mathsf{ch}, \mathsf{UV}) \leftarrow m_{\mathsf{ach}}$
2: **if** $\mathsf{id}_S \neq \bar{\mathsf{id}_S}$ **then**
3: abort
4: $m_{\mathsf{acl}} \leftarrow (\mathsf{ch}, \mathsf{tb})$
5: $h \leftarrow \mathsf{H}(m_{\mathsf{acl}})$
6: $m_{\mathsf{acom}} \leftarrow (\mathsf{id}_S, h, \mathsf{UV})$
7: **return** $(m_{\mathsf{acl}}, m_{\mathsf{acom}})$

aRsp(T, m_{acom}):
1: $(\mathsf{id}_S, h, \mathsf{UV}) \leftarrow m_{\mathsf{acom}}$
2: $\mathsf{rc}_T[\mathsf{id}_S].n \leftarrow \mathsf{rc}_T[\mathsf{id}_S].n + 1$
3: $ad \leftarrow (\mathsf{H}(\mathsf{id}_S), \mathsf{rc}_T[\mathsf{id}_S].n, \mathsf{UV})$
4: $\sigma \leftarrow \mathsf{Sign}(\mathsf{rc}_T[\mathsf{id}_S].sk, (ad, h))$
5: $m_{\mathsf{arsp}} \leftarrow (\mathsf{rc}_T[\mathsf{id}_S].\mathsf{cid}, ad, \sigma, \mathsf{rc}_T[\mathsf{id}_S].\mathsf{uid})$
6: $\mathsf{agCon} \leftarrow (\mathsf{id}_S, h, \mathsf{rc}_T[\mathsf{id}_S].n, \mathsf{UV})$
7: $\mathsf{sid} \leftarrow (\mathsf{H}(\mathsf{id}_S), \mathsf{rc}_T[\mathsf{id}_S].\mathsf{cid}, h, \mathsf{rc}_T[\mathsf{id}_S].n)$
8: **return** $(m_{\mathsf{arsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon})$

aVrfy($\mathsf{id}_S, m_{\mathsf{acl}}, m_{\mathsf{arsp}}$):
1: $(\mathsf{ch}, \mathsf{tb}) \leftarrow m_{\mathsf{acl}}, (\mathsf{cid}, ad, \sigma, \mathsf{uid}) \leftarrow m_{\mathsf{arsp}}$
2: $(h, n, \mathsf{UV}) \leftarrow ad$
3: **if** $\mathsf{ch}_S \neq \mathsf{ch} \vee \mathsf{tb}_S \neq \mathsf{tb} \vee h \neq \mathsf{H}(\mathsf{id}_S) \vee \mathsf{UV}_S \neq \mathsf{UV} \vee n \leq \mathsf{rc}_S[\mathsf{cid}].n \vee \mathsf{Ver}(\mathsf{rc}_S[\mathsf{cid}].pk, (ad, \mathsf{H}(m_{\mathsf{acl}})), \sigma) = 0$ **then**
4: **return** $(0, \mathsf{rc}_S, \perp, \perp, \perp)$
5: $\mathsf{agCon} \leftarrow (\mathsf{id}_S, \mathsf{H}(m_{\mathsf{acl}}), n, \mathsf{UV})$
6: $\mathsf{rc}_S[\mathsf{cid}].n \leftarrow n$
7: $\mathsf{sid} \leftarrow (h, \mathsf{cid}, \mathsf{H}(m_{\mathsf{acl}}), n)$
8: **return** $(1, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon})$

Figure 5.8: The WebAuthn protocol functions.

with the client message m_{acl}, to the server. The aVrfy algorithm is also similar to rVrfy. It inputs m_{acl} and m_{arsp} and verifies if the information received is correct, with the major difference being that it now verifies the assertion signature σ using the previously stored assertion public key pk, and also checks if the signature counter received is not greater than the counter stored in the previous registration/authentication session. If everything is correct, it sets the agCon and sid identically to the token, updates the signature counter n, and returns successfully.

5.4.3. PIA Authentication Model and WebAuthn security

As with the syntax, our model is very similar to the ePIA model defined in [2] and the ePIAA model defined in [39], with some important changes that we discuss below.

Authentication Model

As is shown in Figure 5.9, the passwordless authentication (pla-auth) security of a PIA protocol PIA is defined with a security experiment $\mathsf{Expt}^{\mathsf{pla\text{-}auth}}_{\mathsf{PIA}}$, executed between a challenger and an adversary $\mathcal{A}$.

Trust model For attestation types None and Self, as in prior work, we assume that the PIA adversary $\mathcal{A}$ is passive during registration, since the server has no prior knowledge of attestation material stored in the specific token of interest. Indeed, if the adversary can be active during registration, it is impossible to prove WebAuthn secure for these modes, as noted in [39]. For attestation type Basic, and as in [39], we can prove PIA security even with active registration, because the guarantee provided to the server is merely that the credential has been created by some token in the target group. However, the server does not know which token in the group produced the credential and, in fact, we know from our discussion of rogue key attacks that the server does not know if this token is owned by a given user or by an adversary. We allow groups to be dynamically generated by the adversary. For attestation type Basic, this corresponds to creating a new batch. The adversary is allowed to create as many groups as it wants, and to assign tokens to them at will (note that in modes None and Self there is only one group). Then, the adversary is allowed to corrupt the attestation material of all tokens except the tokens in the target batch of interest. Fine-grained credential corruption (not including the attestation material) is still allowed within the batch. This is a strengthening of the model in [39] and does not have a significant impact on the WebAuthn security proof, while simultaneously simplifying the description of the model.

Session oracles and registration contexts To model concurrent or sequential PIA protocol instances (i.e., sessions) of a server S (associated with id_S) and sequential PIA sessions of a token T, we use $\pi^i_{r,S}$ and $\pi^j_{r,T}$ to denote their i-th and j-th registration instances, and $\pi^i_{a,S}$ and $\pi^j_{a,T}$ to denote their i-th and j-th authentication instances. The execution status of a session oracle $\pi^k_{ph,P}$ ($ph \in \{r,a\}$, $P \in \{S,T\}$), denoted by $\pi^k_{ph,P}.\mathsf{st}_{\mathsf{exe}}$, is either of $\{\perp, \mathsf{running}, \mathsf{accepted}\}$, where $\perp$ means the session oracle is not yet initialized, in which case we simply write $\pi^k_{ph,P} = \perp$. Session identifiers sid and agreed contents agCon are specific to a session. Registration contexts rc_S, rc_T are global to a server or token, respectively, and we abuse notation to allow them to be indexed by the (unique) identity of a token or server, respectively, as $\mathsf{rc}_S[T]$ or $\mathsf{rc}_T[S]$. This is well defined as we impose a single registration run between a given pair (S,T).

Attestation types To account for attestation, we deviate from [39] and define a group initialization algorithm $(\mathsf{gpars}, \mathsf{rc}) \xleftarrow{\$} \mathsf{GInit}$ that creates a new group, which is cryptographically defined by some public group parameters gpars and a private registration context rc. This public gpars is taken as input by the servers and the private rc is taken as input by

tokens in the same group as their initial registration context. For attestation types None and Self, such attestation material gpars and rc are empty. For the Basic mode, GInit uses a key generation algorithm to output an attestation key pair, and then assigns the private key to rc and the public key (with potentially other public parameters) to gpars.

We model attestation in this way for simplicity, as we only aim to capture attestation types None, Self, and Basic. These are the most common for USB authenticators, which are the main focus of this work. This deviates from the approach in [39], where the model captures all types of attestation, including attCA and anonCA.

Session partnership We say that a server registration session $\pi^i_{r,S}$ partners with a token registration session $\pi^j_{r,T}$ if and only if $\pi^i_{r,S}.\mathsf{sid} = \pi^j_{r,T}.\mathsf{sid}$. We define partnership for authentication sessions as $\pi^i_{a,S}.\mathsf{sid} = \pi^j_{a,T}.\mathsf{sid}$, and, furthermore, require that they can be associated to unique partnered registration sessions $\pi^{i'}_{r,S}.\mathsf{sid} = \pi^{j'}_{r,T}.\mathsf{sid}$ such that $\pi^i_{a,S}.\mathsf{cid} = \pi^{i'}_{r,S}.\mathsf{cid}$. In other words, authentication partnership guarantees that there is unique registration partnership between the same server and token that establish the (unique) credential identifier that the server recovers at the end of the authentication run. Many authentication runs can, of course, be bound to the same registration sessions.

Looking ahead, with the above session partnership, our model is stronger than the models in [2, 39], in the sense that our model further guarantees that authentication binds the user to a unique registration that took place before between the same server/token pair, identified by a credential identifier cid. This also allows us to capture the typical scenario where cid is used by the server to identify the correct application-specific identifier. Note that our session partnership definition is, in the above sense, the same as [1].

Advantage measure For a PIA protocol PIA, its advantage (with respect to the security experiment $\mathsf{Expt}^{\mathsf{pla\text{-}auth}}_{\mathsf{PIA}}$ shown in Figure 5.9) is defined for any adversary $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{pla\text{-}auth}}_{\mathsf{PIA}}(\mathcal{A}) = \Pr[\mathsf{Expt}^{\mathsf{pla\text{-}auth}}_{\mathsf{PIA}}(\mathcal{A}) = 1]$$

Differences to prior work

As discussed above, our model introduces two differences to [2, 39], in order to capture a guarantee provided by FIDO2, which was captured by the original model in [1] but lost in the more recent works [2, 39]: (i) a separation between registration and authentication sessions for modeling the partnership between them, and (ii) the explicit handling of

Partnerships(S, i, cid):
1: **if** $\exists i'$ such that $\pi^{i'}_{r,S}.\text{cid} = \text{cid}$ **then**
2: Retrieve $(S, i', \text{gid}, \text{cid})$ from $\mathcal{L}_{\text{reg}}$
3: **if** $\exists T, j'$ such that $\pi^{i'}_{r,S}.\text{sid} = \pi^{j'}_{r,T}.\text{sid}$ **then**
4: **if** $\exists j$ such that $\pi^{i}_{a,S}.\text{sid} = \pi^{j}_{a,T}.\text{sid}$ **then**
5: **return** $(\text{gid}, i', T, j', j)$
6: **else return** $(\text{gid}, i', T, j', \perp)$
7: **else return** $(\text{gid}, i', \perp, \perp, \perp)$
8: **return** $(\perp, \perp, \perp, \perp, \perp)$

Win-auth (S, i, cid):
1: **if** $\exists (S_1, i_1, ph_1) \neq (S_2, i_2, ph_2)$ s.t. $\pi^{i_1}_{ph_1,S_1}.\text{sid} = \pi^{i_2}_{ph_2,S_2}.\text{sid} \neq \perp$ **then return** 1
2: **if** $\exists (T_1, j_1, ph_1) \neq (T_2, j_2, ph_2)$ s.t. $\pi^{j_1}_{ph_1,T_1}.\text{sid} = \pi^{j_2}_{ph_2,T_2}.\text{sid} \neq \perp$ **then return** 1
3: **if** $\exists (S_1, i_1) \neq (S_2, i_2)$ s.t. $\pi^{i_1}_{r,S_1}.\text{cid} = \pi^{i_2}_{r,S_2}.\text{cid} \neq \perp$ **then**
4: **return** 1
5: **if** $\exists (S', i', ph'), (T', j', ph')$ s.t. $\pi^{i'}_{ph',S'}.\text{sid} = \pi^{j'}_{ph',T'}.\text{sid} \neq \perp$ **and** $(S', T') \notin \mathcal{L}_{\text{corr}}$ **and** $T'.\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $\pi^{i'}_{ph',S'}.\text{agCon} \neq \pi^{j'}_{ph',T'}.\text{agCon}$ **then return** 1
6: $(\text{gid}, _, T, _, j) \leftarrow$ Partnerships(S, i, cid)
7: // Attestation broken: wrong group or no registration partner
8: **if** $\text{gid} = \perp$ **or** $(\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $(T = \perp$ **or** $T.\text{gid} \neq \text{gid}))$ **then**
9: **return** 1
10: **else**
11: // Authentication broken: no authentication partner
12: **if** $\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **then**
13: **if** $(S, T) \notin \mathcal{L}_{\text{corr}}$ **and** $j = \perp$ **then return** 1
14: **return** 0

Expt$^{\text{pla-auth}}_{\text{PIA}}(\mathcal{A})$:
1: $\mathcal{L}_{\text{reg}} \leftarrow \emptyset$; $\mathcal{L}_{\text{corr}} \leftarrow \emptyset$; $\mathcal{L}_{\text{corrG}} \leftarrow \emptyset$; $G \leftarrow \emptyset$; $\text{gid} \leftarrow 0$
2: win-auth $\leftarrow 0$
3: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
4: **return** win-auth

Reg $((S, i), (T, j), \text{tb}, \text{UV}, \text{gid})$:
1: // This oracle replaces the **rChall**, **rResp** and **rCompl** oracles in the passive registration mode
2: **if** $T.\text{gid} = \perp$ **or** $\pi^{i}_{r,S} \neq \perp$ **or** $\pi^{j}_{r,T} \neq \perp$ **or** $\text{rc}_T[S] \neq \perp$ **or** $G[\text{gid}] = \perp$ **then return** $\perp$
3: $m_{\text{rch}} \xleftarrow{\$} \text{rChal}(\pi^{i}_{r,S}, \text{tb}, \text{UV})$
4: $(m_{\text{rcom}}, m_{\text{rcl}}) \leftarrow \text{rCom}(\text{id}_S, m_{\text{rch}}, \text{tb})$
5: $(m_{\text{rrsp}}, \text{rc}_T, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{rRsp}(\pi^{j}_{r,T}, m_{\text{rcom}})$
6: $(d, \text{rc}_S, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{rVrfy}(\pi^{i}_{r,S}, m_{\text{rcl}}, m_{\text{rrsp}}, G[\text{gid}].\text{gpars})$
7: $\mathcal{L}_{\text{reg}} \leftarrow \mathcal{L}_{\text{reg}} \cup \{(S, i, \text{gid}, \text{cid})\}$
8: **return** $(d, m_{\text{rch}}, m_{\text{rcl}}, m_{\text{rcom}}, m_{\text{rrsp}})$

NewGroup $()$:
1: $(\text{gpars}, \text{rc}) \xleftarrow{\$} \text{GInit}$
2: $G[\text{gid}] \leftarrow (\text{gpars}, \text{rc})$
3: $\text{gid} \leftarrow \text{gid} + 1$

NewToken (gid, T):
1: **if** $G[\text{gid}] = \perp$ **then return** $\perp$
2: **if** $T.\text{gid} \neq \perp$ **then return** $\perp$
3: $T.\text{gid} \leftarrow \text{gid}$
4: $\text{rc}_T \leftarrow G[\text{gid}].\text{rc}$

rChall $((S, i), \text{tb}, \text{UV})$:
1: **if** $\pi^{i}_{r,S} \neq \perp$ **then return** $\perp$
2: $m_{\text{rch}} \xleftarrow{\$} \text{rChal}(\pi^{i}_{r,S}, \text{tb}, \text{UV})$
3: **return** m_{rch}

rCompl $((S, i), m_{\text{rcl}}, m_{\text{rrsp}}, \text{gid})$:
1: **if** $\pi^{i}_{r,S} = \perp$ **or** $\pi^{i}_{r,S}.\text{st}_{\text{exe}} \neq \text{running}$ **or** $G[\text{gid}] = \perp$ **then return** $\perp$
2: $(d, \text{rc}_S, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{rVrfy}(\pi^{i}_{r,S}, m_{\text{rcl}}, m_{\text{rrsp}}, G[\text{gid}].\text{gpars})$
3: **if** $d = 1$ **then** $\mathcal{L}_{\text{reg}} \leftarrow \mathcal{L}_{\text{reg}} \cup \{(S, i, \text{gid}, \text{cid})\}$
4: **return** d

Corrupt (S, T):
1: **if** $\text{rc}_T[S] = \perp$ **then return** $\perp$
2: $\mathcal{L}_{\text{corr}} \leftarrow \mathcal{L}_{\text{corr}} \cup \{(S, T)\}$
3: **return** $\text{rc}_T[S]$

rResp $((T, j), m_{\text{rcom}})$:
1: **if** $\pi^{j}_{r,T} \neq \perp$ **or** $T.\text{gid} = \perp$ **then return** $\perp$
2: $(m_{\text{rrsp}}, \text{rc}_T, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{rRsp}(\pi^{j}_{r,T}, m_{\text{rcom}})$
3: **return** m_{rrsp}

aChall $((S, i), \text{tb}, \text{UV})$:
1: **if** $\pi^{i}_{a,S} \neq \perp$ **then return** $\perp$
2: $m_{\text{ach}} \xleftarrow{\$} \text{aChal}(\pi^{i}_{a,S}, \text{tb}, \text{UV})$
3: **return** m_{ach}

aCompl $((S, i), m_{\text{acl}}, m_{\text{arsp}})$:
1: **if** $\pi^{i}_{a,S} = \perp$ **or** $\pi^{i}_{a,S}.\text{st}_{\text{exe}} \neq \text{running}$ **then return** $\perp$
2: $(d, \text{rc}_S, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{aVrfy}(\pi^{i}_{a,S}, m_{\text{acl}}, m_{\text{arsp}})$
3: **if** $d = 1$ **and** win-auth $= 0$ **then**
4: win-auth $\leftarrow$ Win-auth(S, i, cid)
5: **return** d

CorruptGroup (gid):
1: **if** $G[\text{gid}] = \perp$ **then return** $\perp$
2: $\mathcal{L}_{\text{corrG}} \leftarrow \mathcal{L}_{\text{corrG}} \cup \{\text{gid}\}$
3: **return** $G[\text{gid}]$

aResp $((T, j), m_{\text{acom}})$:
1: **if** $\pi^{j}_{a,T} \neq \perp$ **or** $T.\text{gid} = \perp$ **then return** $\perp$
2: $(m_{\text{arsp}}, \text{rc}_T, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{aRsp}(\pi^{j}_{a,T}, m_{\text{acom}})$
3: **return** m_{arsp}

Figure 5.9: Security experiment, winning conditions, and oracle definitions for the PIA passwordless authentication security experiment. Code in blue represents the added modifications with respect to [2, 39]. We let $\mathcal{O}$ denote the set of all of the security experiment oracles that are available to $\mathcal{A}$. The winning condition procedure Win-auth is called in the aCompl oracle whenever the server authentication session accepts.

credential identifiers and group identifiers.[9] As mentioned before, we simplify our model to capture the simplest attestation types, as they are the most commonly used modes in practical USB-based tokens today, and handling all possible attestation types is not our focus. Instead, we tailor our definition to the settings where there is no attestation (or just simple Self attestation) or where Basic attestation is used. For this, we let the server registration verification algorithm input some group parameters, and impose that the server can only accept registrations from tokens whose attestation material is consistent with a given set of group parameters.

Authentication security of WebAuthn

We do not restate the passwordless authentication security of WebAuthn in this model, as it has been proved in [1]. Indeed, the security proofs given in [1] suffice to show that WebAuthn satisfies the PIA authentication notion we consider here: the proof begins by excluding collisions in credential identifiers, and then relies on the uniqueness of credential identifiers to pinpoint a unique registration session that established the public key under which the authentication took place. For attestation types None, Self, in which no hardcoded attestation material is used, the proof holds when the adversary is restricted only to passive registration attacks and can create only one group for which the attestation parameters are empty. For Basic attestation the adversary can create an arbitrary number of groups. Rather than fixing a group *a priori*, we just allow the adversary to choose the target group adaptively, but this is of no consequence to the proof: as stated in [1], the only way the attacker could succeed in registering a key that is outside of the group fixed by a server verification run would be to either corrupt the group or forge an attestation signature.

5.5. PIA+mPACA composed security analysis

Our security results from Section 5.3 show that CTAP 2.1+, as an mPACA protocol, offers an additional guarantee that the client can only accept a response from the specific token with which it previously established an authenticated channel. Therefore, the natural question that follows is: how does replacing CTAP 2.1 with CTAP 2.1+ impact the security analysis of FIDO2 as a whole? How does the composed PIA+PACA model change when considering mPACA protocols instead?

[9] For simplicity, we also keep the algorithm negotiation parts of PIA implicit in the experiment code.

In this Section, we introduce our PIA+mPACA model, highlighting our changes with respect to the original PIA+PACA models from [1, 2], especially regarding our stronger trust model that imposes fewer restrictions on active attacks from the adversary. Then, we prove that the composed PIA+mPACA protocol achieves *user authentication* security in our model, and end with a reflection on how this composed model offers further guarantees beyond those from current FIDO2.

5.5.1. PIA+mPACA authentication model

The main highlight of our composed PIA+mPACA protocol analysis is that we can now strengthen the trust model to allow active attacks on both registration and authentication runs, independently of attestation types. Our composed model for the PIA+mPACA authentication security is based on the PIA authentication security model presented in Section 5.4 and our proposed mPACA model from Section 5.3. Following the approach from [1, 2], we consider a security experiment $\mathsf{Expt}^{\mathsf{ua}}_{\mathsf{PIA+mPACA}}$, presented in Figures 5.10 and 5.11, which is executed between a challenger and an adversary $\mathcal{A}$ against the ua (*user authentication*) security of $\mathsf{PIA+mPACA}$.

Trust model We follow the trust model in [1, 2] and assume the communication channel between servers and clients is authenticated in the sense that the client is assured as to the identity of the server. This captures the guarantees of a secure connection, established for example with TLS. We maintain the trust model from mPACA unchanged, which means any client session can only complete Bind passively. However, we now allow active composed model adversaries during registration runs, regardless of the attestation type, whereas in the PIA and previous composed models we could only deal with active registration when using Basic attestation. Indeed, the only difference between attestation types $\mathsf{None/Self}$ with respect to mode Basic is that for the former we do not allow mPACA clients to be compromised during registration, whereas for the latter we can allow this. We note that this is a stronger model than that adopted in [2], in that we can deal with active attacks during registration, because we have upgraded PACA to mPACA to provide a bi-directional authenticated channel.

Session oracles and partnership We maintain the session oracles defined for PIA and mPACA, as well as all protocol variables, internal states and partnership definitions, but follow the approach from [2] by using $\tilde{\pi}$ and π to refer to PIA and mPACA sessions respectively.

Experiment oracles The ua experiment gives the adversary access to all of the SUF-t experiment's oracles except for Auth-C, Validate-T, Auth-T and Validate-C. Furthermore, the adversary also has access to all unchanged oracles from the pla-auth experiment, except for rChall, rResp, rCompl, aChall, aResp and aCompl, which are redefined for this experiment in Figure 5.10, and except for Reg, which is absent due to the assumption that $\mathcal{A}$ can always actively interfere with any registration session.

The rChall oracle now additionally takes a client session as input, which prepares the message from the server session $\tilde{\pi}^i_{r,S}$ by calling rCom to produce a client message m_{rcl} and a command m_{rcom}. This command is then authenticated using the mPACA oracle Auth-C, producing a tag t_{cl}. In addition to m_{rch}, both messages output by rCom and t_{cl} are given to $\mathcal{A}$. The changes to aChall are analogous.

The rResp oracle now additionally inputs an mPACA token session, a tag t_{cl} and a user decision bit d. It starts by querying the Validate-T oracle on the m_{rcom} message and tag t_{cl}, aborting if the status is not accepted. Then, it authenticates the PIA token response m_{rrsp} via the mPACA oracle Auth-T, producing a tag t_{tk}, which is also given to $\mathcal{A}$. The changes to aResp are analogous.

Finally, the rCompl oracle additionally receives a tag t_{tk}, fetches the client session associated with the current server session, and queries Validate-C on the token response m_{rrsp} and t_{tk}, failing if the status is not accepted. The changes to aCompl are analogous, with the addition that it now calls Win-ua (described in Figure 5.11) to verify the winning conditions whenever a server session $\tilde{\pi}^i_{a,S}$ accepts.

We also define two lists $\mathcal{L}_{\text{pla-paca-S}}$ and $\mathcal{L}_{\text{pla-paca-T}}$ to, respectively, link every PIA server session with its associated mPACA client session in rChall and aChall, and every PIA token session with its associated mPACA token session in rResp and aResp.

Winning conditions and advantage measure The winning conditions specified in Win-ua intuitively guarantee that PIA and mPACA sessions can be uniquely identified by their derived session identifiers and, furthermore, as in the PIA experiments, that registration sessions obtain unique credential identifiers. Then, the adversary wins if it breaks the PIA authentication security at any point, regardless of whether it is using compromised mPACA clients or not. Furthermore, when using uncompromised mPACA clients, the adversary also wins if it can convince the PIA server to pair with a registration or authentication session that is hosted by some other token than the one bound to the unique

rChall (S, i, C, k, tb, UV):
1: **if** $\tilde{\pi}^i_{r,S} \neq \perp$ **then return** $\perp$
2: $m_{\mathsf{rch}} \xleftarrow{\$} \mathsf{rChal}(\tilde{\pi}^i_{r,S}, \mathsf{tb}, \mathsf{UV})$
3: $(m_{\mathsf{rcl}}, m_{\mathsf{rcom}}) \xleftarrow{\$} \mathsf{rCom}(\mathsf{id}_S, m_{\mathsf{rch}}, \mathsf{tb})$
4: $\mathsf{resp} \xleftarrow{\$} \mathsf{Auth\text{-}C}(C, k, m_{\mathsf{rcom}})$
5: **if** $\mathsf{resp} = \perp$ **then return** $\perp$
6: $(m_{\mathsf{rcom}}, t_{\mathsf{cl}}) \leftarrow \mathsf{resp}$
7: $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}} \leftarrow \mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}} \cup \{(\mathsf{reg}, S, i, C, k)\}$
8: **return** $(m_{\mathsf{rch}}, m_{\mathsf{rcl}}, m_{\mathsf{rcom}}, t_{\mathsf{cl}})$

aChall (S, i, C, k, tb, UV):
1: **if** $\tilde{\pi}^i_{a,S} \neq \perp$ **then return** $\perp$
2: $m_{\mathsf{ach}} \xleftarrow{\$} \mathsf{aChal}(\tilde{\pi}^i_{a,S}, \mathsf{tb}, \mathsf{UV})$
3: $(m_{\mathsf{acl}}, m_{\mathsf{acom}}) \xleftarrow{\$} \mathsf{aCom}(\mathsf{id}_S, m_{\mathsf{ach}}, \mathsf{tb})$
4: $\mathsf{resp} \xleftarrow{\$} \mathsf{Auth\text{-}C}(C, k, m_{\mathsf{acom}})$
5: **if** $\mathsf{resp} = \perp$ **then return** $\perp$
6: $(m_{\mathsf{acom}}, t_{\mathsf{cl}}) \leftarrow \mathsf{resp}$
7: $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}} \leftarrow \mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}} \cup \{(\mathsf{auth}, S, i, C, k)\}$
8: **return** $(m_{\mathsf{ach}}, m_{\mathsf{acl}}, m_{\mathsf{acom}}, t_{\mathsf{cl}})$

rResp (T, j, j', m_{rcom}, t_{cl}, d):
1: **if** $\tilde{\pi}^j_{r,T} \neq \perp$ **or** $T.\mathsf{gid} = \perp$ **then return** $\perp$
2: $\mathsf{status} \xleftarrow{\$} \mathsf{Validate\text{-}T}(T, j', m_{\mathsf{rcom}}, t_{\mathsf{cl}}, d)$
3: **if** $\mathsf{status} \neq \mathsf{accepted}$ **then return** $\perp$
4: $(m_{\mathsf{rrsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{rRsp}(\tilde{\pi}^j_{r,T}, m_{\mathsf{rcom}})$
5: $(m_{\mathsf{rrsp}}, t_{\mathsf{tk}}) \xleftarrow{\$} \mathsf{Auth\text{-}T}(T, j', m_{\mathsf{rrsp}})$
6: $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}T}} \leftarrow \mathcal{L}_{\mathsf{pla\text{-}paca\text{-}T}} \cup \{(\mathsf{reg}, T, j, j')\}$
7: **return** $(m_{\mathsf{rrsp}}, t_{\mathsf{tk}})$

aResp (T, j, j', m_{acom}, t_{cl}, d):
1: **if** $\tilde{\pi}^j_{a,T} \neq \perp$ **or** $T.\mathsf{gid} = \perp$ **then return** $\perp$
2: $\mathsf{status} \xleftarrow{\$} \mathsf{Validate\text{-}T}(T, j', m_{\mathsf{acom}}, t_{\mathsf{cl}}, d)$
3: **if** $\mathsf{status} \neq \mathsf{accepted}$ **then return** $\perp$
4: $(m_{\mathsf{arsp}}, \mathsf{rc}_T, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$} \mathsf{aRsp}(\tilde{\pi}^j_{a,T}, m_{\mathsf{acom}})$
5: $(m_{\mathsf{arsp}}, t_{\mathsf{tk}}) \xleftarrow{\$} \mathsf{Auth\text{-}T}(T, j', m_{\mathsf{arsp}})$
6: $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}T}} \leftarrow \mathcal{L}_{\mathsf{pla\text{-}paca\text{-}T}} \cup \{(\mathsf{auth}, T, j, j')\}$
7: **return** $(m_{\mathsf{arsp}}, t_{\mathsf{tk}})$

rCompl (S, i, m_{rcl}, m_{rrsp}, gid, t_{tk}):
1: **if** $\tilde{\pi}^i_{r,S} = \perp$ **or** $\tilde{\pi}^i_{r,S}.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{running}$ **or** $G[\mathsf{gid}] = \perp$ **then return** $\perp$
2: Retrieve $(\mathsf{reg}, S, i, C, k)$ from $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}}$
3: $\mathsf{status} \leftarrow \mathsf{Validate\text{-}C}(C, k, m_{\mathsf{rrsp}}, t_{\mathsf{tk}})$
4: **if** $\mathsf{status} \neq \mathsf{accepted}$ **then return** $\perp$
5: $(d, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$}$
$\mathsf{rVrfy}(\tilde{\pi}^i_{r,S}, m_{\mathsf{rcl}}, m_{\mathsf{rrsp}}, G[\mathsf{gid}].\mathsf{gpars})$
6: **if** $d = 1$ **then**
7: $\quad \mathcal{L}_{\mathsf{reg}} \leftarrow \mathcal{L}_{\mathsf{reg}} \cup \{(S, i, \mathsf{gid}, \mathsf{cid})\}$
8: **return** d

aCompl (S, i, m_{acl}, m_{arsp}, t_{tk}):
1: **if** $\tilde{\pi}^i_{a,S} = \perp$ **or** $\tilde{\pi}^i_{a,S}.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{running}$ **then return** $\perp$
2: Retrieve $(\mathsf{auth}, S, i, C, k)$ from $\mathcal{L}_{\mathsf{pla\text{-}paca\text{-}S}}$
3: $\mathsf{status} \leftarrow \mathsf{Validate\text{-}C}(C, k, m_{\mathsf{arsp}}, t_{\mathsf{tk}})$
4: **if** $\mathsf{status} \neq \mathsf{accepted}$ **then return** $\perp$
5: $(d, \mathsf{rc}_S, \mathsf{cid}, \mathsf{sid}, \mathsf{agCon}) \xleftarrow{\$}$
$\mathsf{aVrfy}(\tilde{\pi}^i_{a,S}, m_{\mathsf{acl}}, m_{\mathsf{arsp}})$
6: **if** $d = 1$ **and** $\mathsf{win\text{-}ua} = 0$ **then**
7: $\quad \mathsf{win\text{-}ua} \leftarrow \mathsf{Win\text{-}ua}(S, i, \mathsf{cid})$
8: **return** d

Figure 5.10: Oracles of the ua security experiment for the composed model. Code in blue highlights the differences to the PIA oracles shown in 5.4. Differences that are specific from mPACA are colored in red.

client that it communicates with. For a composed PIA+mPACA protocol PIA+mPACA, its ua advantage is defined for any adversary $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{ua}}_{\mathsf{PIA+mPACA}}(\mathcal{A}) = \Pr[\mathsf{Expt}^{\mathsf{ua}}_{\mathsf{PIA+mPACA}}(\mathcal{A}) = 1]$$

Differences to prior work We adopt the style of presentation of [2], but our composed model expresses the winning condition in a way that is closer to the one defined in [1]: we express the adversary's advantage purely as a function of its probability of breaking a server-side authentication guarantee and we do not include PACA-specific command forgery checks[10]. We can do this because we establish a stronger result based on mPACA. A winning condition closer to the one in [2] needs to be considered to capture the composed security of the current version of FIDO2, as we discuss next. Furthermore, our composed model inherits from our mPACA and PIA definitions the strengthenings we introduced for each of these primitives, namely the ability to dynamically choose attestation

[10]The authors in [1] also consider user-side guarantees, which are also provided by FIDO2, but we do not consider them here and leave clarifying and expanding the study of these guarantees as future work.

$\text{Expt}^{\text{ua}}_{\text{PIA+mPACA}}(\mathcal{A})$:
1: $\mathcal{L}_{\text{reg}} \leftarrow \emptyset$; $\mathcal{L}_{\text{corr}} \leftarrow \emptyset$; $\mathcal{L}_{\text{corrG}} \leftarrow \emptyset$; $G \leftarrow \emptyset$; gid $\leftarrow 0$ // From PIA
2: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$ // From mPACA
3: $\mathcal{L}_{\text{pla-paca-S}}, \mathcal{L}_{\text{pla-paca-T}} \leftarrow \emptyset$ // For grouping PIA and mPACA sessions
4: win-ua $\leftarrow 0$
5: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
6: **return** win-ua

Win-ua (S, i, cid):
1: // If there exists any collision between server or token PIA sessions, $\mathcal{A}$ wins
2: **if** $\exists (S_1, i_1, ph_1) \neq (S_2, i_2, ph_2)$ s.t. $\tilde{\pi}^{i_1}_{ph_1,S_1}.\text{sid} = \tilde{\pi}^{i_2}_{ph_2,S_2}.\text{sid} \neq \perp$ **then return** 1
3: **if** $\exists (T_1, j_1, ph_1) \neq (T_2, j_2, ph_2)$ s.t. $\tilde{\pi}^{j_1}_{ph_1,T_1}.\text{sid} = \tilde{\pi}^{j_2}_{ph_2,T_2}.\text{sid} \neq \perp$ **then return** 1
4:
5: // If there exists any collision between client or token mPACA sessions, $\mathcal{A}$ wins
6: **if** $\exists (C_1, k_1), (C_2, k_2)$ *s.t.* $(C_1, k_1) \neq (C_2, k_2)$ **and** $\pi^{k_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{k_2}_{C_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{k_1}_{C_1}.\text{sid} = \pi^{k_2}_{C_2}.\text{sid}$ **then return** 1
7: **if** $\exists (T_1, j'_1), (T_2, j'_2)$ *s.t.* $(T_1, j'_1) \neq (T_2, j'_2)$ **and** $\pi^{j'_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{j'_2}_{T_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{j'_1}_{T_1}.\text{sid} = \pi^{j'_2}_{T_2}.\text{sid}$ **then return** 1
8:
9: // If there exist two distinct server sessions with the same cid, $\mathcal{A}$ wins
10: **if** $\exists (S_1, i_1) \neq (S_2, i_2)$ s.t. $\tilde{\pi}^{i_1}_{r,S_1}.\text{cid} = \tilde{\pi}^{i_2}_{r,S_2}.\text{cid} \neq \perp$ **then return** 1
11:
12: // If there exists a server session and a token session that agree on the sid but not on the agCon, $\mathcal{A}$ wins. This captures a rogue key attack by registering a key from another batch.
13: **if** $\exists (S', i', ph'), (T', j', ph')$ s.t. $\tilde{\pi}^{i'}_{ph',S'}.\text{sid} = \tilde{\pi}^{j'}_{ph',T'}.\text{sid} \neq \perp$ **and** $(S', T') \notin \mathcal{L}_{\text{corr}}$ **and** $T'.\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $\tilde{\pi}^{i'}_{ph',S'}.\text{agCon} \neq \tilde{\pi}^{j'}_{ph',T'}.\text{agCon}$ **then return** 1
14:
15: // If the server session $\tilde{\pi}^{i}_{a,S}$ that accepted has no corresponding server registration session, $\mathcal{A}$ wins. If $\mathcal{A}$ never corrupted the token group associated with $\tilde{\pi}^{i'}_{r,S}$, and $\tilde{\pi}^{i}_{a,S}$ has no registration partner T, or $\tilde{\pi}^{i}_{a,S}$ registered with the wrong gid, $\mathcal{A}$ wins. If $\tilde{\pi}^{i}_{a,S}$ is not partnered with one of T's authentication sessions, and $\mathcal{A}$ never corrupted T, $\mathcal{A}$ wins
16: $(\text{gid}, i', T, j', j) \leftarrow \text{Partnerships}(S, i, \text{cid})$
17: **if** gid $= \perp$ **or** $(\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $(T = \perp$ **or** $T.\text{gid} \neq \text{gid}))$ **then return** 1
18: **else**
19: **if** gid $\notin \mathcal{L}_{\text{corrG}}$ **then**
20: // The PIA registration token session $\tilde{\pi}^{j'}_{r,T}$ that is partnered with $\tilde{\pi}^{i'}_{r,S}$ is associated with an mPACA token session $\pi^{l'}_{T}$ that is the Bind partner of a client session $\pi^{k_1}_{C_1}$ that is partnered with $\tilde{\pi}^{i'}_{r,S}$ (unless $\pi^{k_1}_{C_1}$ was compromised).
21: Retrieve (reg, T, j', l') from $\mathcal{L}_{\text{pla-paca-T}}$ // Because we know at this point that $\tilde{\pi}^{j'}_{r,T}$ has completed a registration with $\tilde{\pi}^{i'}_{r,S}$ (from Partnerships), we know from the code in rResp that mPACA session $\pi^{l'}_{T}$ must exist in the list
22: Retrieve $(\text{reg}, S, i', C_S, k_S)$ from $\mathcal{L}_{\text{pla-paca-S}}$
23: $(C_T, k_T) \leftarrow \text{tokenBindPartner}(T, l')$
24: // The client C_T to which the token T is bound is *not* the same as C_S, which is the client linked to the server session $\tilde{\pi}^{i}_{S}$
25: **if** $(C_S, k_S) \neq (C_T, k_T)$ **then**
26: $(T_S, m) \leftarrow \text{clientBindPartner}(C_S, k_S)$
27: **if** $\pi^{k_S}_{C_S}.\text{compromised} = \text{false}$ **and** $\pi^{m}_{T_S}.\text{pinCorr} = \text{false}$ **then return** 1
28: **if** $((C_T, k_T) = (\perp, \perp)$ **or** $\pi^{k_T}_{C_T}.\text{compromised} = \text{false})$ **and** $\pi^{l'}_{T}.\text{pinCorr} = \text{false}$ **then return** 1
29: **if** $(S, T) \notin \mathcal{L}_{\text{corr}}$ **and** $j = \perp$ **then return** 1
30: **else if** $(S, T) \notin \mathcal{L}_{\text{corr}}$ **then**
31: // The PIA authentication token session $\tilde{\pi}^{j}_{a,T}$ that is partnered with $\tilde{\pi}^{i}_{a,S}$ is associated with an mPACA token session π^{l}_{T} that is the Bind partner of a client session $\pi^{k_2}_{C_2}$ that is partnered with $\tilde{\pi}^{i}_{a,S}$ (unless $\pi^{k_2}_{C_2}$ was compromised).
32: Retrieve (auth, T, j, l) from $\mathcal{L}_{\text{pla-paca-T}}$ // Because we know at this point that $\tilde{\pi}^{j}_{a,T}$ has completed an authentication with $\tilde{\pi}^{i}_{a,S}$, we know from the code in aResp that mPACA session π^{l}_{T} must exist in the list
33: Retrieve $(\text{auth}, S, i, C'_S, k'_S)$ from $\mathcal{L}_{\text{pla-paca-S}}$
34: $(C'_T, k'_T) \leftarrow \text{tokenBindPartner}(T, l)$
35: **if** $(C'_S, k'_S) \neq (C'_T, k'_T)$ **then**
36: $(T'_S, m') \leftarrow \text{clientBindPartner}(C'_S, k'_S)$
37: **if** $\pi^{k'_S}_{C'_S}.\text{compromised} = \text{false}$ **and** $\pi^{m'}_{T'_S}.\text{pinCorr} = \text{false}$ **then return** 1
38: **if** $((C'_T, k'_T) = (\perp, \perp)$ **or** $\pi^{k'_T}_{C'_T}.\text{compromised} = \text{false})$ **and** $\pi^{l}_{T}.\text{pinCorr} = \text{false}$ **then return** 1
39: **return** 0

Figure 5.11: The ua security experiment and winning conditions for the PIA+mPACA composed model. We call $\mathcal{O}$ the set of all of the security experiment's oracles that are available to $\mathcal{A}$. The winning condition procedure Win-ua is called in the aCompl oracle whenever the server session accepts. Composed model-specific winning conditions are in blue.

groups and the explicit guarantee that a PIA authentication session is bound to a unique registration session in the same token. Finally, the main novelty is that we explicitly deal with active attacks in the composed model for the registration phase, even for attestation types None and Self, even though in these settings we need to restrict ourselves to uncompromised clients in registration.

5.5.2. PIA+mPACA authentication security

Here we present our main results for composed security of PIA+mPACA, as well as what they mean for rogue key attacks. Then, in Subsection 5.5.3, we contrast our results with the weaker guarantees provided by current FIDO2.

Theorem 5.3 shows that mPACA guarantees a stronger form of composed security than in prior works, since we can allow partial active attacks during both registration and authentication runs, even without attestation. This is a more realistic model when considering USB tokens, as shown with the attacks described in Chapter 4. The intuition here is that the composed model speaks only about the guarantees provided to servers that are uniquely bound to honest clients that are out of the adversary's control: excluding a break of mPACA, we know that such clients (if uncompromised) will be communicating with a unique token instance via a bi-directional secure channel, which means that we do not need to consider an active PIA attacker in the analysis of composed model security.

We provide two theorems, which intuitively capture the security guarantees for different attestation types.

Theorem 5.3. *(Attestation* none *and* self*) Consider the setting in which the adversary is only allowed to create a single attestation group and forbid the possibility to compromise clients involved in registration sessions.*

Then, if there exists an adversary $\mathcal{A}$ against the composed security of an mPACA protocol mPACA and a PIA protocol PIA, then there exist adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that:

$$\mathsf{Adv}^{ua}_{PIA+mPACA}(\mathcal{A}) \leq \mathsf{Adv}^{SUF\text{-}t}_{mPACA}(\mathcal{B}_1) + \mathsf{Adv}^{pla\text{-}auth}_{PIA}(\mathcal{B}_2)\ .$$

Furthermore, $\mathcal{B}_2$ only requires access to a passive PIA registration oracle.

Proof. (Sketch.) Our proof proceeds in two game hops. In the first hop, we modify the composed security model so that the adversary loses whenever it breaks the mPACA security guarantee when interacting with oracles rResp, rCompl, aResp or aCompl (seen in

Fig. 5.12). Any adversary $\mathcal{A}$ for which the probability of winning the composed security game varies visibly can be transformed into an mPACA attacker $\mathcal{B}_1$ with the same advantage via a trivial reduction. Note that $\mathcal{B}_1$ controls all the details of the PIA protocol in this reduction. Also observe that, after this hop, and because we disallow the compromise of clients involved in registration runs, the adversary is now restricted to passive behavior when dealing with these sessions.

In the second hop, $\mathcal{A}$ loses if it breaks a PIA guarantee (seen in Fig. 5.13). We reduce any distinguishing advantage between the two games by constructing a reduction $\mathcal{B}_2$ to PIA security. Note here that we need to argue that we can program the trace of a passive registration run into the composed model oracles, where the adversary has some active attacking power. Here is how the reduction can do this:

- when the adversary calls rChall, there are two cases. Either the client has a unique token partner, or it does not have any binding state: this is guaranteed by passive binding and the mPACA winning guarantees. If the client has no binding state, then note that the authentication oracle Auth-C will fail and so will the rChall oracle. Otherwise, the reduction can pinpoint the unique token and choose an unused PIA registration session to use its own PIA registration oracle and obtain a registration trace. The first message in this trace is programmed into the output of rChall.
- when the adversary calls rResp, it is either the case that it is delivering the message to the unique mPACA oracle associated with some prior registration query or not. If not, then the mPACA validate condition will fail and there is nothing to do. Otherwise, the reduction maps the j-th token session in the composed model to the token session in the PIA model that it preemptively chose to obtain the registration trace. The second message of the passive registration trace is programmed in the output of the oracle.
- when the adversary calls rCompl, the reduction simply needs to check if the message received comes from an mPACA token session that is linked with the PIA token session that is the registration partner of the server session that was involved in the initial rChall oracle.

The remaining part of the behavior of B_2 is a simple reduction where it uses its own oracles to answer the queries placed by $\mathcal{A}$.

rResp $(T, j, j', m_{\text{rcom}}, t_{\text{cl}}, d)$:
1: **if** $\tilde{\pi}^{j}_{r,T} \neq \perp$ **or** $T.\text{gid} = \perp$ **then return** $\perp$
2: status $\xleftarrow{\$}$ Validate-T$(T, j', m_{\text{rcom}}, t_{\text{cl}}, d)$
3: **if** status $\neq$ accepted **then return** $\perp$
4: **if** Token-Win-SUF-t$(T, j', m_{\text{rcom}}, t_{\text{cl}}, d) = 1$ **then**
5: game ends returning 0
6: $(m_{\text{rrsp}}, \text{rc}_T, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{rRsp}(\tilde{\pi}^{j}_{r,T}, m_{\text{rcom}})$
7: $(m_{\text{rrsp}}, t_{\text{tk}}) \xleftarrow{\$} \text{Auth-T}(T, j', m_{\text{rrsp}})$
8: $\mathcal{L}_{\text{pla-paca-T}} \leftarrow \mathcal{L}_{\text{pla-paca-T}} \cup \{(\text{reg}, T, j, j')\}$
9: **return** $(m_{\text{rrsp}}, t_{\text{tk}})$

rCompl $(S, i, m_{\text{rcl}}, m_{\text{rrsp}}, \text{gid}, t_{\text{tk}})$:
1: **if** $\tilde{\pi}^{i}_{r,S} = \perp$ **or** $\tilde{\pi}^{i}_{r,S}.\text{st}_{\text{exe}} \neq$ running **or** $G[\text{gid}] = \perp$ **then return** $\perp$
2: Retrieve (reg, S, i, C, k) from $\mathcal{L}_{\text{pla-paca-S}}$
3: status $\leftarrow$ Validate-C$(C, k, m_{\text{rrsp}}, t_{\text{tk}})$
4: **if** status $\neq$ accepted **then return** $\perp$
5: **if** Client-Win-SUF-t$(C, k, m_{\text{rcom}}, t_{\text{tk}}) = 1$ **then**
6: game ends returning 0
7: $(d, \text{rc}_S, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$}$ rVrfy$(\tilde{\pi}^{i}_{r,S}, m_{\text{rcl}}, m_{\text{rrsp}}, G[\text{gid}].\text{gpars})$
8: **if** $d = 1$ **then**
9: $\mathcal{L}_{\text{reg}} \leftarrow \mathcal{L}_{\text{reg}} \cup \{(S, i, \text{gid}, \text{cid})\}$
10: **return** d

aResp $(T, j, j', m_{\text{acom}}, t_{\text{cl}}, d)$:
1: **if** $\tilde{\pi}^{j}_{a,T} \neq \perp$ **or** $T.\text{gid} = \perp$ **then return** $\perp$
2: status $\xleftarrow{\$}$ Validate-T$(T, j', m_{\text{acom}}, t_{\text{cl}}, d)$
3: **if** status $\neq$ accepted **then return** $\perp$
4: **if** Token-Win-SUF-t$(T, j', m_{\text{acom}}, t_{\text{cl}}, d) = 1$ **then**
5: game ends returning 0
6: $(m_{\text{arsp}}, \text{rc}_T, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$} \text{aRsp}(\tilde{\pi}^{j}_{a,T}, m_{\text{acom}})$
7: $(m_{\text{arsp}}, t_{\text{tk}}) \xleftarrow{\$} \text{Auth-T}(T, j', m_{\text{arsp}})$
8: $\mathcal{L}_{\text{pla-paca-T}} \leftarrow \mathcal{L}_{\text{pla-paca-T}} \cup \{(\text{auth}, T, j, j')\}$
9: **return** $(m_{\text{arsp}}, t_{\text{tk}})$

aCompl $(S, i, m_{\text{acl}}, m_{\text{arsp}}, t_{\text{tk}})$:
1: **if** $\tilde{\pi}^{i}_{a,S} = \perp$ **or** $\tilde{\pi}^{i}_{a,S}.\text{st}_{\text{exe}} \neq$ running **then return** $\perp$
2: Retrieve $(\text{auth}, S, i, C, k)$ from $\mathcal{L}_{\text{pla-paca-S}}$
3: status $\leftarrow$ Validate-C$(C, k, m_{\text{arsp}}, t_{\text{tk}})$
4: **if** status $\neq$ accepted **then return** $\perp$
5: **if** Client-Win-SUF-t$(C, k, m_{\text{acom}}, t_{\text{tk}}) = 1$ **then**
6: game ends returning 0
7: $(d, \text{rc}_S, \text{cid}, \text{sid}, \text{agCon}) \xleftarrow{\$}$ aVrfy$(\tilde{\pi}^{i}_{a,S}, m_{\text{acl}}, m_{\text{arsp}})$
8: **if** $d = 1$ **and** win-ua $= 0$ **then**
9: win-ua $\leftarrow$ Win-ua(S, i, cid)
10: **return** d

Figure 5.12: First hop of the composed model proof. Code in red represents the changes to the code corresponding to the first hop. The changes in Win-ua corresponding to breaking mPACA are redundant, because if the game reaches Win-ua, then it must have passed through mPACA and triggered the same conditions inside Client-Win-SUF-t.

At this point, the composed model adversary can only win if it breaks a composed model-specific condition: the server accepts an authentication run, but the bound PIA oracles are not residing in the token that is bound to the correct mPACA client.

This can only happen when at least one of the conditions in lines 19, 20, 28 or 29 from Figure 5.13 (corresponding to lines 27, 28, 37 or 38 from Figure 5.11) is true, which requires that the client session bound to the token that completed the registration/authentication run is not the same as the client session linked to the server session that finished that registration/authentication run. Also note that, if these winning conditions were activated, it is at a point in the experiment where we can always find the unique relationship between registration and authentication sessions of servers and tokens, via the Partnerships method. We now explain why this cannot occur.

The condition in line 19 models the rogue key attack scenario: the adversary is able to register a key generated by a token PIA session that is not protected by the mPACA session of the client that is connected to the server, and this mPACA session is not compromised. Note that for the server to have accepted the registered key, the client connected to it must have accepted the message containing that key. However, this would have implied an mPACA break which we excluded in hop 1.

Win-ua (S, i, cid):

1: **if** $\exists (S_1, i_1, ph_1) \neq (S_2, i_2, ph_2)$ s.t. $\tilde{\pi}^{i_1}_{ph_1,S_1}.\text{sid} = \tilde{\pi}^{i_2}_{ph_2,S_2}.\text{sid} \neq \perp$ **then** game ends returning 0
2: **if** $\exists (T_1, j_1, ph_1) \neq (T_2, j_2, ph_2)$ s.t. $\tilde{\pi}^{j_1}_{ph_1,T_1}.\text{sid} = \tilde{\pi}^{j_2}_{ph_2,T_2}.\text{sid} \neq \perp$ **then** game ends returning 0
3: **if** $\exists (C_1, k_1), (C_2, k_2)$ *s.t.* $(C_1, k_1) \neq (C_2, k_2)$ **and** $\pi^{k_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{k_2}_{C_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{k_1}_{C_1}.\text{sid} = \pi^{k_2}_{C_2}.\text{sid}$ **then return** 1
4: **if** $\exists (T_1, j'_1), (T_2, j'_2)$ *s.t.* $(T_1, j'_1) \neq (T_2, j'_2)$ **and** $\pi^{j'_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{j'_2}_{T_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{j'_1}_{T_1}.\text{sid} = \pi^{j'_2}_{T_2}.\text{sid}$ **then return** 1
5:
6: **if** $\exists (S_1, i_1) \neq (S_2, i_2)$ s.t. $\tilde{\pi}^{i_1}_{r,S_1}.\text{cid} = \tilde{\pi}^{i_2}_{r,S_2}.\text{cid} \neq \perp$ **then** game ends returning 0
7:
8: **if** $\exists (S', i', ph'), (T', j', ph')$ s.t. $\tilde{\pi}^{i'}_{ph',S'}.\text{sid} = \tilde{\pi}^{j'}_{ph',T'}.\text{sid} \neq \perp$ **and** $(S', T') \notin \mathcal{L}_{\text{corr}}$ **and** $T'.\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $\tilde{\pi}^{i'}_{ph',S'}.\text{agCon} \neq \tilde{\pi}^{j'}_{ph',T'}.\text{agCon}$ **then** game ends returning 0
9:
10: $(\text{gid}, i', T, j', j) \leftarrow \text{Partnerships}(S, i, \text{cid})$
11: **if** $\text{gid} = \perp$ **or** $(\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **and** $(T = \perp$ **or** $T.\text{gid} \neq \text{gid}))$ **then** game ends returning 0
12: **else**
13: **if** $\text{gid} \notin \mathcal{L}_{\text{corrG}}$ **then**
14: Retrieve (reg, T, j', l') from $\mathcal{L}_{\text{pla-paca-T}}$
15: Retrieve $(\text{reg}, S, i', C_S, k_S)$ from $\mathcal{L}_{\text{pla-paca-S}}$
16: $(C_T, k_T) \leftarrow \text{tokenBindPartner}(T, l')$
17: **if** $(C_S, k_S) \neq (C_T, k_T)$ **then**
18: $(T_S, m) \leftarrow \text{clientBindPartner}(C_S, k_S)$
19: **if** $\pi^{k_S}_{C_S}.\text{compromised} = \text{false}$ **and** $\pi^{m}_{T_S}.\text{pinCorr} = \text{false}$ **then return** 1
20: **if** $((C_T, k_T) = (\perp, \perp)$ **or** $\pi^{k_T}_{C_T}.\text{compromised} = \text{false})$ **and** $\pi^{l'}_{T}.\text{pinCorr} = \text{false}$ **then return** 1
21: **if** $(S, T) \notin \mathcal{L}_{\text{corr}}$ **and** $j = \perp$ **then** game ends returning 0
22: **else if** $(S, T) \notin \mathcal{L}_{\text{corr}}$ **then**
23: Retrieve (auth, T, j, l) from $\mathcal{L}_{\text{pla-paca-T}}$
24: Retrieve $(\text{auth}, S, i, C'_S, k'_S)$ from $\mathcal{L}_{\text{pla-paca-S}}$
25: $(C'_T, k'_T) \leftarrow \text{tokenBindPartner}(T, l)$
26: **if** $(C'_S, k'_S) \neq (C'_T, k'_T)$ **then**
27: $(T'_S, m') \leftarrow \text{clientBindPartner}(C'_S, k'_S)$
28: **if** $\pi^{k'_S}_{C'_S}.\text{compromised} = \text{false}$ **and** $\pi^{m'}_{T'_S}.\text{pinCorr} = \text{false}$ **then return** 1
29: **if** $((C'_T, k'_T) = (\perp, \perp)$ **or** $\pi^{k'_T}_{C'_T}.\text{compromised} = \text{false})$ **and** $\pi^{l}_{T}.\text{pinCorr} = \text{false}$ **then return** 1
30: **return** 0

Figure 5.13: Second hop of the composed model proof. Code in red represents the changes to the code corresponding to the second hop.

The condition in line 20 models the scenario where $\mathcal{A}$ is successful in registering a key generated by a token that is out of its control: the adversary, controlling a possibly compromised client that is connected to the server, registers a key generated by a token PIA session in an uncorrupted token mPACA session that has no relation to the client controlled by the adversary. This would imply the adversary was able to break mPACA by breaking into this token and convincing it to answer a response request, which we excluded in hop 1.

The justification for the winning conditions in lines 28 and 29 being unreachable is the same as above, but these lines correspond to different practical attack scenarios that are excluded by our proof. In line 28, the adversary would be hijacking an authentication session established via an uncorrupted client and authenticating using its own token. In line 29, the adversary is able to break into the user's token and impersonate the user in

an authenticated session that it controls.

This leaves only one last option for $\mathcal{A}$ to try to win, which is through the conditions in lines 3 and 4 (corresponding to lines 6 and 7 from Figure 5.11). However, these are also never reached, since they mean that $\mathcal{A}$ wins by breaking the mPACA guarantees regarding uniqueness of sid values in client and token sessions, which is not also possible after the first hop.

Therefore, at this point $\mathcal{A}$ can never win the game, and the proof is concluded.

□

Theorem 5.4. *(Attestation* basic*) If there exists an adversary $\mathcal{A}$ against the composed security of an mPACA protocol $mPACA$ and a PIA protocol PIA, then there exist adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that:*

$$\mathsf{Adv}^{ua}_{PIA+mPACA}(\mathcal{A}) \leq \mathsf{Adv}^{SUF\text{-}t}_{mPACA}(\mathcal{B}_1) + \mathsf{Adv}^{pla\text{-}auth}_{PIA}(\mathcal{B}_2) \ .$$

The proof of this theorem is similar to the previous one, only that the reduction to PIA security is now simpler given the adaptive power of the adversary.

Note that our results for the mPACA security of CTAP 2.1+ in Section 5.3, and the results we describe in Section 5.4 for the security of WebAuthn in various attestation types imply that these composed security results apply to this improved version of FIDO2. We next discuss how our results capture rogue key attacks.

Rogue key attacks

Recall the rogue key attack, which can be seen in Figure 5.14. This attack is possible against the current instantiation of FIDO2 because the client has no way of verifying the origin of a token response (in CTAP2) *and* because the server may not have any information that *uniquely* identifies the token from which is expects a response (in WebAuthn). That is indeed the case for the most common attestation types None, Self and Basic, although in the latter case it may have some information that allows recognizing a target batch (or group)[11].

The PIA+PACA security model in [2] does not capture these rogue key attacks, because in the entire registration flow it is assumed that the adversary is passive. If this was not the

[11] A relying party might, for example, set a rule that only tokens from a specific manufacturer can be used, which would effectively allow it to reject attestation signatures from every token that is a part of the manufacturer's batch.

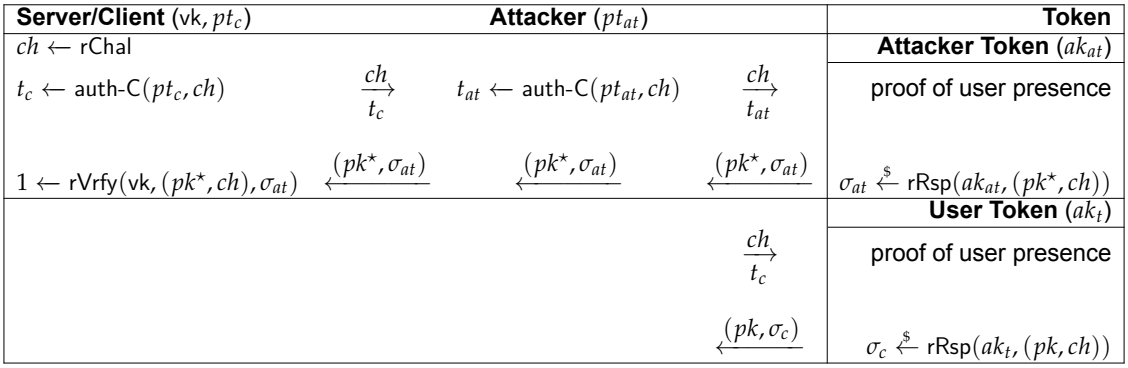


Figure 5.14: The rogue key attack (against a registration run) in Basic attestation. The diagram is based on Figure 4.3, which we recover here for convenience and with names of functions and variables adapted to the syntaxes introduced in the previous Sections. Procedures are simplified for better presentation. If the attacker has a token from the same batch as the user, then $ak_{at} = ak_t$.

case, then the rogue key attack would lead to a break of the protocol in this model because the messages sent by the token to the client are not authenticated. This means that an adversary can simply use its PACA oracles in the security experiment to gain control over some other token, use it to generate a credential for the challenge created by the target server, and feed it to the honest client who will pass it along to the server.

Even if one considers Basic attestation, and therefore active composed model adversaries during registration, PIA offers at most the guarantee that a token from a specific batch is registered, but it cannot ensure that it is the intended user token. Rogue key attacks can still occur within the batch.

To see how our result addresses rogue key attacks, in *all* attestation types, consider the scenario where an active attacker is trying to launch a rogue key attack against the composed PIA+mPACA protocol, but it does not have the ability to corrupt the client that the user is relying on. Then, our results above guarantee that the server will only accept a credential generated by the token that is uniquely bound to that client, which in turn is uniquely bound by a TLS connection to the server. As soon as the client is under the adversary's control this guarantee no longer holds, and rogue key attacks can take place. We note that this is only true for the upgraded version of mPACA that we proposed, and does not hold for PIA+PACA composed protocols.

Summary of the results

The two theorems and respective proofs stated above establish the following intuitive results:

- For attestation types None and Self one must consider that registration is carried out via an honest client, as otherwise one gets no PIA guarantees. In particular, the server would not even be assured that the credential comes from a hardware token.
- For attestation type Basic we can guarantee PIA security even if the server is interacting with a compromised client. However, rogue key attacks are still possible in this setting.
- For all attestation types, if a token is interacting with an honest client, then the token will only reply to challenges if it is unlocked by a client or attacker using the correct user PIN.
- For all attestation types, if an interaction with the server occurs via an honest client session (be it in authentication or registration), then the composed PIA+mPACA security guarantees that the server session connected to that honest client session will only ever accept from a PIA session originating in the unique token session bound to that client session via mPACA. *This means in particular that we guarantee security against rogue key attacks.*

Next, we present a brief discussion on the guarantees provided by the current version of FIDO2.

5.5.3. Composed authentication security of FIDO2

When considering the current version of FIDO2 we can no longer rely on its PACA component CTAP2 to resolve the problem of rogue key attacks, and must therefore consider the weaker TOFU security model for composed security to hold, as is done in [2]. However, even in this case, the adversary can try to take advantage of the lack of authentication in messages going from the token back to the client that is bound to the server. The composed model guarantee for current FIDO2 is therefore much weaker than what we have presented above. We describe the implications next.

First of all, one cannot guarantee that the server only accepts an authentication response that comes from the token that is PACA-bound to the client. Indeed, the attacker could potentially convince the client to send back to the server a response that comes from his own maliciously-controlled token, thereby leading the server to log-in the user under a different account. Second, even though such an attack is still possible, the role of PACA in composed FIDO2 security is still relevant: it guarantees that, if the user's token is not compromised and only interacts with honest clients, then the attacker cannot break

into the token and impersonate the user. Formally this can be captured by taking the approach in [2] to composed model security: one requires that the attacker cannot break PIA security and, furthermore, that tokens bound to honest clients only issue responses to PIA authentication requests if these requests come from their unique PACA partner.

In the next Chapter, we propose a different modification to CTAP2 that provably makes it resistant to active attacks during Bind, which includes man-in-the-middle attacks on the ECDH key exchange *and* the aforementioned mauled ciphertext attack that enables an attacker to perform offline dictionary attacks on the PIN.

6. FIDO2 security against active binding attacks

In Chapter 5, we analysed the authentication security of FIDO2, showing the protocol is not resistant to rogue key attacks like those described in Chapter 4 because neither the client nor the server can cryptographically pinpoint the origin of a response to a unique token during registration. To fix this problem, and effectively eliminate an entire class of attacks based on hijacking a token's response to a client command, we proposed CTAP 2.1+ as an improvement over CTAP 2.1, which adds bi-directional authentication between clients and tokens, ensuring that a client can only ever accept a response from a token with which it previously established an authenticated channel.

Strengthening the SUF-t trust model To prove CTAP 2.1+ is resistant to rogue key attacks, we extended the PACA syntax and security model used in [2] to mPACA. In this new model, we captured rogue key attacks by adding a new winning condition where an adversary could win if it successfully forged a response that is accepted by a client session. Under the new mPACA security model, we were able to prove that CTAP 2.1+ satisfies SUF-t security, just as CTAP 2.1 achieves SUF-t security under the PACA security model.

However, we also showed that CTAP 2.1+ (and by extension CTAP 2.1) cannot be proven secure if the adversary is active during Bind. The *trusted binding* restriction is *necessary* to prove the authentication security of any version of CTAP2. If an adversary has the power to deliver an ECDH public key to a client session, then it can perform the man-in-the-middle attack described in Chapter 4. Even in the partially trusted binding setting introduced in [2], where the adversary can only deliver the last Bind message to the client session, the model proves to be too strong, enabling the adversary to perform *offline* dictionary attacks on the PIN.

One could strengthen CTAP 2.1+ further, so that it can be proven secure under the *trusted binding* setting from [2], by authenticating the pinUvAuthToken ciphertext sent by the token to the client in the last message of Bind. This is exactly what occurs when the encrypted PIN is transmitted to the token during Setup, and ensures an adversary cannot simply alter the ciphertext and have the client decrypt the result.

However, to prove CTAP 2.1+ secure against *all* active attacks during Bind, one needs to prove its SUF security (without any kind of *trusted binding*), as initially described in [1].

This means the trust model must allow the adversary to freely interact with client and token sessions during the entire Bind subprotocol. It is easy to see that an adversary can trivially win against the SUF experiment when mPACA is instantiated with CTAP 2.1+, by obtaining the pinUvAuthToken or the PIN hash through the man-in-the-middle attack without having to corrupt or compromise any session oracle. As a result, CTAP 2.1+ (and by extension any version of CTAP2) cannot be proven SUF secure in the PACA or mPACA security model. To stop the man-in-the-middle attack from breaking an mPACA protocol, and thus achieve SUF security, the Bind subprotocol that produces the shared binding state between a token and a client must be authenticated somehow.

In this Chapter, we show how CTAP 2.1+ can be further improved to provide security against man-in-the-middle attacks, by authenticating the entire Bind procedure with a Password-Authenticated Key Exchange (PAKE) [19, 20]. Then, we prove that the resulting protocol, which we call sCTAP 2.1+ (for *secure CTAP 2.1+*), is SUF secure in our mPACA model.

We start in Section 6.1, where we provide a brief description of the PAKE-based solution for CTAP 2.0 proposed by Barbosa et al. in [1], and then explain our novel results in Section 6.2. In Section 6.3, we introduce our updated mPACA model, on which we remove the *trusted binding* setting and define the SUF security experiment. Then, in Section 6.4, we introduce the syntax of sCTAP 2.1+, as well as how and where it differs from CTAP 2.1+[1]. In Section 6.5, we show our results on the SUF security of sCTAP 2.1+ as an mPACA protocol. Finally, in Section 6.6, we overview some concrete PAKE protocols we believe may be viable options for integration with sCTAP 2.1+.

6.1. Background

To the best of our knowledge, only Barbosa et al. [1] proposed modifying CTAP2 to achieve resistance against man-in-the-middle attacks during Bind. The modified protocol, which the authors named sPACA (for *secure* PACA), uses a PAKE instead of an unauthenticated ECDH key exchange to resist active attacks. We provide a brief overview of their work next.

[1]Note that, because CTAP 2.1+ and CTAP 2.1 are identical until Bind is finished, and because sCTAP 2.1+ only differs from CTAP 2.1+ until Bind is finished, any comparison drawn between sCTAP 2.1+ and CTAP 2.1+ also applies to CTAP 2.1 (since the modifications from sCTAP 2.1+ could all be applied to CTAP 2.1).

The sPACA protocol Recall from Chapter 5 that the authors showed CTAP 2.0 only achieves UF-t security, due to the unauthenticated ECDH key exchange and the possibility of multiple clients sharing the same pinToken. To improve the protocol, the authors proposed a new protocol called sPACA, which adds two crucial modifications to CTAP 2.0: (i) the replacement of the unauthenticated ECDH key exchange with a Password-Authenticated Key Exchange (PAKE), and (ii) the replacement of the pinToken as the MAC key that authenticates commands with the session key derived from the PAKE. With the first change enabling the removal of the *trusted binding* setting, and the second enabling the strong variant of the unforgeability experiment, the authors were able to prove the SUF security of sPACA[2] in their PACA model. The protocol replaces the entire flow of communications during Bind but leaves Setup unaffected, where ECDH is still used to set the PIN hash on the authenticator. Consequently, while trusted binding is no longer necessary, the trust-on-first-use setting is still maintained during Setup.

Instantiating sPACA While the authors do not propose to instantiate sPACA with a specific PAKE, they do note that it can be instantiated with protocols like CPace [46] or SPAKE2 [47, 48], as long as they include *explicit authentication*. Indeed, this part is crucial if one wishes to implement Bind with a PAKE, because the authenticator must be capable of detecting wrong PIN guesses. However, note that this explicit authentication is crucial from the client to the token only. In sPACA, the binding state is set as the derived session key from the PAKE, and therefore the token no longer sends the pinToken back to the client in the last step of Bind (in fact, the pinToken does not even exist in sPACA). This means the client's binding state can be set *before* it sends its message with the PIN attempt to the token and, consequently, that the last token-to-client message of Bind is no longer *strictly* necessary for each party to agree on the binding state. As a result, and as the authors point out, this last step may be excluded for efficiency gains[3] without impact on the proof. Nevertheless, they chose to keep this last message flow to achieve *explicit mutual authentication*. With this step, the message flow of sPACA stays the same as that of CTAP 2.0, where the last message from the token to the client conveys the information that the former accepted the PIN attempt and has derived its binding state.

[2] As the authors note in the paper, the ability to prove the *strong unforgeability* of sPACA comes from replacing the pinToken with the session key output by the PAKE. If this session key was used to replace the shared secret that encrypts the pinToken instead, then the protocol would still only be UF secure (recall that CTAP 2.0 allows multiple clients to be simultaneously partnered with the same token).

[3] The authors also note that if the last step of Bind is removed, because the client already has its binding state, then it could also send the authenticated command for registration or authentication, for example, with the message containing the PIN attempt. This would spare an entire round-trip, but would require more modifications to CTAP2's established flow.

6.2. Our results

In this work, we update the results from Barbosa et al. [1] to CTAP 2.1+ (and by extension CTAP 2.1, since the two protocols are identical during Bind), including the more fine-grained model, proof and protocol syntax. The new protocol, which we call sCTAP 2.1+, implements Bind with a PAKE in a way that perfectly fits with the current CTAP2 flow. Concretely, our main contributions in this Chapter are:

- updating the CTAP 2.1+ syntax to include the modifications required to replace the ECDH key exchange with a generic PAKE;
- the introduction of a new PIN/UV Auth Protocol, used in sCTAP 2.1+, which provides the low-level PAKE logic exposed through the (slightly modified) interface defined in the CTAP 2.1 specification;
- removing the trusted binding setting over the mPACA model by reintroducing the Send-Bind-C oracle without the restrictions from [2], i.e., by allowing the adversary to call this oracle on *any* client session and on *every* Bind state of execution;
- providing a detailed, pseudocode proof of the SUF security of sCTAP 2.1+, as an mPACA protocol.

6.3. Updated authentication model

We use the same mPACA authentication model introduced in Chapter 5, with some modifications to allow the adversary to actively engage with client sessions. Then, we define the SUF security of an mPACA protocol with a new security experiment $\mathsf{Expt}^{\mathsf{SUF}}_{\mathsf{mPACA}}$, executed between a challenger and an adversary $\mathcal{A}$.

Trust model As is the case with any PACA or mPACA *unforgeability* experiment, we assume a fully authenticated communication channel between clients and tokens during Setup. Additionally, we now remove the *trusted binding* restriction, and allow adversaries to launch active attacks on both client and token sessions during Bind, at any stage of execution. Passive executions via the Execute oracle remain unchanged from the other security experiments.

Session oracles and variables Every session oracle and variable present in the SUF-t experiment is naturally inherited here. Additionally, we introduce a new boolean variable $\pi^{j}_{C}.\mathsf{pinCorr}$ for client sessions, due to the fact that an adversary that corrupts the user

$\text{Expt}^{\text{SUF}}_{\text{mPACA}}(\mathcal{A})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: win-SUF $\leftarrow 0$
4: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
5: **return** win-SUF

tokenBindPartner (T, i):
1: **if** $\exists (C, j)$ s.t. $\pi^i_T.\text{sid} = \pi^j_C.\text{sid}$ **then**
2: **return** (C, j)
3: **return** $(\perp, \perp)$

Token-Win-SUF (T, i, M, t, d):
1: **if** $d \neq$ accepted **then return** 1
2:
3: **if** $\exists (C_1, j_1), (C_2, j_2)$ $s.t.$ $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{j_2}_{C_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{j_1}_{C_1}.\text{sid} = \pi^{j_2}_{C_2}.\text{sid}$ **then return** 1
4:
5: **if** $\exists (T_1, i_1), (T_2, i_2)$ $s.t.$ $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{i_2}_{T_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{i_1}_{T_1}.\text{sid} = \pi^{i_2}_{T_2}.\text{sid}$ **then return** 1
6:
7: $(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
8: **if** $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
9: **if** $(C, j) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
10: **if** $\pi^i_T.\text{pinCorr} = \text{false}$ **then**
11: **return** 1
12: **return** 0

clientBindPartner (C, j):
1: **if** $\exists (T, i)$ s.t. $\pi^j_C.\text{sid} = \pi^i_T.\text{sid}$ **then**
2: **return** (T, i)
3: **return** $(\perp, \perp)$

Client-Win-SUF (C, j, M, t):
1: **if** $\exists (C_1, j_1), (C_2, j_2)$ $s.t.$ $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{j_2}_{C_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{j_1}_{C_1}.\text{sid} = \pi^{j_2}_{C_2}.\text{sid}$ **then return** 1
2:
3: **if** $\exists (T_1, i_1), (T_2, i_2)$ $s.t.$ $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{i_2}_{T_2}.\text{st}_{\text{exe}} =$ bindDone **and** $\pi^{i_1}_{T_1}.\text{sid} = \pi^{i_2}_{T_2}.\text{sid}$ **then return** 1
4:
5: $(T, i) \leftarrow \text{clientBindPartner}(C, j)$
6: **if** $(T, i, M, t) \notin \mathcal{L}_{\text{authT}}$ **then**
7: **if** $(T, i) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
8: **if** $\pi^j_C.\text{pinCorr} = \text{false}$ **then**
9: **return** 1
10: **return** 0

Figure 6.1: mPACA SUF security experiment and winning conditions. Changes from the SUF-t experiment are in blue. $\mathcal{O}$ denotes all oracles available to $\mathcal{A}$, as shown in Figure 6.2.

pin is now able to complete Bind with a client session π^j_C without interacting with a token session.

Experiment oracles and winning conditions Figure 6.2 shows the $\text{Expt}^{\text{SUF}}_{\text{mPACA}}$ oracles, where the only differences from the mPACA SUF-t experiment are the addition of a new oracle Send-Bind-C, which we recover from [2] and expand to handle any part of the Bind process, and a tweak to the Execute oracle to update the state of the pin corruption. Concretely, Send-Bind-C is analogous to Send-Bind-T, in that it allows an adversary to send a message m to a client session π^j_C to initiate, continue or complete a Bind procedure, without interaction with any token session. This is also reflected in the $\text{Expt}^{\text{SUF}}_{\text{mPACA}}$ winning conditions in Figure 6.1, where an adversary additionally wins if it can successfully finish Bind with a client session without that session having a partner token session and without corrupting the pin.

Advantage measure The authentication advantage of $\mathcal{A}$ against the SUF security experiment of an mPACA protocol mPACA is defined as:

NewT (T, initialData):
1: **if** $\text{st}_T \neq \perp$ **then return** $\perp$
2: st_T.initialData $\leftarrow$ initialData
3: Reboot(st_T)
4: **return**

CorruptUser (U):
1: **if** $\mathcal{L}_{\text{valid}}[U] = \perp$ **then return** $\perp$
2: $\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ true
3: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
4: **return** pin

Reboot (T):
1: **if** $\text{st}_T = \perp$ **then return** $\perp$
2: **for all** i *s.t.* $\pi_T^i \neq \perp$ **do**
3: $\quad \pi_T^i$.isValid $\leftarrow$ false
4: Reboot(st_T)
5: **return**

Compromise (C, j):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **then return** $\perp$
2: π_C^j.compromised = True
3: **return** π_C^j.bs

Send-Bind-T (T, i, m):
1: **if** $\text{st}_T = \perp$ **then return** $\perp$
2: **if** $\pi_T^i = \perp$ **then**
3: $\quad \pi_T^i \leftarrow \text{st}_T$
4: **if** $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone **or** π_T^i.isValid = false **then return** $\perp$
5: π_T^i.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
6: **if** $\pi_T^i.\text{st}_{\text{exe}} =$ waiting **then**
7: $\quad m_T \leftarrow$ Bind-T(π_T^i, m)
8: $\quad c_{pt} \,||$ calledReboot $\leftarrow m_T$
9: $\quad$ **if** calledReboot = true **then**
10: $\quad\quad$ **for all** i' *s.t.* $\pi_T^{i'} \neq \perp$ **do**
11: $\quad\quad\quad \pi_T^{i'}$.isValid $\leftarrow$ false
12: $\quad$ **else if** $\pi_T^i.\text{st}_{\text{exe}} =$ bindDone **then**
13: $\quad\quad$ **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
14: $\quad\quad\quad \pi_T^{i'}$.isValid $\leftarrow$ false
15: **else**
16: $\quad m_T \leftarrow$ Bind-T(π_T^i, m)
17: **return** m_T

Auth-C (C, j, M):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **or** π_C^j.isValid = false **then return** $\perp$
2: $(M, t) \xleftarrow{\$}$ auth-C(π_C^j, M)
3: $\mathcal{L}_{\text{authC}} \xleftarrow{\cup} \{(C, j, M, t)\}$
4: **return** (M, t)

Auth-T (T, i, M):
1: **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq$ bindDone **or** π_T^i.isValid = false **then return** $\perp$
2: $(M, t) \xleftarrow{\$}$ auth-T(π_T^i, M)
3: $\mathcal{L}_{\text{authT}} \xleftarrow{\cup} \{(T, i, M, t)\}$
4: **return** (M, t)

NewU (U):
1: **if** $\mathcal{L}_{\text{valid}}[U] = \perp$ **then**
2: $\quad$ pin $\xleftarrow{\$} \mathcal{P}$
3: $\quad \mathcal{L}_{\text{valid}}[U] \leftarrow$ pin
4: $\quad \mathcal{L}_{\text{corrupt}}[U] \leftarrow$ false
5: **return**

Setup (T, i, C, j, U):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
2: **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
3: $\pi_T^i \leftarrow \text{st}_T$
4: trans $\xleftarrow{\$}$ Setup(π_T^i, π_C^j, pin)
5: π_T^i.isValid, π_C^j.isValid $\leftarrow$ false
6: st_T.user $\leftarrow U$
7: **return** trans

Execute (T, i, C, j):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
2: **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
3: $\pi_T^i \leftarrow \text{st}_T$
4: π_C^j.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
5: trans, $m_C, m_T \leftarrow \perp$
6: **while** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **do**
7: $\quad m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
8: $\quad m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
9: $\quad$ trans $\leftarrow$ trans $|| \, m_T \, || \, m_C$
10: **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
11: $\quad \pi_T^{i'}$.isValid $\leftarrow$ false
12: **return** trans

Send-Bind-C (C, j, m, U):
1: **if** $\pi_C^j = \perp$ **then**
2: $\quad \pi_C^j$.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}[U]$
3: **if** $\pi_C^j.\text{st}_{\text{exe}} =$ bindDone **then return** $\perp$
4: **if** π_C^j.isValid = false **then return** $\perp$
5: **if** $\mathcal{L}_{\text{valid}}[U] = \perp$ **then return** $\perp$
6: $m_C \leftarrow$ Bind-C(π_C^j, U, m)
7: **return** m_C

Validate-T (T, i, M, t, d):
1: **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq$ bindDone **or** π_T^i.isValid = false **then return** $\perp$
2: status $\leftarrow$ validate-T(π_T^i, M, t, d)
3: **if** status = accepted **then** win-SUF $\leftarrow$ Token-Win-SUF(T, i, M, t, d)
4: **return** status

Validate-C (C, j, M, t):
1: **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **or** π_C^j.isValid = false **then return** $\perp$
2: status $\leftarrow$ validate-C(π_C^j, M, t)
3: **if** status = accepted **then** win-SUF $\leftarrow$ Client-Win-SUF(C, j, M, t)
4: **return** status

Figure 6.2: Oracles for mPACA SUF security experiment. Changes from the SUF-t experiment are in blue.

$$\mathsf{Adv}^{\mathsf{SUF}}_{\mathsf{mPACA}}(\mathcal{A}) = \Pr[\mathsf{Expt}^{\mathsf{SUF}}_{\mathsf{mPACA}}(\mathcal{A}) = 1] \,.$$

6.4. sCTAP 2.1+ as an mPACA protocol

In this Section, we introduce the high and low level syntax of sCTAP 2.1+ as an mPACA protocol, and then highlight its main differences to CTAP 2.1+.

6.4.1. Session and protocol variables

Every session and protocol variable from CTAP 2.1/CTAP 2.1+ is naturally inherited in sCTAP 2.1+. Additionally, in sCTAP 2.1+ we add two new variables to the token's internal state: $\mathsf{st}_T.\mathsf{id}_U$ and $\mathsf{st}_T.\mathsf{id}_T$. These contain the user party identifier id_U, which corresponds to the id_C introduced in the PAKE syntax (see Section 2.4 from Chapter 2) since in CTAP2 it is the user that possesses the pin, and the token party identifier id_T, both of which generated and set during Setup. For sCTAP 2.1+ specifically, we derive these *unique* identities from a *unique* variable $U.\mathsf{uName}$ that is maintained by the user and sent to the token. In practice, uName could be a unique user identifier the user knows (such as an email used in all authentication flows with relying parties involving the user's token), or a unique token name chosen and kept by the user. Then, each party derives unique id_U and id_T by appending a client-side bit 0 or a token-side bit 1 to uName, respectively. Note that $U.\mathsf{uName}$ is not sent in encrypted form, since id_U and id_T are assumed to be public. The only requirement is that it has to be unique per user.

The session identifier sid, which is maintained by both client and token sessions, still corresponds to the entire trace of Bind (which is now different because the messages being exchanged are part of the PAKE protocol).

We also add a new $\mathsf{bindError}$ state of execution to client sessions (that is now defined as $\pi^j_C.\mathsf{st}_{\mathsf{exe}} \in \{\perp, \mathsf{waiting}, \mathsf{bindStart}, \mathsf{bindError}, \mathsf{bindDone}\}$), which is set to true if the client fails to validate the final message sent by the token during Bind. This is specific to sCTAP 2.1+, since other mPACA protocols like CTAP 2.1+ do not validate this last token-to-client message.

6.4.2. High-level flow and syntax of sCTAP 2.1+

We now present the high-level flow description of sCTAP 2.1+, seen in Figure 6.3, with the corresponding function definitions in Figure 6.4. However, first note that, in sCTAP 2.1+, setting a pin still requires establishing a shared secret through the traditional ECDH key

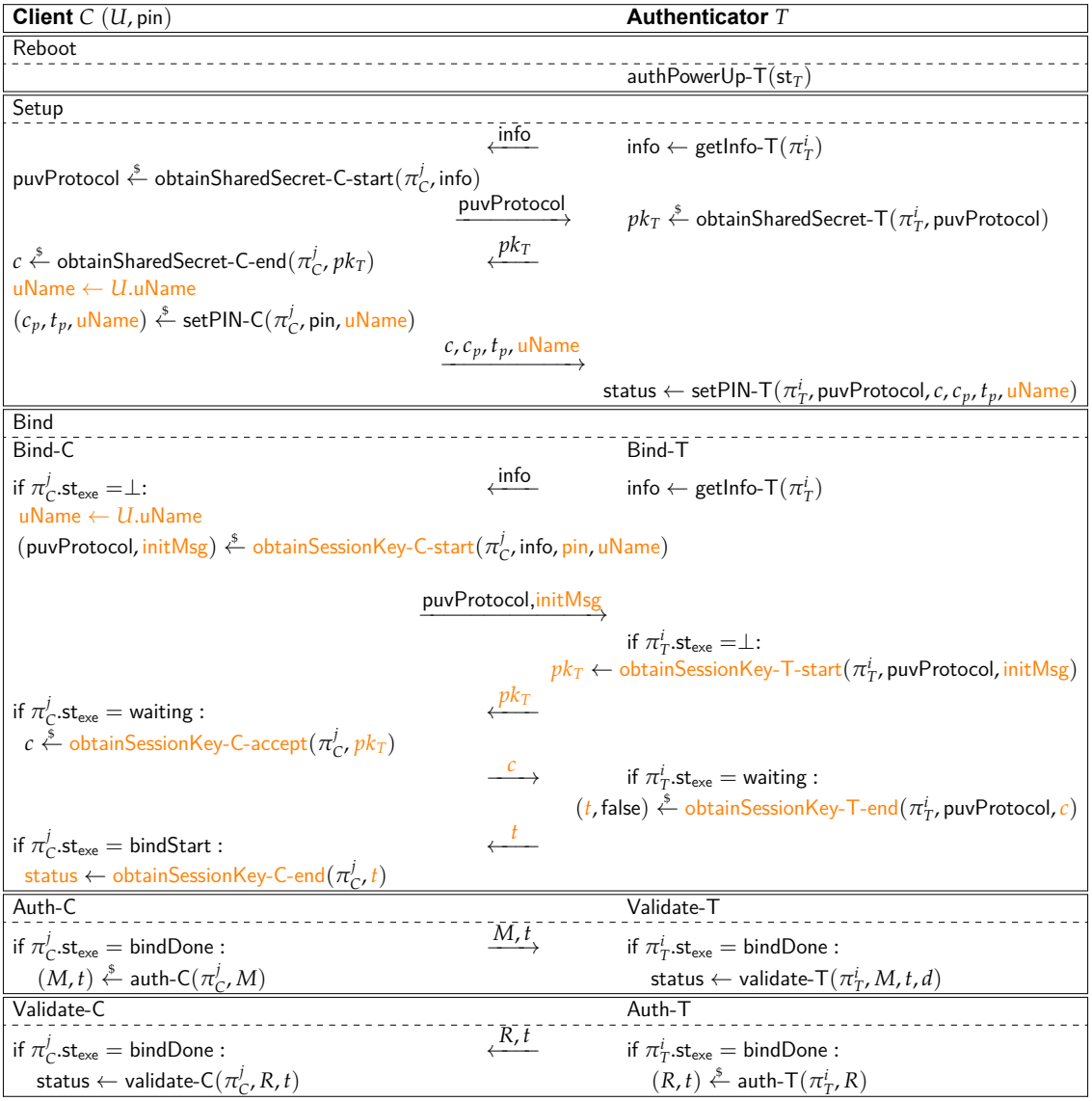


Figure 6.3: sCTAP 2.1+ as an mPACA protocol. The diagram follows the same structure as in Figure 5.3, with differences from CTAP 2.1+ marked in orange.

exchange. This is because using a PAKE to establish an authenticated channel between a client and a token requires both parties to be in possession of a shared PAKE password, which in sCTAP 2.1+ is the pin. For a token, this only happens *after* Setup.

The Reboot subprotocol calls the authPowerUp-T function, which inputs the token's internal state st_T, resets the consecutive pin attempts counter $\text{st}_T.m$, and runs initialize-T on every supported PIN/UV Auth Protocol instantiation, which resets its internal state.

The Setup subprotocol starts with token T sending its info, which contains the supported PIN/UV Auth Protocol instantiations, to client C. When C receives this information, it runs obtainSharedSecret-C-start, where it chooses one PIN/UV Auth Protocol instantiation, generates a fresh ECDH key pair, and sends its choice back to token T. The token then runs

obtainSharedSecret-T, which fetches its previously generated ECDH public key pk_T, and sends it back to C. Upon receiving the token's public key, C first runs obtainSharedSecret-C-end, which executes the PIN/UV Auth Protocol encapsulate function on the token's public key to output a shared secret K and its own ECDH public key c, and then runs setPIN-C. This function inputs the user's pin and the unique identifier uName, encrypts the pin into a ciphertext c_p using K, and then authenticates the concatenation of c_p and uName to produce a tag t_p. The client public key c, ciphertext c_p, tag t_p and identifier uName are then sent to the token, which runs decapsulate on c to obtain the shared secret K, verifies tag t_p against the concatenation of c_p and uName and decrypts c_p to obtain the pin. Then, the token derives the two identifiers id_U and id_T from concatenating 0 or 1 to uName, respectively, stores the pin, id_U and id_T in its static storage st_T and sets its retry counter to the value of pinRetriesMax. Note that, after Setup, the token can always use these stored identifiers to perform subsequent Bind procedures with a client.

Bind starts with the client receiving the token info and running obtainSessionKey-C-start. This function inputs the user pin, info and the unique identifier uName, derives id_U and id_T by appending 0 or 1 to uName, and runs initialize-C from its selected PIN/UV Auth Protocol instantiation puvProtocol (described next). This function inputs the pin, id_U and id_T and outputs the first PAKE message initMsg (the contents of which are protocol-specific). This message is appended to the sCTAP 2.1+ client session sid and is sent to token T, along with the selected puvProtocol. T runs obtainSessionKey-T-start, where it immediately calls regenerate with initMsg, pin, id_U and id_T as parameters to produce the token public key pk_T (which is not a simple ECDH public key, but rather a PAKE-specific public key that may contain additional data). This public key pk_T is then appended, alongside initMsg, to the token session sid and sent back to C.

At this point, C executes obtainSessionKey-C-accept, which runs encapsulate on the received PAKE public key pk_T and outputs c. This output is part of the PAKE protocol, and because we impose that any PAKE protocol being used in sCTAP 2.1+ must provide *explicit* mutual authentication, c additionally contains a tag, with the tag's generation also being protocol-specific. The client then sets the state of execution $\pi_C^j.\mathsf{st}_{\mathsf{exe}}$ to bindStart, appends c to the sid, and sends it to T. Upon receiving c, the T executes obtainSessionKey-T-end, which immediately decrements the pinRetries counter and runs decapsulate on c. This operation produces the session key K and a key confirmation tag t, provided the pin that was used in obtainSessionKey-T-start was correct and c was also produced using the same pin (input in the first step). Otherwise, the verification fails, which is why the pinRetries counter is

getInfo-T(π_T^i):
1: info $\leftarrow$ (st$_T$.version, st$_T$.puvProtocolList)
2: π_T^i.sid $\leftarrow$ π_T^i.sid || info

setPIN-C(π_C^j, pin, uName):
1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: $c_p \xleftarrow{\$} \pi_C^j$.puvProtocol.encrypt($\pi_C^j$.K, pin)
3: $m \leftarrow c_p$ || uName
4: $t_p \xleftarrow{\$} \pi_C^j$.puvProtocol.authenticate(π_C^j.K, m)
5: **return** (c_p, t_p, uName)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p, uName):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pin $\neq \perp$ **then**
2: **return** $\perp$
3: $K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
4: $m \leftarrow c_p$ || uName
5: **if** $K = \perp \vee$ st$_T$.puvProtocol.verify(K, m, t_p) = false **then**
6: **return** $\perp$
7: pin $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_p)
8: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
9: st$_T$.pin $\leftarrow$ pin
10: st$_T$.id$_U$ = uName || 0
11: st$_T$.id$_T$ = uName || 1
12: st$_T$.pinRetries $\leftarrow$ pinRetriesMax
13: **return** accepted

obtainSessionKey-C-start(π_C^j, info, pin, uName):
1: Parse(version, puvProtocolList) $\leftarrow$ info
2: **if** version = 2.0 **then return** $\perp$
3: select puvProtocol $\leftarrow$ puvProtocolList
4: π_C^j.puvProtocol $\leftarrow$ puvProtocol
5: id$_U$ = uName || 0
6: id$_T$ = uName || 1
7: initMsg $\xleftarrow{\$}$ π_C^j.puvProtocol.initialize-C(pin, id$_U$, id$_T$)
8: π_C^j.st$_{exe}$ $\leftarrow$ waiting
9: π_C^j.sid $\leftarrow$ π_C^j.sid || info || puvProtocol || initMsg
10: **return** (puvProtocol, initMsg)

obtainSessionKey-C-end(π_C^j, t):
1: **if** π_C^j.st$_{exe}$ = bindError **then**
2: **return** $\perp$
3: (K, valid) $\leftarrow$ π_C^j.puvProtocol.finalizeBind(t)
4: **if** valid = false **then**
5: π_C^j.st$_{exe}$ $\leftarrow$ bindError
6: **return** $\perp$
7: π_C^j.bs $\leftarrow$ K
8: π_C^j.st$_{exe}$ $\leftarrow$ bindDone
9: π_C^j.sid $\leftarrow$ π_C^j.sid || t || false
10: **return** accepted

authPowerUp-T(st$_T$):
1: **if** st$_T$.version $= \perp \wedge$ st$_T$.puvProtocolList $= \perp$ **then**
2: (version, puvProtocolList) $\leftarrow$ st$_T$.initialData
3: st$_T$.version $\leftarrow$ version
4: st$_T$.puvProtocolList $\leftarrow$ puvProtocolList
5: **for all** puvProtocol $\in$ st$_T$.puvProtocolList **do**
6: st$_T$.puvProtocol.initialize-T()
7: st$_T$.m $\leftarrow$ 3

obtainSessionKey-T-start(π_T^i, puvProtocol, initMsg):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList **then return** $\perp$
2: st$_T$.puvProtocol.regenerate(initMsg, st$_T$.pin, st$_T$.id$_U$, st$_T$.id$_T$)
3: $pk_T \leftarrow$ st$_T$.puvProtocol.getPublicKey()
4: π_T^i.st$_{exe}$ $\leftarrow$ waiting
5: π_T^i.sid $\leftarrow$ π_T^i.sid || puvProtocol || initMsg || pk_T
6: **return** pk_T

obtainSessionKey-C-accept(π_C^j, pk_T):
1: $c \leftarrow \pi_C^j$.puvProtocol.encapsulate(pk_T)
2: π_C^j.st$_{exe}$ $\leftarrow$ bindStart
3: π_C^j.sid $\leftarrow$ π_C^j.sid || pk_T || puvProtocol || c
4: **return** c

obtainSessionKey-T-end(π_T^i, puvProtocol, c):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries = 0 **then**
2: **return** ($\perp$, false)
3: st$_T$.pinRetries $\leftarrow$ pinRetries $-$ 1
4: (K, t) $\leftarrow$ st$_T$.puvProtocol.decapsulate(c)
5: ▷ *Pin is invalid if* (K, t) $= \perp$, *since this means the validation of the PAKE tag in* c *was unsuccessful*
6: **if** (K, t) $= \perp$ **then**
7: st$_T$.puvProtocol.initialize-T()
8: st$_T$.m $\leftarrow$ st$_T$.m $-$ 1
9: **if** st$_T$.m = 0 **then**
10: authPowerUp-T(st$_T$)
11: **return** ($\perp$, true)
12: **else**
13: **return** ($\perp$, false)
14: st$_T$.m $\leftarrow$ 3
15: st$_T$.pinRetries $\leftarrow$ pinRetriesMax
16: **for all** puvProtocol' $\in$ st$_T$.puvProtocolList **do**
17: st$_T$.puvProtocol'.resetPuvToken()
18: π_T^i.bs $\leftarrow$ st$_T$.puvProtocol.pt
19: π_T^i.st$_{exe}$ $\leftarrow$ bindDone
20: π_T^i.canValidate $\leftarrow$ true
21: π_T^i.sid $\leftarrow$ π_T^i.sid || puvProtocol || c || t || false
22: **return** (t, false)

Figure 6.4: sCTAP 2.1+ protocol functions. Changes from CTAP 2.1+ are marked in orange. New functions are marked in blue. Any function that is not presented here remains unchanged from CTAP 2.1+.

decremented *before* decapsulate runs, and decapsulate returns K and t as $\perp$. If the process is successful, the session key is set as the binding state, c and the key confirmation tag t are appended to the sid, and the latter is sent to C. Finally, upon receiving t, C executes obtainSessionKey-C-end, which runs finalizeBind, a new PIN/UV Auth Protocol function we introduce that inputs the key confirmation tag t and outputs the session key K and a boolean flag indicating the success or failure of the operation. If the tag verification fails, then the client session sets the state of execution to bindError, and terminates with $\perp$. Otherwise, it sets the binding state as the session key K, appends t to the sid, and returns accepted.

PIN/UV Auth Protocol syntax Notice that the high-level syntax we present here keeps the logic related with dealing with multiple PIN/UV Auth Protocol instantiations, like the resetting of the pinUvAuthToken in obtainSessionKey-T-end and the initialization of each instantiation in authPowerUp-T. In reality, our sCTAP 2.1+ high-level syntax is slightly adapted to use our new instantiation of PIN/UV Auth Protocol, which we introduce next, and therefore in this description we assume only this new instantiation is used during Bind. This is due to the necessary changes we applied to the protocol's syntax that could not be applied in the lower-level functionality of the PIN/UV Auth Protocol, such as the regeneration of a new public key on the token before sending it to the client and the decrement of the pinRetries counter *before* running decapsulate (since the pin verification is performed on decapsulate now).

6.4.3. Low-level syntax of sCTAP 2.1+

We now describe our new proposed PIN/UV Auth Protocol instantiation, which we call PIN/UV Auth Protocol 3. Note that adding a PAKE to the established CTAP2 flow requires deep changes in the lower level, cryptographic core of the protocol, and as a result our proposed instantiation of the PIN/UV Auth Protocol significantly differs from PIN/UV Auth Protocol 1 and PIN/UV Auth Protocol 2. Nevertheless, we attempted to adapt our modifications to the existing PIN/UV Auth Protocol interface as much as possible, which is why we maintain function names such as encapsulate, decapsulate or resetPuvToken unchanged.

As mentioned above, we assume this instantiation to be used *only* in Bind, and that the client and the token negotiate one of the other two instantiations to perform Setup.

initialize-C $(\mathsf{pin}, \mathsf{id}_U, \mathsf{id}_T)$:
1: $\mathsf{state}_C \leftarrow \perp$
2: ▷ *state_C contains the PAKE internal state, including the secret key*
3: $\mathsf{state}_C, \mathsf{msg}_1 \xleftarrow{s} \mathsf{PAKE.initStart}(\mathsf{pin}, \mathsf{id}_U, \mathsf{id}_T)$
4: **return** msg_1

expand (pt):
1: $pt_e \leftarrow \mathcal{H}_3(pt)$
2: $\mathsf{Parse}(K_C, K_T) \leftarrow pt_e$, s.t. $|K_C| = 2\lambda$
3: **return** (K_C, K_T)

finalizeBind (msg_4):
1: $(K_C, \mathsf{status}) \leftarrow \mathsf{PAKE.initEnd}(\mathsf{msg}_4, \mathsf{state}_C)$
2: **return** (K_C, status)

encapsulate (msg_2):
1: $\mathsf{msg}_3, \mathsf{state}_C \leftarrow \mathsf{PAKE.initAccept}(\mathsf{msg}_2, \mathsf{state}_C)$
2: ▷ *msg_3 contains the PAKE ciphertext and tag*
3: **return** msg_3

authenticate (K', m):
1: $\mathsf{Parse}(K'_1, K'_2) \leftarrow K'$, s.t. $|K'_1| = 2\lambda$
2: $t \leftarrow \mathsf{MAC}(K'_1, m)$
3: **return** t

initialize-T ():
1: $pt \leftarrow \perp, \mathsf{state}_T \leftarrow \perp$

getPublicKey ():
1: **return** pk_T

resetPuvToken ():
1: $pt \leftarrow K_T$

regenerate $(\mathsf{msg}_1, \mathsf{pin}, \mathsf{id}_U, \mathsf{id}_T)$:
1: ▷ *state_T contains the PAKE internal state, including the secret key*
2: $\mathsf{state}_T, \mathsf{msg}_2 \xleftarrow{s} \mathsf{PAKE.respStart}(\mathsf{msg}_1, \mathsf{pin}, \mathsf{id}_U, \mathsf{id}_T)$
3: ▷ *msg_2 contains the PAKE public key and other protocol-specific data*
4: $pk_T \leftarrow \mathsf{msg}_2$
5: **return**

decapsulate (msg_3):
1: $(K_T, \mathsf{msg}_4) \leftarrow \mathsf{PAKE.respEnd}(\mathsf{state}_T, \mathsf{msg}_3)$
2: ▷ *Pair (K_T, msg_4) can be $(\perp, \perp)$ if PAKE.respEnd fails msg_4 contains the response tag from the token to the client (key confirmation step for explicit mutual authentication)*
3: **return** (K_T, msg_4)

verify (K', m, t):
1: $\mathsf{Parse}(K'_1, K'_2) \leftarrow K'$, s.t. $|K'_1| = 2\lambda$
2: $t \leftarrow \mathsf{MAC}(K'_1, m)$
3: **return** $[[t = t']]$

Figure 6.5: The sCTAP 2.1+ PIN/UV Auth Protocol 3 (only used in Bind). PAKE refers to a generic PAKE construction. Each PAKE function follows the syntax introduced in Section 2.4 of Chapter 2. All changes from the PIN/UV Auth Protocol instantiation from CTAP 2.1+ are marked in blue.

The initialize-C and initialize-T functions are used by the client and the token to initialize their internal PAKE states state_C and state_T, respectively. These states are protocol-specific, but contain at the very least the pin and party identifiers id_U and id_T. With initialize-C, executed at the beginning of Bind, the client also runs PAKE.initStart, which is the first step that is executed by the PAKE initiator. This function inputs the pin and party identifiers and outputs the first PAKE message msg_1 and new state_C. With initialize-T, the token also erases the previous pinUvAuthToken pt (we maintain this naming even though there is no longer a pinUvAuthToken in this instantiation of the PIN/UV Auth Protocol). This function is executed on Reboot, or during Bind in obtainSessionKey-T-end.

The regenerate function runs PAKE.respStart, which inputs the pin and the two party identifiers, as well as the first message msg_1 from initialize-C (which corresponds to initMsg in Figure 6.3), and outputs the updated state state_T and the second PAKE message msg_2, which contains the PAKE public key and other specific data. The latter is set as the internal public key pk_T that will be returned by getPublicKey.

The encapsulate function runs PAKE.initAccept function, which inputs state_C and msg_2 and

outputs an updated state_C and the third message msg_3. This message contains the PAKE ciphertext/public key (depending on the concrete type of PAKE being considered) and a key confirmation tag. In decapsulate, the token runs PAKE.respEnd, which inputs state_T and msg_3 and outputs the PAKE session key K_T and the fourth message msg_4, containing the key confirmation tag to be sent to the client.

Finally, we introduce a new finalizeBind function in PIN/UV Auth Protocol 3, which is necessary for our explicit mutual authentication goal. This function runs the fifth and final function PAKE.initEnd, which inputs the state state_C and fourth message msg_4 and outputs the session key K_C and a flag indicating the success or failure of the procedure. This is the step where key confirmation occurs on the client side, and the procedure is successful only if the validation of the authentication tag in msg_4 succeeds.

The PAKE syntax Recall that the PAKE syntax introduced in Chapter 2 fixes a concrete set of functions and message flow. This is so that any such PAKE can be included in sCTAP 2.1+ without changing the message flow of CTAP 2.1+, as can be seen in PIN/UV Auth Protocol 3. The initialization of the client produces the first PAKE message, which is sent to the token with the puvProtocol. Then, the token generates its PAKE public key in the same stage as a token in CTAP 2.1+ would fetch its own ECDH public key. Because of this, the rest of the protocol execution proceeds with the same message flow, where the concrete messages exchanged are different, but the stages of execution on which they are sent or received are the same.

6.4.4. Differences to CTAP 2.1+

We now highlight the main differences between CTAP 2.1+ and sCTAP 2.1+, which can be seen in Figures 6.3 and 6.4.

High-level flow

The first visible difference is related to the generation and transmission of the party identifiers during Setup. These are generated by the client, authenticated alongside the encrypted pin, and sent to the token, where they are stored for future Bind runs. Note that these identifiers are public, since they are not encrypted with the pin.

The Bind procedure is significantly different from CTAP 2.1+. One of our core goals with sCTAP 2.1+ was to maintain the message flow of CTAP 2.1/CTAP 2.1+ unchanged, in order to minimize the required modifications to adapt the protocol for a PAKE. This can be seen in the diagram from Figure 6.3, where differences to Figure 5.3 are related to

the concrete procedures executed on the client and the token, and on the content of the exchanged messages, but crucially not on the number of messages nor on when they are sent. Most of the core modifications are related to the lower-level PIN/UV Auth Protocol functions that implement the PAKE logic, which we described above, but there are some core modifications to the high-level protocol syntax that are worth pointing out.

First and foremost, regarding the procedures executed by each party, we introduce a new obtainSessionKey prefix for all functions involved in Bind, which replaces the obtainSharedSecret and obtainPinUvAuthToken prefixes (although the obtainSharedSecret functions still exist on Setup). This is a syntax change to clarify what these functions are now doing, since the Bind procedure on sCTAP 2.1+ no longer uses pinUvAuthTokens and shared secrets derived from an ECDH key exchange. Instead, the establishment of an authenticated channel happens through the derivation of a shared PAKE session key, which each party sets as its binding state. Internally, however, the obtainSessionKey functions are quite similar to their counterparts from CTAP 2.1+, and so this modification is mainly to clarify the role each function plays in this protocol. Note, however, that there was one function that was completely removed from sCTAP 2.1+: obtainPinUvAuthToken-C-start. This was the function that encrypted the pinHash, set the state of execution to bindStart, and returned the ciphertext. In sCTAP 2.1+, this step no longer exists, because the pin is never sent to the token. Therefore, we merged the logic of obtainSharedSecret-C-end and obtainPinUvAuthToken-C-start into obtainSessionKey-C-accept.

The content of the exchanged messages also underwent profound changes with sCTAP 2.1+. First, the client now sends an initial message initMsg to the token, which corresponds to the first PAKE message. Second, the token public key pk_T is different from the traditional ECDH public key, since it is PAKE-specific and, consequently, *may* have a very different structure than the public keys used in CTAP 2.1+. Third, the message c sent by the client to the token is no longer just the client public key that is output by encapsulate, but a PAKE-specific public key (or ciphertext, if the PAKE is KEM-based) and a key confirmation tag. Also, as previously mentioned, ciphertext c_{ph} is no longer sent to the token. Finally, the last message sent by the token to the client in sCTAP 2.1+ is the second key confirmation tag, instead of the encrypted pinUvAuthToken (c_{pt}).

Low-level flow

As mentioned, PIN/UV Auth Protocol 3 differs significantly from the other instantiations, due to the nature of the protocol changes we propose. Besides the obvious modifications

in the implementation of core functions, which were detailed above and can be seen in Figure 6.5, one of the most notable changes is the absence of the encrypt and decrypt functions, which are no longer necessary because the pin (or more concretely the pinHash) and the pinUvAuthToken are no longer transmitted between the client and the token. This is a core change to the protocol, which completely removes the need to run AES-256 on either party.

Another notable difference is the introduction of the finalizeBind function, which as previously mentioned is the last function run by the client during Bind and verifies the key confirmation tag from the token to provide explicit mutual authentication[4].

Besides the addition or removal of functions, we also split the PIN/UV Auth Protocol initialize function into initialize-C and initialize-T, since they play different roles in PIN/UV Auth Protocol 3. However, note that this separation in names is simply for clarifying which party runs which implementation, and that in practice the function name initialize would still be maintained on a real implementation.

Finally, we also modified some existing functions with respect to the parameters they receive. Concretely, the regenerate and initialize-C functions, which do not receive parameters in the PIN/UV Auth Protocol interface, have been changed in PIN/UV Auth Protocol 3 to receive four and three parameters, respectively. This was necessary due to the type of PAKE protocols we consider for sCTAP 2.1+, where the client acts as the initiator, and must already have the pin and party identifiers to produce the first PAKE message msg_1, and the token must have msg_1, the pin and the party identifiers to produce a PAKE public key pk_T.

6.5. Authentication security of sCTAP 2.1+

We now state our authentication security results for sCTAP 2.1+, where we show that it provides resistance against man-in-the-middle attacks by proving that it achieves SUF security in our updated mPACA security model.

We assume that sCTAP 2.1+ uses PIN/UV Auth Protocol 2 during Setup, and thus all assumptions from CTAP 2.1+ regarding the random oracle $\mathcal{H}_2$, the encryption scheme SKE and the ECDH group also apply here, although they are relevant only for Setup queries.

[4]Note that, in an effort to preserve the interface as much as possible, one could maintain the encrypt and decrypt functions, where the former would be unused and the latter would perform the operations of finalizeBind (since both are executed at the same stage of Bind). This would maintain the current PIN/UV Auth Protocol interface function names from CTAP2 but would, in our view, make our instantiation more confusing.

The assumptions held for CTAP 2.1+ on the pin distribution, random oracle $\mathcal{H}_3$ and MAC also hold for sCTAP 2.1+. Additionally, we assume the underlying PAKE construction provides *explicit mutual authentication* (ma), *perfect forward secrecy* (pfs) and *sid collision resistance* (sid-coll), as defined in Section 2.4 from Chapter 2.

Users with multiple tokens Recall from the model description in Chapter 5 that a user U in our model is simply a holder of a single PIN, and that all tokens setup by U share the same PIN. The token identifier id_T, which is necessary for our proof, must be unique for every independent PAKE password used in the security game (the PAKE challenger samples a random independent password for each pair of parties). Therefore, in our model, every token setup by a user U has the same PIN *and* identifier id_T, which is derived by the unique uName that belongs to U.

The following theorem states the SUF security of sCTAP 2.1+:

Theorem 6.1. *For every efficient adversary $\mathcal{A}$ that makes at most q_S, q_E, q_{SendC} and q_{SendT} queries to Setup, Execute, Send-Bind-C and Send-Bind-T, and at most $q_{\mathcal{H}_3}$ queries to $\mathcal{H}_3$, there exist efficient adversaries $\mathcal{B}_1$, $\mathcal{B}_3$, $\mathcal{B}_4$, $\mathcal{B}_5$, $\mathcal{B}_6$, $\mathcal{B}_8$ and $\mathcal{B}_9$ such that:*

$$\begin{aligned}\mathit{Adv}^{\mathit{SUF}}_{\mathit{sCTAP\ 2.1+}}(\mathcal{A}) \leq\ & q_S\ \mathit{Adv}^{\mathit{sCDH}}_{\mathit{ECDH}}(\mathcal{B}_1) + q_S\ \mathit{Adv}^{\mathit{IND\text{-}1CPA}}_{\mathit{SKE}}(\mathcal{B}_3) \\ & + \mathit{Adv}^{\mathit{ma}}_{\mathit{PAKE}}(\mathcal{B}_4) + \mathit{Adv}^{\mathit{pfs}}_{\mathit{PAKE}}(\mathcal{B}_5) + \mathit{Adv}^{\mathit{sid\text{-}coll}}_{\mathit{PAKE}}(\mathcal{B}_6) + \frac{q_{\mathcal{H}_3}\ (q_E + q_{\mathit{Send\text{-}T}})}{2^{2\lambda}} \\ & + (q_E + q_{\mathit{Send\text{-}T}})\ \mathit{Adv}^{\mathit{SUF\text{-}CMA}}_{\mathit{MAC}}(\mathcal{B}_8) + (q_E + q_{\mathit{Send\text{-}C}})\ \mathit{Adv}^{\mathit{SUF\text{-}CMA}}_{\mathit{MAC}}(\mathcal{B}_9)\ .\end{aligned}$$

We now present a proof sketch for Theorem 6.1, and refer to Appendix F for the complete, pseudocode proof.

Proof. (Sketch.) Let Pr_i denote the probability that Game i outputs 1.

Game 0 This is the original experiment. Therefore,

$$\mathrm{Pr}_0 = \mathrm{Adv}^{\mathrm{SUF}}_{\mathrm{sCTAP\ 2.1+}}(\mathcal{A})\ .$$

Game 1 In this game, we replace every symmetric session key K_1 and K_2 derived in encapsulate and decapsulate, which now only happens on Setup queries, with independent random values $\tilde{K_1}$ and $\tilde{K_2}$. We then define bad as the event that, at some point during

the game, π_C^j and π_T^i are involved in a Setup query and $\mathcal{A}$ queries $\mathcal{H}_2$ with Z, and bound the probability of bad by constructing an adversary $\mathcal{B}_1$ against the sCDH security of the underlying ECDH group, such that if $\mathcal{A}$ causes bad, $\mathcal{B}_1$ wins the sCDH game. Because the ECDH key exchange only occurs during Setup queries (which are all passive), we have $|\mathsf{Pr}_0 - \mathsf{Pr}_1| \leq q_S \, \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1)$.

Game 2 In this game, in every Setup query, the pin is no longer stored in the token state st_T, being stored instead in a new strucuture accessible only by the challenger. Since this model never allows $\mathcal{A}$ to corrupt any token, Games 1 and 2 are functionally identical. Therefore, $\mathsf{Pr}_1 = \mathsf{Pr}_2$.

Game 3 This game replaces every encrypted pin in Setup queries with an encrypted constant pin 0000. We bound the advantage of $\mathcal{A}$ against Game 3 by reduction to the IND-1CPA security of the underlying encryption scheme SKE. Therefore, we can construct an adversary $\mathcal{B}_3$ against the IND-1CPA security of SKE such that $|\mathsf{Pr}_2 - \mathsf{Pr}_3| \leq q_S \, \mathsf{Adv}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{B}_3)$.

Game 4 In this game, $\mathcal{A}$ loses if it breaks the *explicit mutual authentication* (ma) guarantee of the underlying PAKE protocol, by making any token or client session finish Bind without having a unique PAKE partner. We define this bad event bad, and bound the advantage of $\mathcal{A}$ against Game 4 by reduction to the ma security of PAKE by constructing an adversary $\mathcal{B}_4$ such that if $\mathcal{A}$ causes bad, $\mathcal{B}_4$ wins the ma game. Therefore, we have that $|\mathsf{Pr}_3 - \mathsf{Pr}_4| \leq \mathsf{Adv}^{\mathsf{ma}}_{\mathsf{PAKE}}(\mathcal{B}_4)$.

Game 5 In this game, unless a token or client session's pin was corrupted *and* $\mathcal{A}$ engaged with that session using Send-Bind-T or Send-Bind-C, every session key K is replaced with a new uniformly random key $\tilde{K}$. Recall that since Game 4 $\mathcal{A}$ can only actively attack a session by corrupting the pin, otherwise the ma security of PAKE is broken and the game aborts. We then bound the advantage of $\mathcal{A}$ against Game 5 by reduction to the *perfect forward secrecy* (pfs) security of the underlying PAKE. Therefore, we can construct an adversary $\mathcal{B}_5$ against the pfs security of PAKE such that $|\mathsf{Pr}_4 - \mathsf{Pr}_5| \leq \mathsf{Adv}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{B}_5)$.

Game 6 In this game, $\mathcal{A}$ loses if it breaks the *sid collision resistance* (sid-coll) guarantee of the underlying PAKE, by making any two token or client sessions finish Bind with the same PAKE sid. We define this bad event bad, and bound the advantage of $\mathcal{A}$ against Game 6 by reduction to the sid-coll security of PAKE by constructing an adversary $\mathcal{B}_6$ such

that if $\mathcal{A}$ causes bad, $\mathcal{B}_6$ wins the sid-coll game. Therefore, we have that $|\mathsf{Pr}_5 - \mathsf{Pr}_6| \leq \mathsf{Adv}^{\text{sid-coll}}_{\mathsf{PAKE}}(\mathcal{B}_6)$.

Game 7 In this game, $\mathcal{A}$ loses if it queries random oracle $\mathcal{H}_3$ with any PAKE session key K generated throughout the experiment and set as the binding state of a token or client session, but for which it has no information. These session keys correspond to sessions where the adversary behaved passively, which includes all of the Execute queries, and possibly some of the Send-Bind-T queries (passive Send-Bind-C queries recover the key generated in Send-Bind-T queries). The two games are identical until bad, and so we have $|\mathsf{Pr}_6 - \mathsf{Pr}_7| \leq q_{\mathcal{H}_3}\ (q_{\mathsf{E}} + q_{\text{Send-T}})/2^{2\lambda}$. After this game, the MAC keys used by these sessions to authenticate commands are information theoretically hidden from the adversary.

Game 8 In this game, when $\mathcal{A}$ queries Validate-T with message M and a tag t on a token session π^i_T that finished Bind passively (which happens in Execute queries and *can* happen in Send-Bind-T queries), such that (M, t) would constitute a valid forgery, the challenger still rejects (recall that $\mathcal{A}$ cannot win against any token session π^i_T that was actively attacked in Send-Bind-T queries, since after Game 7 this means T's pin must have been corrupted). We consider this bad event and bound the advantage of $\mathcal{A}$ by reducing to the SUF-CMA security of the underlying MAC, by constructing an efficient adversary $\mathcal{B}_8$ against the SUF-CMA security of MAC such that if bad happens, $\mathcal{B}_8$ wins the SUF-CMA game. Therefore, we have $|\mathsf{Pr}_7 - \mathsf{Pr}_8| \leq (q_{\mathsf{E}} + q_{\text{Send-T}})\ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_8)$.

Game 9 In this game, when $\mathcal{A}$ queries Validate-C with message M and a tag t and a client session π^j_C that finished Bind passively (which happens in Execute queries and *can* happen in Send-Bind-C queries), such that (M, t) would constitute a valid forgery, the challenger still rejects. Much like in Game 8, we bound the advantage of $\mathcal{A}$ against Game 9 via reduction to the SUF-CMA security of MAC. Therefore, there exists an efficient adversary $\mathcal{B}_9$ such that $|\mathsf{Pr}_8 - \mathsf{Pr}_9| \leq (q_{\mathsf{E}} + q_{\text{Send-C}})\ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_9)$.

Final analysis After the modifications in Game 9, $\mathcal{A}$ can no longer win. Just as in the SUF-t experiment for CTAP 2.1+, for $\mathcal{A}$ to win it must satisfy at least one of the four conditions in Token-Win-SUF, or one of the three conditions in Client-Win-SUF. The first condition of Token-Win-SUF is always false, since $d = 1$ must always be true for any Validate-T query to accept a tag t. The second and third conditions of Token-Win-SUF and first and second conditions of Client-Win-SUF are identical, and are also always false. Indeed, collisions

between sCTAP 2.1+ sid values in client and token sessions have been ruled out in Game 6 due to ruling out collisions between PAKE sid values, which are part of the sCTAP 2.1+ sid. Therefore, it can never be the case that two different client or token sessions have the same sid. $\mathcal{A}$ can also never win in the fourth condition of Token-Win-SUF or the third condition of Client-Win-SUF, because to do so it must be able to provide a valid forgery to a token session via Validate-T, which was ruled out in Game 8, or to a client session via Validate-C, which was ruled out in Game 9. Therefore, it is always true that $\mathrm{Pr}_9 = 0$. Note that the running times of all adversaries are close to that of $\mathcal{A}$. □

6.6. Instantiating the PAKE protocol in sCTAP 2.1+

So far, we have described how a generic PAKE protocol may be used to replace unauthenticated ECDH key exchanges during Bind, by fixing its syntax and security properties, and showing how it can be integrated into CTAP2's structure and flow. However, we have not yet discussed what specific protocols may be used for this purpose, nor how their inclusion in CTAP2 may impact its performance and implementation cost. We briefly reflect on the former here, and leave the performance analysis and implementation details for Chapter 7.

6.6.1. Classical PAKE protocols

Barbosa et al. [1] suggested that the PAKE protocol in sPACA may be instantiated with CPace [46, 49, 50] or SPAKE2 [47], and analysed the expected performance of sPACA vs CTAP 2.0 when instantiating the former with CPace.

At its core, CPace is a 2-message flow PAKE protocol that uses ECDH over a group $\mathcal{G}$ to generate public keys based on a string derived from the shared password and other data. This string is mapped to a generator G, with each public key calculated as G^x. Because G is unknown to anyone that does not know the pre-agreed password, an adversary cannot perform a man-in-the-middle attack because it cannot produce a new public key that is accepted by either PAKE party.

While the basic 2-message flow does not provide explicit mutual authentication, it is pointed out in the specification [50] that, if required by the higher-level protocol, additional message flows may be included to achieve this goal (which is the case with sCTAP 2.1+). Additionally, it requires an agreed upon subsession identifier ssid (not to be confused with the PAKE sid) between the two parties, which is used to generate G and, therefore, each

party's public key. This is due to CPace having its security analysis in the Universal Composability (UC) framework, and its role is essentially to ensure that multiple concurrent CPace executions remain secure even when instantiated as a part of a higher level protocol. The pre-agreement of ssid between the two parties may require more messages being exchanged, especially if this pre-agreement requires each party to generate part of the ssid and then combine both parts to generate the full ssid[5]. However, when using CPace on sCTAP 2.1+, this is not a problem, because we already expect a 4-message flow between a client and a token.

As previously mentioned, we prioritised maintaining the message flow of sCTAP 2.1+ identical to CTAP2 with the goal to reduce the complexity of implementing our proposed modifications. Therefore, the first message of *any* PAKE protocol, output by initStart, is sent in the first command the client sends to the token when requesting a key exchange (authenticatorClientPIN with the getKeyAgreement subcommand), and the last (fourth) message output by respEnd is sent by the token to the client as a response to the getPinToken subcommand, with the final client side function initEnd outputing the session key K. When instantiating the underlying PAKE with CPace, we can have initMsg be the first half of the ssid that the client sends to the token, and then have the token send the other half of the ssid to the client along with its newly generated public key. At this point, both parties have the full ssid, and the client has all it needs to produce its public key and authentication tag (which we call c in the diagram from Figure 6.3) and send it to the token, which derives the session key and responds with an authentication tag that it sends to the client. In CPace specifically, the authentication tags are calculated by hashing the concatenation of ssid with the session key K.

6.6.2. Post-Quantum PAKE protocols

While instantiating the PAKE protocol in sCTAP 2.1+ with CPace is a viable option, both in the expected impact on performance and on providing resistance to active attackers during Bind, it is still based on the computational hardness of the Diffie-Hellman problem (more concretely, on the Simultaneous Diffie-Hellman (sSDH) and strong Computational Diffie-Hellman (sCDH) assumptions [50]), and thus not resilient against quantum adversaries.

[5]The CPace specification [50] recommends this approach, but mentions that if it is not a viable option, then at the very least the initiator should generate a fresh random ssid and send it to the responder.

Recently, some new candidate protocols [21, 27, 51–53] that are based on Key Encapsulation Mechanisms (KEM) have been introduced that *may*[6] offer some resistance against quantum adversaries when instantiated with a post-quantum KEM like ML-KEM [24], which is the scheme derived from CRYSTALS-Kyber [22, 23] that was standardized by the National Institute of Standards and Technology (NIST).

We believe it is worth considering this class of KEM-based protocols. We already propose replacing the ECDH key exchange in sCTAP 2.1+ with a PAKE, introducing an entirely different protocol from what currently exists in the specification. A KEM-based PAKE would provide FIDO2 with the desired resistance against active attackers *and* could additionally provide (some) protection against quantum attackers if the protocol uses a post-quantum KEM like ML-KEM. Therefore, we briefly describe three KEM-based PAKE protocols that we believe to be good candidates to use in sCTAP 2.1+: OCAKE [27, 53], a variant based on OCAKE that uses a Half-Ideal Cipher (HIC) and that we call HIC OCAKE [52], and CHIC [21].

The OCAKE protocol

This protocol, which was first introduced in [53], is a generic construction for transforming a KEM into a PAKE that employs a common strategy for KEM-based PAKE protocols: it uses an Ideal Cipher (IC) to encrypt Alice's public key under its shared, low entropy password (where the IC message space is the public key space of the KEM), and sends it to Bob, which decrypts the key using the same password and produces a ciphertext and a key confirmation tag. This tag is calculated on, among other data, the ssid (as in CPace), the password, the encrypted public key, the ciphertext and the pre-session key output by the KEM encapsulate function. Bob then sends this ciphertext and the key confirmation tag to Alice, and generates the final session key by hashing the ssid, encrypted public key, ciphertext, pre-session key and the key confirmation tag. Alice runs decapsulate on the ciphertext to obtain the pre-session key, verifies the received tag and if successful, generates the final session key.

Note that, with the above description, OCAKE only offers explicit authentication from Bob to Alice, whereas we require the PAKE protocol for sCTAP 2.1+ to have explicit mutual authentication. Indeed, Bob does not wait for any response from Alice to derive the final session key, but this can be changed, as pointed out in [27], by having Alice send Bob an

[6] To the best of our knowledge, no KEM-based PAKE protocol has of yet been proven UC secure against quantum adversaries.

additional tag, calculated on the same data as the first key confirmation tag (although the authors include an additional bit of information that signals if the tag was calculated by Bob or Alice). Bob then only produces the final session key if the tag verification succeeds. With this change, OCAKE satisfies our syntax requirements for sCTAP 2.1+.

The HIC OCAKE and CHIC protocols

The HIC OCAKE and CHIC protocols follow the same strategy of encrypting Alice's public key and then producing a ciphertext with a key confirmation tag, with the most significant difference being the way the public key encryption is accomplished. CHIC is based on HIC OCAKE [52], which is a protocol that uses a notion the authors called a Half-Ideal Cipher (HIC), a relaxation of the IC notion that is easier to instantiate and more efficient than other IC constructions over certain group elements. Very basically, with a HIC the length of the message and ciphertext space is extended from the group domain (the KEM public key space in this concrete example) by 2λ bits (for security parameter λ), with the extra bits added for randomness. As a result, the output ciphertext presents a small overhead of 2λ bits over the plaintext. The fundamental difference between an IC and a HIC is that the latter is required to be a pseudorandom permutation *only* in the added 2λ bits of the cipher domain, and not on the group element part of the ciphertext. It is shown in the paper that a HIC can be used as a drop-in replacement of an IC without affecting the UC security of the protocol. Furthermore, the authors showed that a HIC can be instantiated with a *modified 2-Feistel* (m2F) construction, which is based on a *2-Feistel* (2F) construction from [54].

The CHIC protocol improves on HIC OCAKE by eliminating the ciphertext overhead entirely through a repurposing of the m2F construction. The authors show that, if the underlying KEM has splittable keys (which essentially means that the public key can be split into two components where one is a group element and the other is a bit-string), then the split bit-string part of the public key can replace the random 2λ bit string in the m2F construction, and the resulting PAKE is still UC secure. Notably, one KEM scheme that has splittable public keys is Kyber (and ML-KEM), where the public key consists of a 32-byte seed ρ and a group element M. This way of encrypting a KEM public key through the m2F construction results in a ciphertext with the same length as the public key itself, because no additional pseudorandom bit-string is used in the encryption process.

The rest of the protocol is fairly straightforward, where Alice sends the encrypted public key to Bob, which decrypts the key using the inverse m2F construction and runs the KEM

encapsulate function to obtain the KEM ciphertext and pre-session key. The key confirmation tag and session key are both calculated using two hash functions with the same input, which includes the ssid, the encrypted and the decrypted KEM public key, the KEM ciphertext and the pre-session key[7]. This, as the authors point out, can be leveraged when instantiating the protocol. Indeed, if the tag and the key each have a length of 2λ, then the same result can be achieved by using a single hash function with an output size of 4λ, where the tag and the key correspond, respectively, to the leftmost and the rightmost 2λ bits, further reducing the computational cost of calculating these two elements of the protocol.

As was the case with OCAKE, CHIC can also achieve explicit mutual authentication if Alice sends Bob an additional tag, calculated on the same data.

Instantiating sCTAP 2.1+ with CHIC

We believe the improvements CHIC provides in terms of efficiency gains and simplicity of implementation make it a viable option for inclusion in sCTAP 2.1+, where bandwidth overhead reduction and the computational cost of the underlying protocol operations are especially important[8]. Furthermore, the building blocks of CHIC are fairly simple to implement, with the most challenging being perhaps the block cipher used in the m2F construction, which the authors instantiated with the Rijndael version that uses 256-bit blocks and keys, and the hash function that maps the seed p into a group element, which the authors instantiated with the rejection sampling procedure that is already used by Kyber, repurposed to sample only a polynomial vector of size k instead of a polynomial matrix of size $k \times k$, for $k \in 2, 3, 4$.

In Chapter 7, we introduce our Rust implementation of CHIC, which was adapted from the author's reference implementation in C [9], as well as its integration on commercial implementations of CTAP2 maintained by Mozilla and Nitrokey.

[7] Notably, the password is not included as an input, and is only used in the m2F construction. As the authors note, this is advantageous in the sense that it reduces the amount of time the password has to be stored in memory.

[8] Recent publications introduced two PAKE constructions, named NoIC [55] and Tempo [56], which are based on CHIC and may therefore also be good candidates for sCTAP 2.1+. However, our work was conducted prior to these publications, and therefore we did not consider them for our implementation.

[9] https://github.com/mbbarbosa/pake-constructions

7. Experimental results

In the previous Chapters, we have shown how FIDO2 can be strengthened to provably prevent rogue key and man-in-the-middle attacks, by introducing bi-directional authentication in the client-authenticator communication channel and authenticating the Bind procedure to resist active attacks. As previously mentioned, one of the core goals with our proposed changes was to make them as easy as possible to integrate with the current FIDO2 standard by making them modular, so that changes to current implementations can be contained to specific parts of the protocol, and backwards compatible with prior iterations, so that any client or authenticator that includes our proposed changes works seamlessly with existing implementations. Indeed, the CTAP 2.1+ protocol only differs from CTAP 2.1 after Bind is finished, and sCTAP 2.1+ modifies the entire Bind procedure (with minimal tweaks to Setup), but does not change anything *after* Bind ends, which means that these changes can coexist naturally or be applied to CTAP 2.1 independently of each other.

Implementation of CTAP 2.1+ and sCTAP 2.1+

To better understand the performance and integration costs of our proposed modifications, we implemented the rogue key attack fix (CTAP 2.1+) and the man-in-the-middle fix (sCTAP 2.1+) in commercial implementations of CTAP2. For client-side modifications, we used the authenticator-rs CTAP2 implementation from Mozilla[1]. This project is used by Firefox as a module that implements all the CTAP2 logic, but can also be used independently from Firefox (similarly to libfido2). For token-side modifications, we used the nitrokey-3-firmware CTAP2 implementation from Nitrokey[2], developed and maintained in collaboration with SoloKeys[3]. Both projects are open-source and written in Rust.

This Chapter presents our experimental results from the implementation of CTAP 2.1+ and sCTAP 2.1+[4], and is divided into three sections. Section 7.1 provides a very concise overview of our implementation of CTAP 2.1+. Section 7.2 presents a similar overview for sCTAP 2.1+ and the auxiliary implementation work required for this integration. We present our results on the performance impact from both protocols in Section 7.3.

[1] `https://github.com/mozilla/authenticator-rs`

[2] `https://github.com/Nitrokey/nitrokey-3-firmware`

[3] https://solokeys.com/

[4] Note that whenever we mention the implementation of CTAP 2.1+ and sCTAP 2.1+, we are refering to the implementation of the modifications each protocol introduces to CTAP 2.1. This is simply a simplification of language to improve readability.

7.1. Implementation of CTAP 2.1+

In this Section, we provide a high-level overview over the most important changes from our CTAP 2.1+ implementation on the existing Mozilla and Nitrokey projects: (i) the pinUvAuthToken expansion, (ii) the authentication and validation of token responses[5], and (iii) the implementation of full backwards compatibility.

For simplicity, we do not provide implementation details here. Those can be consulted by examining our publicly available source code[6][7].

7.1.1. The pinUvAuthToken expansion

Recall from Chapter 5 that we introduce a new PIN/UV Auth Protocol function for CTAP 2.1+ called expand, which uses a cryptographic primitive we modeled with $\mathcal{H}_3$. For our implementation of CTAP 2.1+, we opted to instantiate $\mathcal{H}_3$ with HKDF-SHA-256, since it is already a part of the CTAP2 standard, and thus is available for any commercial implementation of the protocol.

The pinUvAuthToken expansion is identical to the derivation of the two shared secret keys in PIN/UV Auth Protocol 2, except that the two strings we pass as parameters to the HKDF-SHA-256 are ”CTAP2.1+ Client key” for the client key and ”CTAP2.1+ Token key” for the token key. We implemented this expansion on the client-side and token-side implementations by adding functions that implement the logic in the expand function from Figure 5.5 to the existing PIN/UV Auth Protocol implementations in both projects.

7.1.2. Authenticating and verifying token responses

CTAP 2.1+ client commands are authenticated by producing an HMAC-SHA-256 tag on the clientDataHash parameter using the aforementioned client key. We followed a similar strategy for authenticated responses, where the token key is used to produce an authentication tag on the response. Concretely, on responses to a authenticatorMakeCredential command, the token key is used to generate an HMAC-SHA-256 tag on the authenticator data and attestation object. On responses to authenticatorGetAssertion commands, the tag is generated on the authenticator data and the assertion signature. In CTAP 2.1+, this tag is included in the token response, and verified by the client before it is accepted.

[5] For simplicity, we have only added the CTAP 2.1+ modifications to the authenticatorMakeCredential and authenticatorGetAssertion commands, since these are the most important authenticated commands in the context of this work. Nevertheless, expanding our modifications to other authenticated commands follows the same procedures.

[6] Client-side (forked from Mozilla): `https://github.com/esquivel71/authenticator-rs_fork`.

[7] Token-side (forked from Nitrokey): `https://github.com/esquivel71/nitrokey-3-firmware_fork`.

7.1.3. Backwards compatibility

We implemented CTAP 2.1+ in a way that provides full backwards compatibility. To do this, we added a mechanism in which the client can send a new ma[8] option flag (like the up or uv flags). If the token receives this ma flag *and* supports CTAP 2.1+, then it performs the pinUvAuthToken expansion, uses the client key to verify the client command, and uses the token key for producing its authenticated response. If not, then the pinUvAuthToken is used as in CTAP 2.1. The client will only send this flag if the token includes a new version of FIDO2, called "FIDO_2_1_P"[9], in its list of supported versions that is included in the authenticatorGetInfo response. Likewise, a client that does not support CTAP 2.1+ and receives this version will simply ignore it.

Note that with this implementation, a client that supports CTAP 2.1+ can interact with any authenticator that supports CTAP 2.1 (or CTAP 2.0) by behaving exactly like a regular CTAP2 client, and vice-versa.

7.2. Implementation of sCTAP 2.1+

For our implementation of sCTAP 2.1+, we chose to instantiate the PAKE with the CHIC protocol. As with CTAP 2.1+, our implementation of sCTAP 2.1+ was performed on Mozilla's and Nitrokey's implementations, although on a different version of those projects that was forked by SandboxAQ[10][11] to include an integration of Kyber for performing the (unauthenticated) key exchange on CTAP2. We decided to build our implementation of sCTAP 2.1+ on these forks for two reasons: (i) the integration of Kyber in CTAP2 by SandboxAQ provided a basis on which we could then implement the integration of CHIC, and (ii) the fact that this would enable the additional performance comparison between Kyber and CHIC, allowing us to better understand the overhead introduced by the latter.

As with CTAP 2.1+, this Section provides a very high-level overview of our implementation work, and does not provide implementation details. Those can be consulted in our open-source implementations [12][13], based on SandboxAQ's forks.

[8] For "mutual authentication" (not to be confused with the *explicit mutual authentication* property from the PAKE.)

[9] This represents FIDO2.1+, which follows the nomenclature used for specifying which FIDO2 versions are supported by authenticators.

[10] `https://github.com/sandbox-quantum/authenticator-rs`

[11] `https://github.com/sandbox-quantum/nitrokey-3-firmware/`

[12] Client-side: `https://github.com/esquivel71/PAKE_authenticator_rs`

[13] Token-side: `https://github.com/esquivel71/PAKE_nitrokey-3-firmware_pqc`

For this implementation, we only considered the Kyber-768 parameter set (the configuration with the NIST Security Category 3), as this is the default parameter set recommended by NIST [24].

7.2.1. Rust implementation of CHIC

As mentioned in Chapter 6, the authors in [21] provide a reference implementation of CHIC, written in C, that uses the Kyber reference implementation underneath. However, because the open-source FIDO2 projects on which we based our implementations of CTAP 2.1+ and sCTAP 2.1+ are written in Rust, we implemented our own version of CHIC in the same language, based on the authors' reference implementation. We very briefly mention the two most important additions from our implementation with respect to the integration of CHIC in CTAP2, and refer to our publicly available source-code[14] for more details.

Explicit mutual authentication

Our implementation of CHIC adds explicit mutual authentication, since this is a requirement we set for any PAKE used in sCTAP 2.1+. This addition was performed in the internal logic of the protocol's implementation, where it now produces a second authentication tag (recall that in sCTAP 2.1+ this corresponds to the tag output by the authenticator and sent to the client in the last message of Bind). However, our implementation leaves the decision of validating the second key confirmation tag to the calling application, which may choose to ignore it if explicit mutual authentication is not required. For sCTAP 2.1+, this means that the validation of the second key confirmation tag (of the last message sent from the token to the client) must be performed on the client-side application logic itself.

Multiple Kyber and ML-KEM implementations

The original C implementation is bound to a specific Kyber implementation, which is compiled directly into an executable. If one wishes to use a different library, it must be added to the Makefile and recompiled into a new executable. In our version, we added support for using *any* implementation of Kyber and ML-KEM underneath, without requiring changes to the CHIC source-code. We added this functionality to open the way to a more thorough benchmark process, because it enabled us to perform the Bind procedure using CHIC with different implementations by merely tweaking the configurations on the Mozilla

[14] https://github.com/esquivel71/chic-pake-rust

and Nitrokey projects. Furthermore, it also has the added benefit of increasing our implementation's versatility, by offering any application that uses our version of CHIC the ability to choose what Kyber/ML-KEM implementation to use.

7.2.2. Jasmin ARMv7-M implementation of ML-KEM

The Jasmin language is a very low level programming language purpose-built for creating fast, high-assurance and verified implementations of cryptographic software, maintained by the Formosa Crypto project. The project maintains a library called libjade, which at the time of writing contains several verified implementations of cryptographic primitives like SHA2, SHA3 and ML-KEM. A full list can be consulted in the library's repository[15].

The added support for different ML-KEM implementations on CHIC opened an interesting path for the inclusion of a verified Jasmin implementation of ML-KEM as well, as this would allow us to compare the performance of the libjade version of ML-KEM in a "real-world" usage scenario, by using it to perform the Bind procedure on commercial implementations of CTAP2 (with CHIC and as a standalone KEM as integrated by SandboxAQ).

Unlike other low level languages like C, Jasmin code must be written for a concrete processor architecture. The libjade implementation of ML-KEM was written for the x86 architecture, meaning that it could be used by CHIC in Mozilla's implementation but not on the Nitrokey, which uses a Cortex-M33 processor with the ARMv8-M architecture. Fortunately, the Jasmin compiler recently added support for the ARMv7-M architecture, which is fully compatible with ARMv8-M, and thus any program compiled for ARMv7-M can run on the Nitrokey. This enabled us to write an ARMv7-M version of ML-KEM based on the x86 implementation (we only implemented ML-KEM-768) and use it on a Nitrokey running our implementation of sCTAP 2.1+. We refer to our open-source implementation for more details[16].

Accessing the Jasmin implementation in Rust

Our version of CHIC is written in Rust. Consequently, while it supports any underlying Kyber or ML-KEM implementation, that implementation must be written in Rust or provide a Rust wrapper for other languages.

To bridge the gap between our CHIC implementation and the Jasmin implementation of ML-KEM (the x86 and ARMv7-M versions), we created a new Rust library that acts as a

[15] https://github.com/formosa-crypto/libjade
[16] https://github.com/formosa-crypto/formosa-mlkem/tree/mlkem_arm

wrapper for any implementation in assembly. This library, named kyber_asm, exposes an API for the three KEM functions keygen, encapsulate and decapsulate that can be consumed by any Rust application, and redirects calls to these functions to their respective assembly implementations.

We refer to our open-source implementation[17] for more details, but the crucial point is that this library enabled us to successfully use our ARMv7-M implementation of ML-KEM on CHIC, running on the Nitrokey firmware, as well as the x86 version on Mozilla's authenticator-rs client-side implementation (we support both architectures).

Kyber and ML-KEM

Note that we have so far differentiated between Kyber and ML-KEM implementations. Indeed, the SandboxAQ fork on which we based this part of our work included a concrete implementation of Kyber, as did the original reference implementation of CHIC. While similar, Kyber and ML-KEM are not the same (see Chapter 2), and our work on the sCTAP 2.1+ implementation involved working with both schemes. Specifically, as explained above, we created an implementation of ML-KEM in Jasmin, and integrated that version in CHIC as well. This, and the x86 version, are the only implementations of ML-KEM we used in this work. All other implementations we used are of Kyber.

7.2.3. Adding CHIC to CTAP2

Next, we describe the high-level changes required for the implementation of sCTAP 2.1+, which can be logically divided in three steps: (i) the addition of a new PIN/UV Auth Protocol, (ii) the process of obtaining the "pinUvAuthToken" [18], including the integration of the explicit mutual authentication support, and (iii) the implementation of full backwards compatibility.

Adding the new PIN/UV Auth Protocol

In Chapter 6 we introduced a new PIN/UV Auth Protocol instantiation that implements the lower-level functionality required to perform the Bind procedure with a generic PAKE. We added this new PIN/UV Auth Protocol instantiation, which we call PIN/UV Auth Protocol 4 (since we attribute PIN/UV Auth Protocol 3 to the already existing Kyber integration), to the Mozilla and Nitrokey implementations.

[17] https://github.com/esquivel71/kyber_asm

[18] We use quotes here because, in sCTAP 2.1+, the pinUvAuthToken no longer exists in its original capacity when performing Bind with a PAKE.

The client-side implementation from Mozilla already supported CTAP 2.1, and thus already had support for different PIN/UV Auth Protocol instantiations. As a result, adding our changes to this implementation was simpler, requiring us implement the internal logic of the PIN/UV Auth Protocol 4 that utilizes CHIC underneath, but exposing that logic through the interface used by the other protocol instantiations (with minimal changes).

The Nitrokey token-side implementation that was forked by SandboxAQ supported only CTAP 2.0. As a result, in order to add PIN/UV Auth Protocol 4, we had to first implement support for multiple PIN/UV Auth Protocol instantiations[19] before adding the internal the logic of PIN/UV Auth Protocol 4. This added support was not strictly necessary for benchmarking Bind using CHIC, but significantly increased the flexibility of our implementation, allowing us to run benchmarks on the different PIN/UV Auth Protocol instantiations without having to recompile the code.

Obtaining a ”pinUvAuthToken”

Our inclusion of CHIC in CTAP2 follows the approach of setting the session key derived by each party as the binding state, i.e., as the key that will be then used for authenticating commands. This is a modification to the protocol that goes beyond simply changing the primitive that performs the unauthenticated key exchange to derive a shared secret, because this approach essentially ”merges” the two actions in one, with a client deriving a shared secret with the token *and* setting that shared secret as the binding state.

On the token-side, the session key is derived and set as the binding state in the same stage of the execution (the PAKE execution reaches *accept* and *terminate* in the same step), i.e., when the token receives the getPinToken subcommand, just like with any other PIN/UV Auth Protocol instantiation. Therefore, the most notable aspect of our implementation here is that PIN/UV Auth Protocol 4 sets the binding state with the session key derived from CHIC, instead of the randomly generated pinUvAuthToken used for the other PIN/UV Auth Protocol instantiations.

On the client-side, the session key is derived before the client sends the getPinToken subcommand to the token. However, because we require explicit mutual authentication, this key is stored until the client receives the token’s response to getPinToken with the second key confirmation tag (calculated by CHIC), and only then is it set as the client’s binding state. Thus, the main changes we added to the client-side implementation are

[19]The SandboxAQ fork we used had a flag to enable Kyber *or* ECDH, which meant that the code had to be recompiled to test different PIN/UV Auth Protocol instantiations.

related to the repurposing of the key derived from encapsulate and the implementation of the verification of the key confirmation tag that the client receives from the token, instead of the encrypted pinUvAuthToken.

Backwards compatibility

Our implementation of sCTAP 2.1+ is fully compatible with existing implementations of CTAP2. The token provides its support of PIN/UV Auth Protocol versions with the authenticatorGetInfo command, and the client only chooses PIN/UV Auth Protocol 4 if it is contained in the authenticatorGetInfo response. Therefore, any token that only supports the standard PIN/UV Auth Protocol instantiations will never receive a request from the client to "obtain a pinUvAuthToken" using CHIC. Likewise, adding CHIC to the token-side implementation does not influence its compatibility with clients that do not support it. A client that does not support PIN/UV Auth Protocol 4 will never choose it, and thus will request a pinUvAuthToken using another PIN/UV Auth Protocol instantiation.

7.3. Performance results

Our implementations show that it is possible to strengthen FIDO2 without incurring high implementation costs or drawbacks regarding compatibility with previous versions. Fixing the rogue key attack vulnerability with CTAP 2.1+ requires simple modifications to each party, and does not use any cryptographic primitives that aren't already part of the CTAP2 protocol. Fixing the vulnerability to man-in-the-middle attacks with sCTAP 2.1+ is slightly more complex, and requires changes to the entire Bind procedure, but those are mostly contained on a new PIN/UV Auth Protocol instantiation. However, we have not yet covered how our fixes affect the performance of CTAP2, nor how much overhead they add to the final binary size that is flashed on the authenticator.

In this Section, we present our results regarding the execution times and binary size overhead for our implementations of CTAP 2.1+ and sCTAP 2.1+.

7.3.1. Benchmarking hardware and software

All of our client-side testing was performed on a Lenovo YOGA 710 14IKB, with 8 GB of RAM and an Intel Core i5-7200U CPU clocked at 2.50GHz, running Linux Mint 21.3. We used the authenticator-rs project's pre-existing set of testing files, located in the examples folder at the root of the repository, as a basis for our benchmarks. For benchmarking CTAP 2.1+, we performed full registration runs (with a previously performed Bind to set

the binding state) with mock data and toggleable bi-directional authentication option. For sCTAP 2.1+, we benchmarked only the Bind procedure, since our fix does not affect the registration/authentication execution.

On the authenticator side, all tests were performed on a Nitrokey 3A NFC Hacker[20], a development-focused programmable authenticator sold by Nitrokey based on an ARM Cortex-M33 microcontroller from NXP called LPC55S69JEV98 [21], on which we flashed all variants of the nitrokey-3-firmware code being tested.

7.3.2. Results for CTAP 2.1+

Recall that our implementation involves adding a new symmetric key expansion step to both sides, an HMAC-SHA-256 generation on the token-side and HMAC-SHA-256 verification on the client-side, and a new field on the token's response containing its authentication tag delivered to the client.

Code-size and bandwidth overhead

Because we reuse pre-existing key expansion and HMAC-SHA-256 computation code, the resulting impact on the code footprint is very small, with an increase of only 0.35% in the binary size for the token. This is to be expected, because our modifications on the token-side are restricted to calling HMAC-SHA-256 an additional time for authenticating responses, implementing the pinUvAuthToken expansion, and implementing the backwards compatibility strategy by adding a new protocol version string. Likewise, the bandwidth increase is only 32-bytes in token-to-client responses.

Time overhead

To measure the impact on the overall execution time of performing an entire registration/authentication flow, we sent 30 back-to-back registration commands to the authenticator, with and without bi-directional authentication (i.e. including and excluding the ma flag on the options parameter), and took the average of all execution times, rounded to three decimal places. The measured impact on round trip time was approximately 13.4%, with a difference in the range of 0.09 seconds. However, note that these results were obtained over a naive adaptation of the code, without any attempt to perform non-trivial optimizations.

[20] https://shop.nitrokey.com/shop/nk3anh-nitrokey-3a-nfc-hacker-404
[21] https://www.nxp.com/part/LPC55S69JEV98

7.3.3. Results for sCTAP 2.1+

We performed the same type of benchmarks for sCTAP 2.1+, including the same measurements to execution times and increase in code size. Note, however, that we integrated a PAKE protocol based on a post-quantum KEM. Thus, instead of only comparing our implementation with the original CTAP2 running the ECDH key exchange, we also compare it with the original SandboxAQ version implementing Bind with Kyber, so that the we can also measure the overhead caused by CHIC specifically.

We also measured the impact of using CHIC with different Kyber and ML-KEM implementations, to assess if the performance and binary size varies noticeably depending on the concrete implementation underneath. However, note that during the development of sCTAP 2.1+, we adapted the SandboxAQ integration to always use the same underlying implementation of Kyber or ML-KEM as CHIC, instead of using only the concrete implementation that was originally included. This ensured that all of our benchmarks on PIN/UV Auth Protocol 3 and 4 were performed on the same Kyber/ML-KEM implementation (i.e. we didn't, for example, run benchmarks on PIN/UV Auth Protocol 4 with ML-KEM and PIN/UV Auth Protocol 3 with Kyber).

Kyber vs ML-KEM For simplicity, from now on we refer to the libjade implementation (x86 and ARMv7-M) as a Kyber implementation, although it is in fact an implementation of ML-KEM. This simplifies our presentation of results obtained from benchmarking the other Kyber implementations we tested in sCTAP 2.1+.

Code-size and bandwidth overhead

With sCTAP 2.1+ we introduce new cryptographic primitives and external code that are not part of the original implementations, and so the increase in the firmware binary size is greater than that of CTAP 2.1+.

To compare the resulting binary sizes, we compiled multiple versions of the Nitrokey firmware: (i) the original Nitrokey firmware implementation, on version 1.6.0[22], without any post-quantum additions; (ii) the (unaltered) SandboxAQ version with the addition of Kyber, using the pqcrypto-kyber implementation[23] (a package that provides Rust bindings to the PQClean C implementation); (iii) our version, based on SandboxAQ's, with the addition of CHIC using the same Kyber pqcrypto-kyber implementation underneath; (iv)

[22]The version on which the SandboxAQ fork we used was built.
[23]https://crates.io/crates/pqcrypto-kyber

the same version with CHIC but using pqc_kyber, Argyle Software's Rust implementation of Kyber; (v) and the same version but with our libjade ARMv7-M implementation. The results can be consulted in Figure 7.1.

Configuration	Binary size (bytes)	Δ (%)	Δ_{Kyber} (%)
ECDH	514 072	–	–
Kyber (pqcrypto)	539 032	4.86%	–
CHIC (pqcrypto)	567 288	10.35%	5.24%
CHIC (pqc_kyber)	572 088	11.29%	6.13%
CHIC (libjade)	588 539	14.49%	9.18%

Figure 7.1: Nitrokey firmware binary sizes, in bytes, for each configuration tested. The Δ column contains the percentage overhead in size that each configuration has over ECDH, and Δ_{Kyber} contains the percentage overhead each CHIC configuration has over Kyber. The firmware binary used to obtain these results was runner-lpc55-nk3xn.bin, which can be found under runners/embedded/artifacts/ on the nitrokey-3-firmware project.

.

One can see from these results that the inclusion of CHIC adds to the overhead already present in SandboxAQ's version of the firmware, with the minimum increase in size being of approximately 5%, corresponding to around 28 kB, when using pqcrypto. Other implementations add more to the size of the firmware, with a clear increase when using the libjade version.

This overall increase is mainly due to the inclusion of CHIC, but may in a smaller part also have been caused by some changes we made to the SandboxAQ version so that multiple PIN/UV Auth Protocol instantiations are supported, since the version we used supported either Kyber or ECDH.

Time overhead

Figure 7.2 shows our results on the execution times of performing Bind for each PIN/UV Auth Protocol instantiation available in our implementation. Our benchmarks were performed identically to CTAP 2.1+, where we ran the benchmarking program to perform 30 executions of Bind for every PIN/UV Auth Protocol instantiation and the three aforementioned implementations of Kyber: pqcrypto-kyber, pqc_kyber, and libjade.

Furthemore, we also benchmarked CHIC in two modes, with the difference between the two being in the instantiation of the hash functions used. In the first mode, we use SHA2 (SHA-256 and SHA-512) for the creation of all key confirmation tags and other hashing operations inside the m2F construction, while in the second we use SHA3 (SHA3-256 and SHA3-512).

Benchmarks	libjade			pqcrypto-kyber			pqc_kyber		
	Time (s)	Δ (%)	Δ_{Kyber} (%)	Time (s)	Δ (%)	Δ_{Kyber} (%)	Time (s)	Δ (%)	Δ_{Kyber} (%)
ECDH	0.4053	0.00%	–	0.4053	0.00%	–	0.4053	0.00%	–
Kyber	0.5050	24.60%	–	0.5054	24.70%	–	0.5029	24.08%	–
CHIC (sha2)	0.5066	24.99%	0.32%	0.5110	26.08%	1.10%	0.5067	25.02%	0.76%
CHIC (sha3)	0.6061	49.54%	20.02%	0.6061	49.54%	19.92%	0.6062	49.57%	20.55%

Figure 7.2: Performance benchmarks for Bind using PIN/UV Auth Protocol 1 (ECDH), 3 (Kyber) and 4 (CHIC). The execution times are represented in seconds, rounded to the 4^{th} decimal place. The Δ column contains the percentage overhead each algorithm has over ECDH, and Δ_{Kyber} contains the percentage overhead each CHIC mode has over Kyber. Values were obtained by averaging 30 consecutive Bind executions for each represented combination of PIN/UV Auth Protocol instance and Kyber implementation.

We can take away some information from the results in Figure 7.2. First and foremost, the different instantiations of Kyber do not seem to significantly impact the execution times of performing Bind, both when using CHIC and Kyber directly.

Second, it is clearly the case that performing Bind with the original unauthenticated ECDH key exchange is faster than with any other of the tested options. Indeed, performing the key agreement with Kyber, with any of the tested implementations, has an execution cost approximately 25% higher than ECDH, clocking in at approximately half a second for a full Bind execution. This might be due to Kyber being executed in the Nitrokey authenticator we used, which is an embedded device. Unlike with x86 architectures, the Cortex-M33 does not have vectorization support, nor does it have as much memory available for memory-intensive operations. Furthermore, the sizes of Kyber key pairs and ciphertexts is much larger than ECDH key pairs, which might cause additional slowdowns when serializing and deserializing these keys, as well as when storing/reading them from the Nitrokey's storage between the getKeyAgreement and getPinToken requests.

As for the performance of CHIC, it essentially depends on what instantiation of the hash function is used. When using SHA2, the CHIC integration adds virtually no overhead over Kyber. This can be seen in the Δ_{Kyber} columns, which directly show how CHIC compares with Kyber for each implementation. When using SHA2, the execution times are nearly identical, with an overhead of around 1% at most, corresponding to approximately 5 milliseconds. Therefore, running Bind with CHIC in this mode incurs the same 25% overhead when compared to ECDH. Note that, because every Kyber operation is also present in CHIC, the additional overhead can basically be attributed to the m2F construction for the encryption/decryption of the token's public key, and the explicit mutual authentication functionality.

When using SHA3, there is a noticeable hit in performance, with the execution time of CHIC being approximately 20% higher than regular Kyber and 50% higher than ECDH. Since the only difference between the two modes with which we tested CHIC is the hash instantiation, this suggests that the differences are most likely due to the constrained environment on the embedded hardware that is running SHA3. This seems to be consistent with our tests on the same configuration on a virtual authenticator (the Nitrokey firmware can be emulated on an x86 architecture and accessed via USB/IP), where this difference between SHA2 and SHA3 was virtually nonexistent.

Note that although there is indeed a 25% (or 50%) increase in execution times when performing Bind with CHIC over ECDH, this in practice corresponds to approximately 0.1 (or 0.2) extra seconds, which are still almost imperceptible to a user using its authenticator to register or authenticate with a relying party.

7.3.4. Backwards compatibility testing

To measure the performance impact of our implementations, we used the authenticator-rs benchmarking tools directly, because our goal was to perform multiple consecutive executions of the protocols and measure the average execution times of Bind and of registration runs. However, as mentioned above, the authenticator-rs project is a module used by Firefox that implements all of the browser's CTAP2 functionalities.

Therefore, to test the backwards compatibility of our implementations, we compiled the Firefox browser from source, using the modified authenticator-rs, and performed several registration and authentication runs on the webauthn.io website using the Nitrokey flashed with the original firmware (i.e., a regular CTAP 2.1 authenticator).

We also performed the opposite experiment, where a modified Nitrokey firmware with our implementations was tested against regular Firefox and Google Chrome browsers.

In both cases, for CTAP 2.1+ and sCTAP 2.1+, we registered no incompatibility whatsoever between our modified implementations and regular CTAP2 clients and authenticators, supporting our claim that our implementations provide full backwards compatibility.

8. Conclusions and future work

While FIDO2 is a great step towards a future of passwordless authentication, in its current iteration it still has some problems that can leave it vulnerable to attacks.

Concretely, based on prior work on the provable security of FIDO2, we have shown that some of the CTAP2's design choices, i.e., the use of an unauthenticated ECDH key exchange and the lack of authentication on authenticator responses, can be exploited in practice, with an attacker being able to obtain unauthorized access to a user's account by performing (i) a man-in-the-middle attack as a stepping stone for an *impersonation attack*, (ii) a rogue key attack that registers its authenticator instead of the user's, or (iii) a combination of both that allows it to register its authenticator alongside the user's. To do so, we developed proof-of-concept implementations for all three attacks, and an attack vector that uses a malicious LibreOffice spreadsheet to install the malicious software on a user's machine. Our attacks require no special privileges to be executed, and require no modifications to the user's browser.

We followed up on this work by proposing two fixes to the protocol to make it resistant to rogue key and man-in-the-middle attacks. For rogue key attacks, we proposed the addition of authenticated responses from the token to the client, effectively creating a bi-directional authenticated channel between the client and the authenticator. We then proved that this modified CTAP2 protocol, which we called CTAP 2.1+, is resistant to rogue key attacks (i.e., that an attacker cannot deliver a message to a client that is accepted unless it knows the $\mathsf{pinUvAuthToken}$), via a security proof based on the security model and protocol syntax from [2]. We also show that FIDO2 as the composition of WebAuthn and CTAP 2.1+ is resistant against rogue key attacks even when the relying party requires no attestation from an authenticator, by proving that a relying party that is connected (via TLS) to an honest client can only accept a registration response from the token uniquely bound to that client (through CTAP 2.1+). This is a stronger result from what is currently possible to obtain using CTAP2.

For man-in-the-middle attacks, we proposed to replace unauthenticated ECDH key exchange with a PAKE, following a similar strategy to [1]. We based our work on the same security model and protocol syntax used for CTAP 2.1+, but with the removal of the assumption that an adversary cannot actively interfere with the protocol during the Bind

phase. We then proved that our modified protocol, which we called sCTAP 2.1+, is resistant against man-in-the-middle attacks in this model.

After showing that our proposed fixes can provably protect FIDO2 against these attacks, we implemented them on commercial implementations of FIDO2 from Mozilla and Nitrokey in Rust.

Our implementation of CTAP 2.1+ provides full backwards compatibility with previous versions, showing that our modifications can be integrated into existing codebases without risking incompatibility with already deployed solutions.

For sCTAP 2.1+, we opted to use CHIC as the underlying PAKE, which is based on a post-quantum KEM. For this work, we instantiated the KEM with both Kyber and ML-KEM. We added modular support for different implementations of these two schemes, which opened the door to the possibility of benchmarking sCTAP 2.1+ with a verified implementation of ML-KEM in Jasmin. To do so, we also had to implement a version of this scheme for ARMv7-M/ARMv8-M architectures. We successfully implemented these modifications in Mozilla's and Nitrokey's implementations and, as with CTAP 2.1+, our additions have full backwards compatibility with previous versions.

We also benchmarked the performance impact and overhead in binary size resulting from our modifications.

Our work shows that FIDO2 can be improved to make it resistant against man-in-the-middle and rogue key attacks. Our modifications are relatively easy to implement, provably increase the protocol's security, are compatible with previous versions, and incur a small performance cost that is, in all likelihood, imperceptible to a user that is registering or authenticating with a relying party.

8.1. Future work

Extending our analysis Our work focused exclusively on USB authenticators, with PIN-based user verification. Extending our analysis to other transport mechanisms, such as NFC or Bluetooth Low Energy, and to other forms of user verification like biometrics, may be an interesting prospect for future work.

Capturing additional CTAP2 features Another natural follow up on our work concerns the inclusion of other CTAP2 features in the protocol's analysis. For example, the HMAC Secret extension allows a platform to obtain a secret symmetric encryption key from the

authenticator, which is scoped to a credential. This involves an additional ECDH key exchange to derive a new shared secret, which is used to encrypt the key sent to the client on a authenticatorGetAssertion command, a feature that is not captured in our models.

Experimenting with more protocols Our results with CHIC and Kyber/ML-KEM suggest that the overhead caused by the PAKE-specific processing is minimal. It may be worth expanding this experiment by adding and benchmarking other post-quantum KEM and PAKE protocols, in order to determine which ones deliver the best performance and smallest overhead on binary size. It may also be worth studying how classical PAKE protocols like CPace fare in comparison. In fact, the authors in [1] already proposed instantiating sPACA with CPace, and reflected on the expected performance impact, although, to the best of our knowledge, no implementation was provided.

Post-quantum signatures in FIDO2 The post-quantum migration of the signature scheme used in WebAuthn is another important topic that warrants further research. Attestation signatures allow a relying party to confirm specific attributes of a token during registration, such as if it hardware-based or from a particular manufacturer. The relying party achieves this by using the PKI to verify that the attestation key is bound to a certificate issued by a trusted authority.

Some work on post-quantum signatures for FIDO2 has already been carried out by SandboxAQ, where they integrated CRYSTALS-Dilithium [57, 58] signatures on the Nitrokey token implementation and on a demo WebAuthn relying party implementation maintained by Yubico[1]. However, the broader ramifications of transitioning the FIDO2 ecosystem to post-quantum attestation signatures still warrant further research.

Modifications to CTAP2, such as switching to a (post-quantum) PAKE, primarily require updating client-side and token-side implementations, and do not directly affect relying parties, manufacturers or the broader PKI. Transitioning to post-quantum attestation signatures would require token manufacturers to provision new attestation keys on their authenticators, and many post-quantum schemes, such as ML-DSA, which is the standardized scheme derived from Dilithium, require keys that are considerably larger than those used in classical algorithms like ECDSA. Attestation certificates within the FIDO2 ecosystem could be affected as well, since the increase in signature size from post-quantum schemes like ML-DSA would result in significantly larger certificates.

[1] https://github.com/sandbox-quantum/java-webauthn-server/

Because the post-quantum migration of attestation signatures involves many stakeholders in the FIDO2 ecosystem (manufacturers, certificate authorities, clients, and relying parties), it is important to study how their adoption would impact the overall system, especially regarding viable migration options for each stakeholder and the difficulties or benefits associated with different post-quantum signature schemes (which vary widely in key and signature size).

References

[1] M. Barbosa, A. Boldyreva, S. Chen, and B. Warinschi, “Provable security analysis of FIDO2,” in *Advances in Cryptology - CRYPTO 2021*, ser. Lecture Notes in Computer Science, vol. 12827. Springer, 2021, pp. 125–156, Full version: https://eprint.iacr.org/2020/756. [Online]. Available: https://doi.org/10.1007/978-3-030-84252-9_5 [Cited on pages ii, iii, 1, 4, 5, 6, 7, 13, 17, 23, 65, 67, 68, 69, 70, 72, 75, 78, 85, 86, 101, 103, 104, 106, 116, 117, 119, 134, 152, and 154.]

[2] N. Bindel, C. Cremers, and M. Zhao, “Fido2, ctap 2.1, and webauthn 2: Provable security and post-quantum instantiation,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1471–1490. [Cited on pages ii, iii, 1, 4, 5, 6, 7, 8, 13, 65, 67, 68, 69, 70, 71, 72, 74, 75, 76, 78, 79, 80, 82, 84, 85, 86, 95, 97, 99, 101, 102, 104, 106, 112, 114, 115, 116, 119, 120, 152, and 204.]

[3] FIDO Alliance, “Fido alliance overview,” https://fidoalliance.org/overview/, 2026, accessed: 2026-01-27. [Cited on page 1.]

[4] Google, “The simplest, most secure way to sign into your accounts without a password,” https://safety.google/safety/authentication/passkey/, 2026, accessed: 2026-01-12.

[5] Microsoft, “What is fido2?” https://www.microsoft.com/en-us/security/business/security-101/what-is-fido2, 2026, accessed: 2026-01-13. [Cited on page 1.]

[6] FIDO Alliance, “Fido alliance,” https://fidoalliance.org/, 2026, accessed: 2026-01-27. [Cited on page 1.]

[7] I. B. Guirat and H. Halpin, “Formal verification of the w3c web authentication protocol,” ser. HoTSoS ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190619.3190640 [Cited on pages 1, 4, and 64.]

[8] C. Jacomme and S. Kremer, “An extensive formal analysis of multi-factor authentication protocols,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 1–15. [Cited on pages 1 and 4.]

[9] W3C, “Web authentication: An API for accessing public key credentials level 2 – W3C recommendation,” April 2021, https://www.w3.org/TR/2021/REC-webauthn-2-20210408/. [Cited on pages 3, 22, 25, 29, 69, 163, 164, 166, 169, 170, 173, 176, 177, 178, 184, 194, and 198.]

[10] F. Alliance, “Client to authenticator protocol (CTAP) – proposed standard,” July 2025, https://fidoalliance.org/specs/fido-v2.2-ps-20250714/fido-client-to-authenticator-protocol-v2.2-ps-20250714.html. [Cited on pages 3, 22, 32, 181, 182, 183, 184, 188, and 196.]

[11] ——, “Passkey adoption doubles in 2024,” https://fidoalliance.org/passkey-adoption-doubles-in-2024-more-than-15-billion-online-accounts-can-leverage-passkeys/, 2026, accessed: 2026-01-03. [Cited on page 3.]

[12] ——, “Fido alliance champions widespread passkey adoption and a passwordless future on world passkey day 2025,” https://fidoalliance.org/fido-alliance-champions-widespread-passkey-adoption-and-a-passwordless-future-on-world-passkey-day-2025/, 2026, accessed: 2026-01-03. [Cited on page 3.]

[13] Descope, “2025 fido report: The passwordless future,” https://www.descope.com/blog/post/2025-fido-report, 2026, accessed: 2026-01-03. [Cited on page 3.]

[14] F. Alliance, “Fido alliance launches the passkey pledge to further accelerate global movement away from passwords,” https://fidoalliance.org/fido-alliance-launches-the-passkey-pledge-to-further-accelerate-global-movement-away-from-passwords/, 2026, accessed: 2026-01-03. [Cited on page 3.]

[15] ——, “The passkey pledge,” https://fidoalliance.org/passkeypledge/, 2026, accessed: 2026-01-03. [Cited on page 3.]

[16] ——, “New fido alliance research shows 87% of u.s. and uk workforces are deploying passkeys for employee sign-ins,” https://fidoalliance.org/new-fido-alliance-research-shows-87-percent-us-uk-workforces-are-deploying-passkeys-for-employee-sign-ins/, 2026, accessed: 2026-01-03. [Cited on page 4.]

[17] O. Pereira, F. Rochet, and C. Wiedling, “Formal analysis of the fido 1.x protocol,” in *Foundations and Practice of Security*, A. Imine, J. M. Fernandez, J.-Y. Marion, L. Logrippo, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2018, pp. 68–82. [Cited on page 4.]

[18] R. Dumitru, D. Genkin, A. Wabnitz, and Y. Yarom, “The impostor among US(B): Off-Path injection attacks on USB communications,” in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5863–5880. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/dumitru [Cited on pages 7, 62, and 71.]

[19] M. Bellare, D. Pointcheval, and P. Rogaway, “Authenticated key exchange secure against dictionary attacks,” Cryptology ePrint Archive, Paper 2000/014, 2000. [Online]. Available: https://eprint.iacr.org/2000/014 [Cited on pages 8, 17, and 117.]

[20] V. Boyko, P. MacKenzie, and S. Patel, “Provably secure password-authenticated key exchange using diffie-hellman,” Cryptology ePrint Archive, Paper 2000/044, 2000. [Online]. Available: https://eprint.iacr.org/2000/044 [Cited on pages 8 and 117.]

[21] A. Arriaga, M. Barbosa, S. Jarecki, and M. Skrobot, “C’est très CHIC: A compact password-authenticated key exchange from lattice-based KEM,” Cryptology ePrint Archive, Paper 2024/308, 2024. [Online]. Available: https://eprint.iacr.org/2024/308 [Cited on pages 9, 136, and 142.]

[22] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS – kyber: a CCA-secure module-lattice-based KEM,” Cryptology ePrint Archive, Paper 2017/634, 2017. [Online]. Available: https://eprint.iacr.org/2017/634 [Cited on pages 9, 15, and 136.]

[23] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS – kyber: Algorithm specifications and supporting documentation (version 3.02),” 2021. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf [Cited on pages 9, 15, and 136.]

[24] National Institute of Standards and Technology, “Module-lattice-based key-encapsulation mechanism standard,” U.S. Department of Commerce, Tech. Rep. Federal Information Processing Standards (FIPS) 203, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf [Cited on pages 15, 16, 136, and 142.]

[25] A. Langlois and D. Stehle, “Worst-case to average-case reductions for module lattices,” Cryptology ePrint Archive, Paper 2012/090, 2012. [Online]. Available: https://eprint.iacr.org/2012/090 [Cited on page 15.]

[26] E. Fujisaki and T. Okamoto, “Secure integration of asymmetric and symmetric encryption schemes,” in *Advances in Cryptology — CRYPTO’ 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 537–554. [Cited on page 15.]

[27] N. Alnahawi, K. Hövelmanns, A. Hülsing, S. Ritsch, and A. Wiesmaier, “Towards post-quantum secure PAKE - a tight security proof for OCAKE in the BPR model,” Cryptology ePrint Archive, Paper 2023/1368, 2023. [Online]. Available: https://eprint.iacr.org/2023/1368 [Cited on pages 17, 18, 136, and 266.]

[28] W3C, “Web authentication: An API for accessing public key credentials level 3 – W3C recommendation,” January 2025, https://www.w3.org/TR/webauthn-3/. [Cited on pages 22, 25, and 68.]

[29] F. Alliance, “Fido alliance passkeys,” https://fidoalliance.org/passkeys. [Cited on page 24.]

[30] FIDO Alliance, “Fido alliance ux guidelines for passkey creation and sign-ins,” FIDO Alliance, Tech. Rep., May 2023, accessed: 2026-07-26. [Online]. Available: https://fidoalliance.org/wp-content/uploads/2023/05/FIDO-Alliance-UX-Guidelines-for-Passkey-Creation-and-Sign-ins.pdf [Cited on page 24.]

[31] ——, “Synced passkey deployment: Emerging practices for consumer use cases,” FIDO Alliance, White Paper, May 2024, accessed: 2026-07-26. [Online]. Available: https://fidoalliance.org/wp-content/uploads/2024/05/Synced-Passkey-Deployment_-Emerging-Practices-for-Consumer-Use-Cases_2024-May-31.pdf [Cited on page 24.]

[32] ——, “Client to authenticator protocol (CTAP) – proposed standard,” January 2019, https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html. [Cited on page 32.]

[33] F. Alliance, “Client to authenticator protocol (CTAP) – proposed standard,” June 2021, https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html. [Cited on pages 32 and 181.]

[34] A. Continella, M. Carminati, M. Polino, A. Lanzi, S. Zanero, and F. Maggi, “Prometheus: Analyzing webinject-based information stealers,” *Journal of Computer Security*, vol. 25, no. 2, pp. 117–137, 2017. [Cited on pages 57 and 61.]

[35] MITRE, “T1111 - Multi-Factor Authentication Interception,” 2021, https://attack.mitre.org/techniques/T1111/. [Cited on pages 57 and 61.]

[36] M. Barbosa, A. Cirne, and L. Esquível, “Rogue key and impersonation attacks on fido2: From theory to practice,” in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES ’23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3600160.3600174 [Cited on page 60.]

[37] J. Blasco, “Sykipot variant hijacks DOD and Windows smart cards,” 2012, https://www.alienvault.com/open-threat-exchange/blog/sykipot-variant-hijacks-dod-and-windows-smart-cards. [Cited on page 61.]

[38] FIDO Alliance, “User authentication specifications overview,” https://fidoalliance.org/specifications/, Accessed August 2025. [Cited on page 64.]

[39] N. Bindel, N. Gama, S. Guasch, and E. Ronen, “To attest or not to attest, this is the question–provable attestation in fido2,” in *ASIACRYPT 2023*. Springer, 2023, pp. 297–328. [Cited on pages 65, 68, 70, 95, 97, 99, 100, 101, and 102.]

[40] A. Popov, M. Nyström, D. Balfanz, and J. Hodges, “The token binding protocol version 1.0,” 2018, https://datatracker.ietf.org/doc/html/rfc8471. [Cited on page 68.]

[41] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux, “Password interception in a SSL/TLS channel,” in *Advances in Cryptology - CRYPTO 2003*, ser. Lecture Notes in Computer Science, vol. 2729. Springer, 2003, pp. 583–599. [Online]. Available: https://iacr.org/archive/crypto2003/27290581/27290581.pdf [Cited on page 85.]

[42] H. Krawczyk and P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF),” RFC 5869, May 2010. [Online]. Available: https://www.rfc-editor.org/info/rfc5869 [Cited on page 87.]

[43] K. Igoe, D. McGrew, and M. Salter, “Fundamental Elliptic Curve Cryptography Algorithms,” RFC 6090, Feb. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6090 [Cited on page 87.]

[44] M. Dworkin, “Recommendation for block cipher modes of operation,” *Methods and Techniques*, 2001. [Cited on page 87.]

[45] M. Bellare, R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication,” in *CRYPTO 1996*. Springer, 1996, pp. 1–15. [Cited on page 87.]

[46] B. Haase and B. Labrique, “AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–48, 2019. [Cited on pages 118 and 134.]

[47] M. Abdalla and D. Pointcheval, “Simple password-based encrypted key exchange protocols,” in *CT-RSA 2005*. Springer, 2005, pp. 191–208. [Cited on pages 118 and 134.]

[48] M. Abdalla and M. Barbosa, “Perfect forward security of SPAKE2,” Cryptology ePrint Archive, Report 2019/1194, 2019, https://eprint.iacr.org/2019/1194. [Cited on page 118.]

[49] M. Abdalla, B. Haase, and J. Hesse, “Security analysis of CPace,” Cryptology ePrint Archive, Paper 2021/114, 2021. [Online]. Available: https://eprint.iacr.org/2021/114 [Cited on page 134.]

[50] ——, “Cpace, a balanced composable pake,” 2022. [Online]. Available: https://www.ietf.org/archive/id/draft-irtf-cfrg-cpace-06.html [Cited on pages 134 and 135.]

[51] T. Bradley, J. Camenisch, S. Jarecki, A. Lehmann, G. Neven, and J. Xu, “Password-authenticated public-key encryption,” Cryptology ePrint Archive, Paper 2019/199, 2019. [Online]. Available: https://eprint.iacr.org/2019/199 [Cited on page 136.]

[52] B. F. D. Santos, Y. Gu, and S. Jarecki, “Randomized half-ideal cipher on groups with applications to UC (a)PAKE,” Cryptology ePrint Archive, Paper 2023/295, 2023. [Online]. Available: https://eprint.iacr.org/2023/295 [Cited on pages 136 and 137.]

[53] H. Beguinet, C. Chevalier, D. Pointcheval, T. Ricosset, and M. Rossi, “GeT a CAKE: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges,” Cryptology ePrint Archive, Paper 2023/470, 2023. [Online]. Available: https://eprint.iacr.org/2023/470 [Cited on page 136.]

[54] I. McQuoid, M. Rosulek, and L. Roy, “Minimal symmetric PAKE and 1-out-of-n OT from programmable-once public functions,” Cryptology ePrint Archive, Paper 2020/1043, 2020. [Online]. Available: https://eprint.iacr.org/2020/1043 [Cited on page 137.]

[55] A. Arriaga, M. Barbosa, and S. Jarecki, “NoIC: PAKE from KEM without ideal ciphers,” Cryptology ePrint Archive, Paper 2025/231, 2025. [Online]. Available: https://eprint.iacr.org/2025/231 [Cited on page 138.]

[56] ——, “Tempo: ML-KEM to PAKE compiler resilient to timing attacks,” Cryptology ePrint Archive, Paper 2025/1399, 2025. [Online]. Available: https://eprint.iacr.org/2025/1399 [Cited on page 138.]

[57] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, “CRYSTALS – dilithium: Digital signatures from module lattices,” Cryptology ePrint Archive, Paper 2017/633, 2017. [Online]. Available: https://eprint.iacr.org/2017/633 [Cited on page 154.]

[58] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, “CRYSTALS – dilithium: Algorithm specifications and supporting documentation (version 3.1),” 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf [Cited on page 154.]

[59] IANA, “Cbor object signing and encryption (cose),” May 2025, https://www.iana.org/assignments/cose/cose.xhtml. [Cited on page 165.]

[60] J. Schaad, “Rfc 9052 cbor object signing and encryption (cose): Structures and process,” June 2025, https://www.rfc-editor.org/rfc/rfc9052. [Cited on page 174.]

[61] C. Bormann and P. Hoffman, “RFC 8949 Concise Binary Object Representation (CBOR),” December 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8949.html [Cited on page 181.]

A. The WebAuthn Protocol

Here, we present a more detailed description of the WebAuthn protocol, introduced in Chapter 3. Reading this Appendix is not required for understanding the work in this thesis, but it does help to paint a more complete picture over how the protocol works, since it is meant to be a self-contained, condensed account of the WebAuthn protocol's elements that are most related to our work. For the sake of completeness, this Appendix contains some repeated information that was already introduced in Chapter 3.

Structure of this Appendix

We provide our description of the WebAuthn protocol in three sections. Section A.1 describes the procedure that is carried out during a registration run, which we separated into three steps. The first step starts with a user requesting to register with a relying party, which immediately gathers all the necessary information that is then handled to the client platform (usually the user's browser). This starts the second step, where the client handles all communications with the authenticator and, when finished, returns the response to the relying party. Finally, in the third step the relying party evaluates the response to assess whether the registration attempt is successful or not. Section A.2 introduces the notion of *attestation*, an extremely important part of the WebAuthn registration process. Finally, Section A.3 focuses on the procedures carried out during subsequent authentication runs. The flow of execution is almost identical to registration, with most differences being on the concrete information that is exchanged between the relying party, client and authenticator.

As mentioned, this description focuses on protocol details that are relevant to our work, and therefore we consider other aspects such as transport mechanisms, or less common use cases of WebAuthn, to be out of scope. The interested reader can consult the full description of the protocol in the specification [9].

A.1. Registration

In WebAuthn, registration is the process of creating a new *credential* (i.e., a new signing key pair) in a user's authenticator, bound to the corresponding relying party and user. The authenticator then returns the *public* part of this credential (the *credential public key*, which we also refer to as *public key*), which is saved by the relying party for later authentication

runs. A registration ceremony starts with a user requesting to register with the relying party. The relying party application gathers all the necessary information for the credential creation and calls navigator.credentials.create, which ”repackages” the information to be sent to the authenticator and then returns the credential creation information back to the relying party, where it is validated and stored, if successful.

A.1.1. Request

When a user performs a registration request, maybe to create a new account or to add a passwordless form of authentication to a pre-existing one, the relying party application starts gathering the necessary information that must be sent to the authenticator, through the navigator.credentials.create method, to create a new *credential*. This method expects an options object of type CredentialCreationOptions (section 5.1.1 from [9]), which contains a field publicKey of type PublicKeyCredentialCreationOptions, shown in Fig. A.1[1], where the information required for the authenticator is gathered by the application, such as relying party and user information, the cryptographic challenge, the types of signature algorithms that are accepted and information about attestation requirements.

We detail the most important fields of PublicKeyCredentialCreationOptions next, and how the relying party application chooses its values depending on its own criteria. Some fields are not detailed here due to being out of scope. The full description of every field can be consulted in Section 5.4 of the specification [9].

```
dictionary PublicKeyCredentialCreationOptions {
    required PublicKeyCredentialRpEntity         rp;
    required PublicKeyCredentialUserEntity       user;

    required BufferSource                             challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

    unsigned long                                timeout;
    sequence<PublicKeyCredentialDescriptor>      excludeCredentials = [];
    AuthenticatorSelectionCriteria               authenticatorSelection;
    sequence<DOMString>                          hints = [];
    DOMString                                    attestation = "none";
    sequence<DOMString>                          attestationFormats = [];
    AuthenticationExtensionsClientInputs         extensions;
};
```

Figure A.1: The PublicKeyCredentialCreationOptions dictionary from the WebAuthn specification (section 5.4).

The rp field contains information about the relying party, where rp.id is the relying party identifier (RP ID) that will be permanently associated to the generated *credential* in the

[1]Some fields of publicKey are also dictionaries defined in the specification. For example, rp is of type PublicKeyCredentialRpEntity, which is composed of two strings name and id.

authenticator (and thus provided by the relying party in further authentication runs), and will usually be set as its *effective domain*[2] of the relying party, and rp.name is the relying party's "human friendly" display name, which may be set equal to the RP ID.

The user field identifies the user account requesting the registration, and consists of a name, displayName and id, all mandatory fields. Both name and displayName are "human-friendly" names, with name being used as an identifier for the user account (e.g. an email or username) and displayName being purely for display (e.g. the first and last name of the user). The id field (also refered to as *user handle*) is a byte sequence with at most 64 bytes, not meant to be displayed, which uniquely identifies the user account, and is chosen by the relying party during registration.

The challenge is a *unique* randomly generated stream of at least 16 bytes (to make guessing infeasible) that must be signed by the authenticator to complete each registration (or authentication) run. Since it plays a critical role in the security of FIDO2, including in preventing replay attacks, its generation must be carried out by the relying party in a trusted environment (e.g. in the backend), without depending on the client.

In pubKeyCredParams, the relying party lists its supported signature algorithms using their COSE (see [59]) algorithm identifier in decreasing order of preference. Each element in this list is of type PublicKeyCredentialParameters, containing a field type, which has only one possible value of "public-key", and a field alg, which contains the COSE identifier.

The relying party *may* also include an excludeCredentials list in publicKey, which is a blacklist of credentials already associated with the user account (bound to the user handle). This is used to prevent the user from registering a new credential for the same account with the same authenticator. If the authenticator contains one of the credentials in excludeCredentials, the client must request that the user use a different one, otherwise the registration ceremony fails.

The authenticatorSelection field is used by the relying party to specify certain requirements the authenticator must meet in the new credential creation, and has four fields: authenticatorAttachment, which if not omitted determines whether the authenticator must be a "platform" or "cross-platform" (roaming) authenticator; residentKey, which specifies if the new credential should be discoverable or not; requireResidentKey, a legacy field that is true

[2]The *effectiveDomain* corresponds to the domain part of the origin of the relying party. The *origin* is a combination of the web protocol, domain and port used to access a website. So, for example, if the relying party's origin is https://www.rp.com:443 (the complete URL), its effectiveDomain would be www.rp.com.

if residentKey is set to ”required” and false otherwise; and userVerification, which determines whether the user must provide some form of user verification (e.g. PIN, biometric) when creating the credential. The relying party can set residentKey to ”discouraged”, ”preferred” or ”required”. When set to ”discouraged”, it is informing the authenticator that creating a non-discoverable (server-side) credential is preferable, although not enforced. When set to ”preferred” or ”required”, it specifies that the authenticator should create a discoverable credential, with the former allowing the creation of a non-discoverable credential when necessary (i.e., the authenticator not supporting discoverable credentials) and the latter forcing the client to return an error when the authenticator cannot create a discoverable credential. The userVerification field works in a similar way, where ”discouraged” means the relying party does not want user verification, ”preferred” means the authenticator must provide user verification if possible, although it will not be enforced, and ”required” means the registration process will fail if the authenticator did not provide user verification. By default, the value will be set to ”preferred”.

The attestation field specifies the relying party's attestation conveyance preference, i.e., how and where the attestation statement must be produced. It can have four values: ”none”, ”indirect”, ”direct” and ”enterprise” (see Section A.2 for more details).

The relying party *may* also showcase its preference on the attestation format produced by setting attestationFormats to different values such as, for example, ”packed”, ”tpm” or ”apple”[3].

Finally, timeout *may* be used by the relying party to set a limit, in milliseconds, on the time it expects the registration process to complete. It is, however, not enforced, and may be modified by the client if the value is not within a reasonable interval.

A.1.2. The create method

After preparing the options object with all the necessary information, the relying party application calls navigator.credentials.create, passing control of the execution to the client. This is where the information provided by the application and the response from the authenticator is validated, processed and forwarded. The full procedure can be consulted in Section 5.1.3 of the specification [9].

[3] A complete list for the types of attestation formats can be consulted in `https://www.iana.org/assignments/webauthn/webauthn.xhtml`.

Sending information to the authenticator

When executing navigator.credentials.create, the client first performs several checks on the information provided by the relying party, to ensure that it is valid before being sent to the authenticator. It starts with publicKey.timeout, replacing it with a client default timeout value if it falls outside its expected values or is not defined. Next, it checks publicKey.user.id, to ensure it is present and has at most 64 bytes in length, throwing an error otherwise, and the relying party identifier publicKey.rp.id. This field is verified to be the same as, or a suffix of, the *effectiveDomain* of the relying party (unless the client supports related origin requests[4]), or automatically set to the relying party's *effectiveDomain* if omitted.

The client then proceeds to create a new list of tuples (type,alg) called credTypesAndPubKeyAlgs, representing all the relying party supported signature algorithms for the authenticator. This step essentially filters out all algorithms in publicKey.pubKeyCredParams that have type set to anything other than "public-key", adding only the valid ones to credTypesAndPubKeyAlgs. If no algorithm has type "public-key", the process throws an error. If publicKey.pubKeyCredParams is empty, then credTypesAndPubKeyAlgs is initialized with tuples ("public-key",-7) and ("public-key",-257), corresponding to ES256 and RS256 respectively.

Next, a new structure called collectedClientData (see Fig. A.2) is created with, among other information, a field type identifying the type of request, set with "webauthn.create" for registration ceremonies, and a field challenge containing the base64url encoding of the challenge issued by the relying party, found in publicKey.challenge. This structure is then serialized into a JSON-compatible format and hashed using SHA-256 to produce a digest called clientDataHash.

```
dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    boolean                     crossOrigin;
    DOMString                   topOrigin;
};
```

Figure A.2: The CollectedClientData dictionary from the WebAuthn specification (section 5.8.1).

[4]If the client supports related origin requests, it may be possible for the relying party (depending on its policy) to set publicKey.rp.id with a value that is not a suffix or equal to its *effectiveDomain*. This allows, for example, a relying party that has multiple *effectiveDomains* for different countries to use a unique id for all of them.

At this point, the client starts actively searching for available authenticators. When an authenticator is detected on the user's machine, it's capabilities are checked against the requirements specified by the relying party in publicKey.authenticatorSelection to determine whether it can be used to perform the registration. It starts by confirming whether the authenticator's attachment modality ("platform" or "cross-platform") is the same as the modality specified in authenticatorAttachment. Then, it checks if the authenticator can store client-side discoverable credentials and perform user verification, and compares that information with the requirements in the authenticatorSelection fields residentKey, requireResidentKey and userVerification, such that if it cannot perform one of these functions, and they are required by the relying party, then it must not be chosen and the search for other authenticators must continue.

After this step, the client defines two boolean variables requireResidentKey and userVerification (not to be confused with the values specified by the relying party). These are the definitive flags that will be sent to the authenticator to determine if credentials must be *discoverable* or if user verification must be provided. Each variable is set to true *only* in one of two situations: (i) if the relying party specified its respective preference as "required", or (ii) if it set the preference as "preferred" *and* the authenticator supports the respective functionality (if authenticatorSelection.residentKey is not present, then requireResidentKey inherits from authenticatorSelection.requireResidentKey).

The client proceeds to define another boolean variable called enterpriseAttestationPossible, which is set to false except when publicKey.attestation is set to "enterprise" *and* the client supports enterprise attestation for this relying party.

Next, two new lists attestationFormats and excludeCredentialDescriptorList are created, the former being initialized as ["none"], if publicKey.attestation is "none", and to the value of publicKey.attestationFormats otherwise, and the latter being set to the list of credentials to exclude specified by the relying party in publicKey.excludeCredentials. In most scenarios, the client will also set a variable requireUserPresence as true, which informs the authenticator that it *must* require proof of user presence, by for example requiring the user to press a button, when creating the new credential.

After processing this information, the client calls the authenticatorMakeCredential function on the selected authenticator, passing as parameters clientDataHash, publicKey.rp,

publicKey.user, requireResidentKey, requireUserPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, enterpriseAttestationPossible and attestationFormats[5], and adds the authenticator (handle) to issuedRequests, a list that contains all authenticators to which an authenticatorMakeCredential call has been made.

Note that the client is always searching for authenticators until it reaches a timeout (defined by its lifetimeTimer). If at any point during this procedure the client or the user aborts a registration or authentication ceremony, either due to a timeout, an explicit cancelation from the user, or simply an error returned from the authenticator, the client will usually send a command for each authenticator in issuedRequests (i.e., that have been sent a command via authenticatorMakeCredential of authenticatorGetAssertion) to cancel its operations by calling authenticatorCancel (if the authenticator has not already returned an error) and will remove it from issuedRequests[6].

Processing the authenticator's response

An authenticator that successfully completes an authenticatorMakeCredential execution returns an attestation object (detailed later), which contains all the relevant information produced by the authenticator, including the newly created public key credential and a *possible* attestation signature on that credential.

The client's role, upon receiving the authenticator's attestation object, is to process this response and return it to the relying party. To do this, it first checks the relying party's attestation conveyance preference (which is set in publicKey.attestation) against the attestation provided by the authenticator. If it requested "direct" or "enterprise", then the client doesn't modify the attestation object at all. If, instead, it requested "indirect", the client *may* replace the attestation statement produced by the authenticator with an anonymized one, although this is not strictly required. Finally, if the relying party requested "none", the client *must* replace any attestation information that may identify the authenticator.

Next, it creates a new object pubKeyCred, of type PublicKeyCredential, the structure that contains all the information to be returned to the relying party application. This object

[5]The authenticatorMakeCredential also receives an extensions map, which includes the requested extensions from the relying party application. We don't expand on these since they are always optional and are out of scope for this work, but redirect to Section 9 of the specification [9] for a complete description of these features.

[6]The only situations where a failure does not result in *every* authenticator having its operations canceled is when an abort happens due to the authenticator being disconnected from the user's machine or when it returns an error other than "InvalidStateError", in which case only that authenticator is removed from issuedRequests.

contains the new public key credential identifier, a field authenticatorAttachment identifying the authenticator attachment modality ("platform" or "cross-platform"), and a field response, which is of type AuthenticatorAttestationResponse. This response will include an attestationObject field with the serialized attestation object returned from the authenticator and a clientDataJSON field with the JSON-serialized collectedClientData (which contains the challenge and type of request "webauthn.create").

This resulting pubKeyCred is the return value of navigator.credentials.create, which hands over control back to the relying party application, right after cancelling all operations on pending authenticators in issuedRequests and removing them from this list.

A.1.3. Validating and storing the credential

In the final stage of the registration ceremony, the relying party verifies all the information returned from navigator.credentials.create in pubKeyCred, if the credential creation was successful, and stores the credential public key for future authentication ceremonies. If the credential creation process failed, the relying party must abort and display an error to the user, proceeding in accordance to the type of error.

The pubKeyCred object, as previously mentioned, has a response field that contains all the information returned by the authenticator in response.attestationObject and the encoded JSON-serialized response.clientDataJSON. The relying party application must first decode and JSON-deserialize the latter to obtain the same collectedClientData structure defined in navigator.credentials.create, so that it can verify if collectedClientData.challenge is the same as the base64url encoding of publicKey.challenge, collectedClientData.type is "webauthn.create" and the collectedClientData.origin is correct[7].

Next, the response.attestationObject is deserialized to obtain the attestation object that was produced by the authenticator (and possibly modified by the client). This object contains the attestation statement attStmt, which contains the attestation signature and certificate, the attestation statement format fmt and some authenticator data authData. The authData fields contains information such as the rpIdHash, which is the result of hashing the RP ID with SHA-256, some flags signaling information about the credential generation process

[7] See Section 13.4.9. of the specification [9] for details on how a relying party may validate a credential's origin. A simple scenario would be, for example, the relying party checking that the origin is the same as its URL.

and the new credential public key. The relying party checks that this credential was generated using one of the signature algorithms specified in publicKey.pubKeyCredParams, and then proceeds to verify the attestation signature, if one was provided by the authenticator.

This verification starts by confirming what attestation format the authenticator used, by checking fmt against the list of supported WebAuthn attestation formats from the Internet Assigned Numbers Authority (IANA). The verification procedure is dependent on the format, but generally speaking the attestation signature in attStmt is performed on authData and the SHA-256 digest of clientDataJSON, which includes the challenge, so the relying party must use this information on the verification procedure. Depending on whether the authenticator produced a self-attestation or not (introduced in Section A.2), the verification uses either the provided new credential public key or the public key in the attestation certificate present in the attStmt, under x5c. If the signature verification was successful, the relying party must also verify the trustworthiness of the signature, using the Public Key Infrastructure (PKI) to check if the certificate in x5c is part of the certificate chain of an acceptable root certificate[8].

The final step to complete the registration ceremony involves creating a new structure credentialRecord, which is the format in which the credential will be stored. This structure will contain, among other information, the credential identifier, which *must* not be longer than 1023 bytes, and *must* be unique, its type (”public-key”), the COSE credential public key which will be used to verify future authentication ceremonies, a flag indicating if user verification was performed, a signature counter signCount with the value of the signature counter in the authenticator data, and *optionally* the attestation object and the clientDataJSON value from the client’s response. This credentialRecord is stored in the user’s account to be used in further authentication ceremonies.

A.2. Attestation

During registration, the relying party *may* require the authenticator to provide some proof about its origin so that it can, for example, know whether it is dealing with a *hardware* authenticator or a software implementation, and act accordingly depending on its trust policy. In these scenarios, it requests the authenticator to provide an *attestation* signature on, among other data, the generated credential public key and the relying party-issued challenge. The general attestation flow, in most occasions, proceeds as follows: (i) the

[8]One possible source for a relying party to obtain these root certificates is the FIDO Metadata Service (https://fidoalliance.org/metadata/).

relying party determines, according to its own policy, if and how it expects to receive the attestation signature (the attestation conveyance preference) and its supported attestation formats; (ii) the client forwards the supported formats to the authenticator; (iii) the authenticator performs the attestation signature on the credential public key and challenge, taking into account the supported formats and its own capabilities; (iv) the client receives the attestation (possibly modifying it depending on the relying party's conveyance preference) and returns it to the relying party; (v) the relying party checks if the attestation provided is acceptable according to its own policy, verifies the signature and, if successful, stores the credential public key on the user's account.

Note that attestation is only present in registration. On an authentication run, a relying party already has at least one credential public key that can be used to authenticate the user, and will only ever accept an authentication attempt performed with a device that has previously registered one of those credentials and, therefore, has the corresponding unique credential private key.

A.2.1. Attestation Conveyance

As covered in Section A.1, the relying party can set a field publicKey.attestation in the publicKey object that is passed to navigator.credentials.create, specifying its preference regarding the attestation conveyance. This information informs the client on how the relying party expects to receive the attestation, and encompasses four possible values. If no attestation is required from the authenticator, then the relying party sets this field to ”none”. This information is passed to the authenticator by including only the element ”none” in the attestationFormats variable passed to authenticatorMakeCredential. If the relying party requires some form of attestation, but does not want to enforce that the attestation must come directly from the authenticator, then it may set the field to ”indirect”, signaling to the client that it may replace any attestation provided by the authenticator using, for example, an Anonymization CA for greater privacy. If, on the other hand, it does want to ensure that the attestation comes directly from the authenticator, without any changes from the client, the field is set to ”direct”.

In some scenarios, the relying party may specify an ”enterprise” attestation conveyance preference. This is usually associated with large organizations that wish to ensure their employees only register organization-issued authenticators on their accounts or that each registration of a credential can be linked to its respective authenticator (and therefore, its user). Because registration in this mode can reveal information that uniquely identifies

an authenticator (such as a unique serial number or identifier) and because this is a potential privacy issue, the client and the organization-issued authenticators must be configured with a pre-approved list of relying party IDs with which enterprise attestation can be performed. On the client side, this is controlled by the organization, who can add or update this list as part of its policy on any of its managed client platforms (i.e., supported browsers for its employees to use). On the authenticator side, the organization must work with the manufacturer to specifically include this list when acquiring a set of authenticators. This helps preserve the user's privacy when registering on relying parties with other domains not related to the organization in question, preventing them from obtaining an authenticator's unique identifiable information even when setting publicKey.attestation with "enterprise".

A.2.2. Attestation Object

When an authenticator returns the new credential public key and attestation signature, it does so in a structure called an attestation object, showcased in Fig. A.3. This structure has three components: the attestation statement format fmt, the attestation statement attStmt and the authenticator data authData.

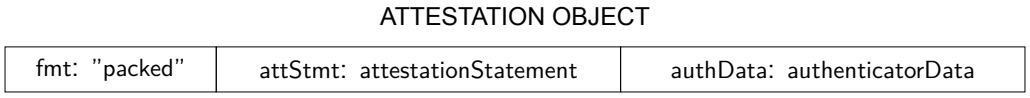


Figure A.3: Structure of the Attestation Object (adapted from Figure 6, Section 6.5 of the specification [9]).

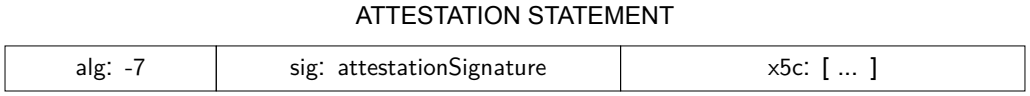


Figure A.4: Structure of the Attestation Statement (adapted from Figure 6, Section 6.5 of the specification [9]).

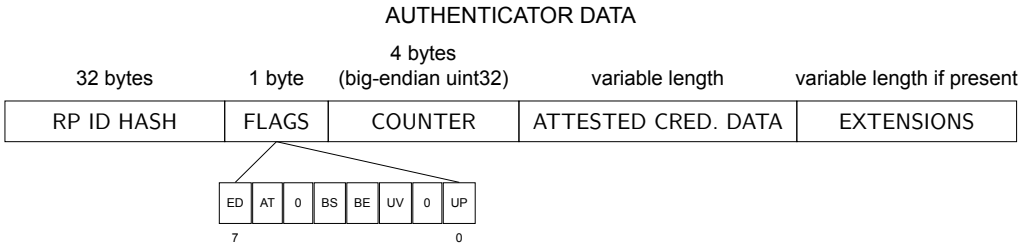


Figure A.5: Structure of the Authenticator Data (adapted from Figure 6, Section 6.5 of the specification [9]).

The simplest field is fmt, which just identifies the format (i.e., the structure) of attStmt.

The attestation statement attStmt, showcased in Fig. A.4, includes the COSE signature algorithm identifier alg, the attestation signature sig calculated on authData and clientDataHash, and a list x5c containing the attestation certificate and possibly its certificate chain (unless the attestation type is Self), to be verified by the relying party.

The authenticator data (authData) structure has at least 37 bytes in length, and conveys information to the relying party regarding the credential generation procedure[9]. Its first field is the rpIdHash, the SHA-256 digest of the relying party identifier (RP ID) to which the new credential is scoped. Then, it has a section flags, composed of 8 bits[10] where each bit signals some information about the credential and its generation. For example, if user verification and user presence were performed, then the UV and UP flags in authData will be set to 1. There is also a signature counter signCount, created the moment the credential is generated and incremented for each signature that is produced by the new credential in future authentications, a structure that represents the credential public key attestedCredentialData and some optional extensions (which are out of scope for this work).

The attestedCredentialData structure contains the authenticator's aaguid (Authenticator Attestation Globally Unique Identifier, which identifies the make and model of the authenticator), the credential identifier credentialId, the length of the credential identifier credentialIdLength and the COSE Key [60] encoded credential public key credentialPublicKey.

Because the attestation signature is generated on the authenticator data structure and the clientDataHash, the relying party can be sure that not only was the challenge signed along with the credential public key, but also that the information conveyed by the authenticator, including the flags informing if user presence and user verification were performed, is correct, as long as it deems the attestation signature trustworthy according to its own policy. The same is true on authentication ceremonies, because the signature produced by the authenticator using the credential private key always includes the authenticator data.

[9] Note that the authenticator data structure is also part of an authenticator's response during an authentication ceremony. In those scenarios, the attestedCredentialData field is absent, because the authenticator does not generate a new credential key pair that needs to be attested. Instead, it produces the rest of the authenticator data fields and uses a previously generated credential private key to sign them alongside the clientDataHash.

[10] In the most recent version of WebAuthn, only 6 are in fact being used to signal information to the relying party, with the remaining 2 being reserved for future use.

A.2.3. Attestation types

Attestation types dictate how and where an attestation signature is produced. In WebAuthn, there are five attestation types[11]: None, Self, Basic, AttCA and AnonCA. In this work, we are mainly interested in the first three, but nevertheless briefly describe every attestation type here for completeness.

None attestation The None type is straightforward, where no attestation signature is produced. This type may be used when an authenticator cannot produce an attestation signature, or when the relying party specifies ”None” as its conveyance preference, in which case the client removes any attestation that might have been produced by the authenticator before delivering the result to the relying party.

Self attestation The Self attestation type occurs when an authenticator does not have an attestation key pair, and therefore signs the authenticator data and client data using the newly generated credential private key. It is up to the relying party to determine whether this is acceptable or not.

Basic attestation In the Basic attestation type (also known as Batch attestation), an authenticator has an attestation key pair (more concretely, an attestation private key and certificate, which contains the public key) included by the manufacturer. As mentioned in Chapter 3, this key pair is shared by many authenticators of the same make and model, with the goal of providing sufficient anonymization. This attestation signature is produced using the attestation private key, and the attestation certificate is provided to the relying party in the attestation statement, to be verified through the PKI.

AttCA attestation If an authenticator is based on a TPM (Trusted Platform Module), it can hold what is called an endorsement key, a unique key provided by the manufacturer that is used to communicate with trusted third parties called Attestation CAs. If this is the case, then this authenticator can produce attestation signatures of type AttCA, where it generates attestation identity key pairs (AIK) and asks an Attestation CA to emit a certificate for each pair. This certificate is returned to the relying party in the attestation statement and used to validate the attestation signature. Crucially, this means that a TPM-based authenticator is not restricted to having only *one* attestation key pair, and that multiple

[11] Unlike the attestation formats, an attestation type is not explicitly written in any data returned from the authenticator, and thus the relying party differentiates between the different attestation types during the signature verification.

registrations with the same authenticator may use different attestation keys, certified by an Attestation CA using that authenticator's unique endorsement key.

AnonCA attestation Finally, the AnonCA type aims at providing a form of attestation that is very similar to AttCA, but with a stronger focus on privacy. In this mode, attestation key pair certificates are issued by an Anonymization CA on a per-credential basis, with any information that uniquely identifies the authenticator or its batch, such as its aaguid, omitted from the certificate that is returned to the relying party. This contrasts with AttCA, where the certificate may, for example, reveal the manufacturer of the authenticator through the certificate's subject, potentially disclosing additional information to the relying party about the origin of the authenticator.

An important detail regarding the Basic, AttCA and AnonCA types of attestation, which is stressed in the specification, is that all three types have the same structure in the response, defined by the attestation format. Therefore, the relying party will generally determine which type of attestation it received through the contents of the certificates themselves.

A.2.4. Attestation Statement Formats

The relying party can use publicKey.attestationFormats to inform about its preferred attestation statement formats. This list is passed to the authenticator to generate an attestation statement in one of those formats, if possible. However, it is *optional*, and the authenticator may produce attestation statements that are not in one of the specified formats.

Each format defines the syntax of the attestation statement, i.e., how this section of the attestation object will be structured, and specifies how the signature and verification procedures must be performed, although every format must include the authenticator data structure and the SHA-256 digest of the client data (clientDataHash). For example, the "packed" format (Fig. A.4), well-suited for hardware devices with limited resources (see Section 8.2. of the specification [9]), specifies that the attestation statement must contain three components: the signature scheme, the signature itself and the certificate (and certificate chain if applicable). The signing procedure concatenates the authenticator data and the clientDataHash and produces a signature using the credential private key if the attestation type is Self, or an attestation private key, if the type is Basic or AttCA, in which case the certificate (or certificates of the same chain) are added to x5c. The verification

procedure is similar, where the signature is verified over the concatenation of the authenticator data and clientDataHash, and then, if the attestation type is not Self, the certificate chain is validated through the PKI.

The attestation statement formats are ultimately chosen by the authenticator, and one format may be used to present attestations of different types. It is up to the relying party to decide, upon receiving an attestation statement, if it must accept or reject the attestation signature based on its policy.

A.3. Authentication

The authentication ceremony is the process of using a previously registered *credential* to authenticate the user. The process is similar to the registration ceremony, where upon receiving a user's request to authenticate, the relying party gathers all the necessary information (including a new random challenge) and sends it, via the navigator.credentials.get method, to an authenticator used in a previous registration to be signed using the credential private key that was generated. This signature is called an *assertion* signature, and is verified by the relying party using the previously stored credential public key. Unlike an *attestation* signature, an *assertion* signature is mandatory.

A.3.1. Request

When a user requests to authenticate with the relying party, to login to its account for example, the relying party gathers the information needed to obtain an authentication assertion from the user's authenticator, so that it can validate its identity. The process is analogous to the registration request, where this information is compiled in a publicKey object of type PublicKeyCredentialRequestOptions, seen in Fig. A.6, which is a field included in the CredentialRequestOptions object passed to navigator.credentials.get.

As was the case in Subsection A.1.1, we explain the most important fields here, and point to Section 5.5 of the specification [9] for a complete description of PublicKeyCredentialRequestOptions.

The rpId field, which is optional, refers to the RP ID, which may have been first defined in the registration in the publicKey.rp.id field, and must be an exact match to the RP ID associated with the credential that will be used to produce the assertion signature. If left empty, this field is set with the relying party's effective domain.

```
dictionary PublicKeyCredentialRequestOptions {
    required BufferSource                challenge;
    unsigned long                        timeout;
    DOMString                            rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
    DOMString                            userVerification = "preferred";
    sequence<DOMString>                  hints = [];
    AuthenticationExtensionsClientInputs extensions;
};
```

Figure A.6: The PublicKeyCredentialRequestOptions dictionary from the WebAuthn specification (section 5.5).

The allowCredentials field is a list where the relying party generally includes all credential identifiers[12] associated with the user's account, provided the user has already been identified at this point. This is the mechanism through which the relying party provides non-discoverable credentials to the authenticator. If the user has not yet been identified, then this list may be empty, which means only discoverable credentials scoped to this relying party may be used to create the assertion signature.

The userVerification, timeout and challenge fields are identical to their counterparts in PublicKeyCredentialCreationOptions.

A.3.2. The credentials.get method

After gathering all the information, the relying party application calls navigator.credentials.get, passing control to the client. The full procedure can be consulted in Section 5.1.4.1 of the specification [9].

Sending information to the authenticator

Most steps regarding the verification and processing of the relying party information are similar to the steps in navigator.credentials.create. Fields such as publicKey.timeout and publicKey.rpId are exactly the same, and thus follow the same rules. The process of creating the collectedClientData is also identical, with the exception of the type field, which is now set with "webauthn.get". This object is then JSON-serialized and hashed with SHA-256 to produce the clientDataHash. When these initial verification steps are performed, the client starts looking for authenticators. The process is similar to registration, where the client employs some criteria based on the information provided by the relying party to assess if a candidate authenticator can be chosen to perform the authentication or not.

[12] More concretely, each member of allowCredentials is of the type PublicKeyCredentialDescriptor, which is also the type of objects in excludeCredentials, and contains three fields: type, which is always "public-key"; id, which is the credential identifier; and transports, which identifies the transport mechanism (usually "usb", "nfc" or "ble") that might be used by the authenticator that created that credential.

Much like in the create method, in navigator.credentials.get the client defines a userVerification boolean value where it is set to true if the relying party defined publicKey.userVerification is not ”discouraged” and the authenticator supports user verification, and to false otherwise. If it is set as ”required”, and the authenticator does not support it, then this authenticator must be discarded.

Next, if the publicKey.allowCredentials field is empty, the client calls authenticatorGetAssertion with parameters publicKey.rpId, clientDataHash and userVerification. Otherwise, it creates a variable allowCredentialDescriptorList, which is a filtered list containing the credential identifiers in publicKey.allowCredentials that exist in the current candidate authenticator[13]. If this filtered list contains no credential identifiers, then the authenticator must be discarded. Otherwise, it is passed to authenticatorGetAssertion, along with the other previously mentioned parameters. During the execution of authenticatorGetAssertion, if the list contains at least one identifier, the user must be prompted to choose which one to use. This is done through the authenticator, if it has some interface with which it can provide information to the user, or by the client if it doesn't.

Note that, as was the case with authenticatorMakeCredential, authenticatorGetAssertion is expected to execute with the boolean variable requireUserPresence always set to true if the assertion signature is to be returned to the relying party for validation, since the relying party always expects the UP flag in the authenticator data to be true.

Processing the authenticator's response

A successful response from an authenticator to the client contains the following information: (i) the identifier of the credential private key used to perform the assertion signature[14]; (ii) an authenticator data structure identical to the one specified in A.2.2, but without the attestedCredentialData field; (iii) the assertion signature itself, which is always performed on the concatenation of the authenticator data and clientDataHash (always in that order); and (iv) the user handle associated with the used credential private key.

The client creates a new PublicKeyCredential object pubKeyCred, which contains the credential identifier, the authentication attachment modality and the response field, of type

[13] This procedure is performed via what is called a ”pre-flight” command in CTAP2, which we explain in Appendix B.

[14] If allowCredentialDescriptorList has *exactly* one credential identifier, and the assertion signature was successfully produced, then the authenticator will not return the credential identifier of the credential private key, because the client already knows it and thus some bandwidth can be saved by not transmitting those extra bytes.

AuthenticatorAssertionResponse, that contains the clientDataJSON, the authenticator data, the assertion signature and the user handle.

As was the case in registration, the client cancels all pending operations on any other authenticator with which it might have started communicating and returns pubKeyCred back to the relying party.

A.3.3. Validation

When the relying party receives pubKeyCred from authenticatorGetAssertion, like in the registration ceremony, it verifies that all the information is valid, before considering the ceremony successful. First, it checks that the credential identifier in pubKeyCred is one of the identifiers in publicKey.allowCredentials, and checks if this credential identifier matches one of the credentials stored in the user account. If it does, then this will be the credential used for the verification of the assertion signature.

Next, it JSON-deserializes the clientDataJSON from pubKeyCred.response to obtain collectedClientData, and performs the same procedure as in registration, i.e., checks if the collectedClientData.challenge is the same as the base64url encoding of publicKey.challenge, if collectedClientData.type is ”webauthn.get” and if the collectedClientData.origin is correct. It also checks if the rpIdHash in the authenticator data (the same field present in the authenticator data in registration) is the same as the SHA-256 digest of the expected RP ID, and if the UP and UV flags are set, with the latter being enforced only if publicKey.userVerification was set to ”required”.

Then, the relying party verifies that the assertion signature is valid over the concatenation of the bytes of authenticator data and the hashed clientDataJSON, using for that the credential public key recovered from the user account. If the verification is successful, it checks the signature counter in the authenticator data against the counter previously stored for that credential in registration. If the counter in the authenticator data is larger, then it is considered valid. If not, then it may justify failing the authentication ceremony, depending on relying party policy, as it may be proof of authenticator cloning or malfunction.

If all verifications are successful, the signature counter for the used credential record is updated to the value of the signature counter in the authenticator data and the authentication ceremony completes successfully.

B. The CTAP2 protocol

Here, we present a more detailed description of the CTAP2 protocol, introduced in Chapter 3. As with the description of WebAuthn in Appendix A, reading this Appendix is optional, but it is nevertheless helpful in providing a more complete picture over how CTAP2 works. As with WebAuthn, it contains some repeated information introduced and explained in Chapter 3 for completeness.

Structure of this Appendix

The CTAP2 specification [10] describes the protocol in three ”layers”: an API[1] of 11 commands that can be invoked by a client to perform operations, the encoding of all messages exchanged between client and authenticator and the transport mechanisms supported to transmit those messages (USB, NFC or Bluetooth).

Because the work in this thesis was primarily focused on the CTAP2 protocol logic and cryptographic core, we provide a very basic description of the message encoding and transport mechanisms in Section B.1, and a more detailed description of the API in Section B.2. Furthermore, this work was based on version 2.1 of CTAP, known as CTAP 2.1 [33], and therefore the API description refers to this version of the protocol (and to CTAP 2.2, since this new version does not significantly change the API or the cryptographic core of the protocol). We provide a summary of the differences between the three versions in Section B.3.

B.1. Low-level encoding and transport

B.1.1. CBOR Message Encoding

Messages transmitted between CTAP2 peers are encoded using the CTAP2 canonical CBOR encoding form, a more restricted version of the CBOR (Concise Binary Object Representation [61]) format. The CBOR format is designed for encoders/decoders with limited resources (essentially a ”low resource” JSON format), and this version employed by CTAP2 reduces its complexity even further to lower the processing power, memory and bandwidth needed to process and transmit messages.

Some of the changes include restrictions on how many bits may be used to represent integer values, forcing all values to have definite length (the original CBOR format allows

[1] The specification uses this term as an abstraction to describe each operation as a function, with input parameters and return values.

for values of indefinite length) and imposing an ordering scheme on map keys. As is mentioned in the specification [10], both parties must reject messages that are not in the CTAP2 canonical CBOR encoding form.

Messages sent to the authenticator represent commands, and so they are structured by specifying the CTAP2 command in the first message byte (0x01 for authenticatorMakeCredential, for example) and an encoded (definite length) CBOR map in the following bytes, which contains the parameters the authenticator expects the invoked command to receive. Each key in the map is an integer value corresponding to the number associated to that parameter in the specification.

B.1.2. Transport Mechanisms

CTAP2 defines three data transport alternatives: USB Human Interface Device (USB HID), Near Field Communication (NFC), and Bluetooth Low Energy (BLE). Authenticators can implement more than one of these transport protocols. The YubiKey 5 NFC[2], for example, can be used on a computer through the USB interface and on a smartphone via NFC. Our work was focused on FIDO2 authenticators using the USB HID transport, so we succinctly describe it here. Other transport mechanisms encapsulate the application logic information (i.e., the CBOR payload) in different ways, but the payload itself is the same.

The protocol governing the rules for framing messages in the USB HID transport is called CTAPHID. The specification introduces three concepts when it comes to messages exchanged by a client and an authenticator: transactions, messages and packets. A *transaction* refers to the process of sending a request from the client followed by a response from the authenticator. Any transaction must always be started by the client and has to either be completed or aborted before another can take place. Likewise, a response can only be sent by the authenticator following a request from the client. Each request and response is called a *message*. Each message has at least one *initialization packet* (Fig. B.1) and, depending on message length, zero or more *continuation packets* (Fig. B.2). Every packet has the same size, but depending on whether it is an initialization packet or a continuation packet, the structure is different, and so is the maximum size for the payload being sent.

[2] https://www.yubico.com/pt/product/yubikey-5-nfc/

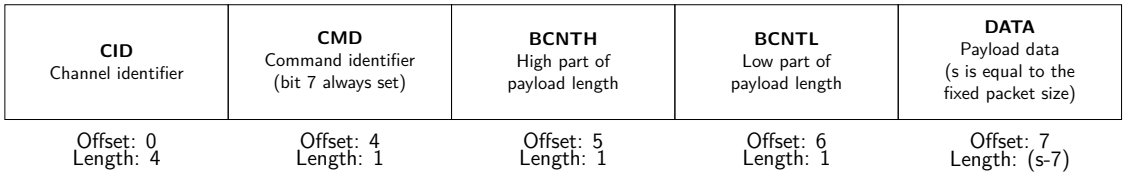


Figure B.1: Initialization packet structure (adapted from the table defined in Section 11.2.4 of the specification [10]).

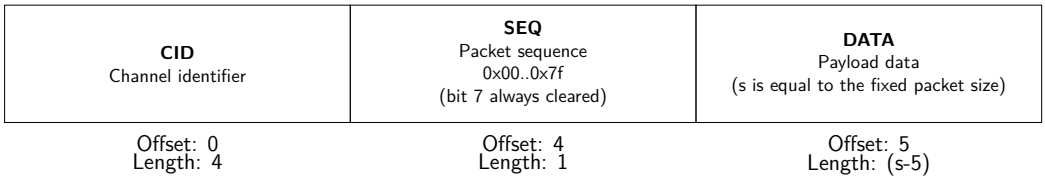


Figure B.2: Continuation packet structure (adapted from the table defined in Section 11.2.4 of the specification [10]).

In the initialization packet, there is an overhead of 7 bytes which are not used for storing the actual payload (for the CTAP2 commands, it would be the CBOR encoded data), but instead for conveying additional information pertaining to the CTAPHID protocol. The CID field, using 4 bytes, contains the unique channel ID negotiated with the authenticator and used for every message exchanged between the two parties. The CMD field, using 1 byte, refers to the CTAPHID command[3] being sent (not to be confused with the CTAP2 authenticator API commands). The BCNTH and BCNTL fields contain information on how long the message will be in total, so the recipient always knows how many packets to expect. Finally, the DATA field contains the first part of the message (if it is too large to fit in one packet).

The continuation packet has a simpler structure, with a slightly smaller overhead. It has the same CID field, but contains a new SEQ field which is a sequence number for each succeeding packet, starting at 0. The DATA field is the same except that now it is slightly larger because the packet has a smaller overhead.

To get a complete message, the receiver needs to assemble the entire CBOR serialized data from the various packets, and parse it to obtain the correct data structure. Both request and response messages have the exact same rules regarding packet structuring and sequencing, so this protocol is followed by both the client and the authenticator.

[3]A detailed list of the commands can be found in Section 11 of the CTAP2 specification [10].

B.2. Authenticator API

The CTAP2 authenticator API exposes the functionalities required to register and authenticate with a relying party, i.e., creating a new credential and producing a corresponding assertion signature. To access these functionalities, in the WebAuthn specification [9], the client calls authenticatorMakeCredential or authenticatorGetAssertion on the authenticator. The details regarding this communication are abstracted away in WebAuthn, but under the hood it involves a series of CTAP2 commands the client must send to the authenticator to securely transmit the necessary information.

Each command is invoked on the authenticator by including its unique identifier in the first byte of the payload (in the initialization packet), with parameters (if necessary) encoded in the following payload bytes. Both input parameters and command responses are structured as (definite length) maps, where keys are integer values in ascending order, starting in 1, and values may have different types. Every key is associated with a specific field. For example, the response from authenticatorGetInfo is a map where key 0x09 refers to the authenticator's supported transport mechanisms as an array of strings. This is an optional field to include in the response, and thus may be absent. If that is the case, key 0x09 must not be used for any other field, since the recipient of the message expects 0x09 to always refer to the supported transports.

At a lower level, CTAP2 commands make use of cryptographic primitives as part of their procedures. These primitives, as of CTAP 2.1, are exposed via the so-called PIN/UV Auth Protocol, an abstract low level API that is then instantiated with a specific implementation that is supported by the authenticator. The main purpose of the PIN/UV Auth Protocol is to provide both parties with the cryptographic tools necessary to create a secure channel and for clients to authenticate commands that are sent to the authenticator. This authentication is performed using a pinUvAuthToken, a cryptographic key which the client requests from the authenticator using a specific instantiation of the PIN/UV Auth Protocol.

In this Section, we describe the PIN/UV Auth Protocol, including the pinUvAuthToken, and introduce the four most important commands in the context of this work: authenticatorGetInfo, with command identifier 0x04; authenticatorClientPIN, with identifier 0x06; authenticatorMakeCredential, with identifier 0x01; and authenticatorGetAssertion, with identifier 0x02. We detail each command's most relevant parameters and response values, and refer to Section 6 of the specification [10] for a comprehensive description of every command.

B.2.1. The PIN/UV Auth Protocol

The PIN/UV Auth Protocol specifies two interfaces, one for the client platform and another for the authenticator. The client interface specifies 5 functions: initialize, encapsulate, encrypt, decrypt and authenticate. The authenticator interface specifies 8: initialize, regenerate, resetPinUvAuthToken, resetPersistentPinUvAuthToken (only in CTAP 2.2), getPublicKey, decapsulate, decrypt and verify.

Each function's implementation is provided in a concrete instantiation of the PIN/UV Auth Protocol. At the time of writing, CTAP 2.1 (and CTAP 2.2) specifies two: PIN/UV Auth Protocol 1 and PIN/UV Auth Protocol 2. Each instantiation maintains its own state, with internal variables such as the pinUvAuthToken and a P-256 ECDH public key. The decision on which instantiation will be used is negotiated between the client and the authenticator, depending on what each party supports. We briefly describe each instantiation next.

PIN/UV Auth Protocol 1

In CTAP 2.0, this was the only PIN protocol available (although support for more was already planned), and thus from CTAP 2.1 onwards it can be seen as a "legacy" PIN protocol.

Authenticator functions The initialize function calls regenerate and then resetPinUvAuthToken, and runs when the authenticator powers-up (i.e., plugging in a USB authenticator). The regenerate function samples a new random P-256 ECDH key pair, and resetPinUvAuthToken samples a new random pinUvAuthToken of length 16 or 32 bytes[4]. Both results are kept as the protocol's state. Whenever the authenticator needs to obtain its previously generated ECDH public key, it can do so via getPublicKey (note that this function *does not regenerate* the ECDH pair).

The decapsulate function receives a client's ECDH public key, performs the key derivation using the authenticator's own previously generated ECDH private key, passes the result's X-coordinate to an internal Key Derivation Function (KDF) function, which in PIN/UV Auth Protocol 1 is instantiated with SHA-256, and outputs the resulting 32-byte shared secret.

The decrypt function decrypts a ciphertext that is a multiple of the AES 128-bit block length (and fails if it's not) using AES-256 in CBC mode with *zero IV*.

[4]This is the only notable difference between CTAP 2.0 and CTAP 2.1 with PIN/UV Auth Protocol 1, because the former imposed no limits on the pinToken length, whereas the latter always imposes either 16 or 32 bytes.

The authenticate function returns the first 16 bytes of the HMAC-SHA-256 output calculated on a given input message and key, and verify computes the same digest and verifies that it matches with the tag received as input.

Client functions When the client runs initialize, it also generates an ECDH key pair, which is kept as state. However, unlike the authenticator, it does not generate a pinUvAuthToken.

The encapsulate function works exactly like decapsulate from the authenticator, with the exception that the output contains the client's ECDH public key as well as the shared secret[5].

The decrypt and authenticate functions are identical to the authenticator's, and encrypt is analogous to decrypt, where it encrypts plaintexts with a length that is a multiple of the AES block length using AES-256 in CBC mode with zero IV.

PIN/UV Auth Protocol 2

This instantiation is a new version introduced in CTAP 2.1 that provides different implementations for some functions, including the use of different primitives in some cases. This is usually the preferable protocol to be used if it is supported by both parties.

First and foremost, the resetPinUvAuthToken does not sample 16-byte pinUvAuthTokens, which means every pinUvAuthToken is now always exactly 32-bytes in length.

The internal KDF function is also changed, and now returns a shared secret key which is the 64-byte concatenation of two 32-byte HKDF-SHA-256 digests calculated over the X-coordinate of the ECDH derived key. The first digest is calculated over an *info* string "CTAP2 HMAC Key" and the other over "CTAP2 AES Key". Each digest is meant to be a 32-byte symmetric key, one for authentication and another for encryption. This contrasts with PIN/UV Auth Protocol 1, where the same 32-byte shared symmetric key may be used for encryption and authentication.

The encrypt function now receives the 64-byte shared secret, uses the rightmost 32 bytes (the AES key part) to encrypt the input plaintext using AES-256 in CBC mode with *random IV*, and returns the concatenation of the IV and ciphertext. The decrypt function is analogous, receiving the shared secret and the concatenated IV and ciphertext and returning the plaintext.

[5]Key Encapsulation Mechanisms (KEMs) usually define encapsulate as a function that receives a public key and outputs a ciphertext and a (symmetric) key. Because the encapsulate and decapsulate CTAP2 functions are implemented, internally, as a Diffie-Hellman key exchange, the client public key returned by encapsulate is considered the output "ciphertext".

With authenticate and verify, only the first 32 bytes of the input key are used, which allows these functions to receive two different types of authentication keys: the 64-byte shared secret derived from the ECDH key exchange, which is used when setting or changing a PIN, and the pinUvAuthToken, used to authenticate or validate commands. These functions also make use of the entire HMAC-SHA-256 digest, and not only the first 16 bytes.

B.2.2. The pinUvAuthToken

The idea behind CTAP2 is to ensure that a client can only issue certain commands to the authenticator, such as requesting credential creations or assertions, if authorized by the user (the owner of the authenticator who is trying to register or authenticate with the relying party). This authorization is given to the client usually through some form of user verification, such as a PIN (a simple password) or biometric information (e.g. a fingerprint reader). When the authenticator successfully verifies the user, it issues a key, called a pinUvAuthToken (which, as previously mentioned, is called pinToken in CTAP 2.0), which is used by the client to create an HMAC-SHA-256 tag that authenticates the command being sent to the authenticator.

The pinUvAuthToken exists as part of the state of a concrete PIN/UV Auth Protocol instantiation, and is one of the most important and central elements of CTAP2. At its core, it is a random 16 or 32 byte cryptographic key, used exclusively for producing HMAC-SHA-256 tags. Additionally, since CTAP 2.1 it also maintains several state variables, the most important of which are: (i) a set of permissions, which determines for which kind of commands this pinUvAuthToken may be used; (ii) the RP ID associated with the permissions set; (iii) a flag that determines if the pinUvAuthToken is in use; (iv) two flags userVerified and userPresent, which inform respectively if user verification and a test of user presence was performed when generating the pinUvAuthToken; (v) and a usage timer[6].

Since CTAP 2.1, the pinUvAuthToken is regenerated every time it is requested by a client, to ensure every pinUvAuthToken provided by the authenticator is unique (excluding the extremely low probability of pinUvAuthToken collisions).

Permissions

The existence of permissions associated with a pinUvAuthToken has only been introduced in CTAP 2.1. Whenever a client requests a pinUvAuthToken from the authenticator via the

[6]This variable is set to the value of initial usage time limit, a variable that sets the maximum time the pinUvAuthToken can stay in use, upper-bounded by a maximum value that is defined by the specification for each concrete transport mechanism (30 seconds for USB and Bluetooth and 19.8 seconds for NFC). Both variables are set during the generation of the pinUvAuthToken.

authenticatorClientPIN command, it *may* (depending on the command) additionally provide a list of permissions for the pinUvAuthToken to have[7]. The most important permissions in the context of this work are mc and ga, respectively for making a new credential and getting an assertion, but CTAP2 defines more permissions such as be (Bio Enrollment), which is used for example to add a fingerprint to the authenticator (similar to adding a PIN), or pcmr (Persistent Credential Management Read Only), which is used for managing discoverable credentials. A full list of the supported permissions can be consulted in Section 6.5.5.7 of the specification [10].

Note that the mc and ga permissions *require* the client to provide the RP ID with which they will be associated, such that the generated pinUvAuthToken can only be used to perform those operations for that specific relying party. If a client requests a pinUvAuthToken via the legacy getPinToken subcommand (see Subsection B.2.4) to a CTAP 2.1 or CTAP 2.2 authenticator, it cannot provide a list of permissions, because this subcommand does not support that parameter. In these situations (and only these), the pinUvAuthToken will have the mc and ga permissions set by default, and the associated RP ID will be the RP ID sent as a parameter in the first command authorized with this pinUvAuthToken.

Whenever a pinUvAuthToken is used on a command that requires a test of user presence, and the user presence is performed, the permissions associated with that pinUvAuthToken are revoked (with the exception of the lbw permission, which is out of scope for this work). This is especially important because it means a pinUvAuthToken *cannot* be used to authorize more than one authenticatorMakeCredential or authenticatorGetAssertion command.

The persistentPinUvAuthToken

When a client requests a pinUvAuthToken with only the pcmr permission, the authenticator returns a persistentPinUvAuthToken instead of a regular pinUvAuthToken. The difference between the two is that this persistent version, introduced in CTAP 2.2, can only be used to authorize read-only operations of authenticatorCredentialManagement, a command used to manage discoverable credentials. The persistentPinUvAuthToken is maintained between reboots (i.e., is not regenerated on initialize) and requests to obtain a new pinUvAuthToken, and is only regenerated when a user changes a PIN or sets a new minimim PIN length for the authenticator via the authenticatorConfig command.

While it is worthy of note, especially because it is one of the new additions from CTAP 2.2, the persistentPinUvAuthToken is not relevant for the work conducted in this thesis because

[7] As a general rule, the client should only request permissions for actions that are absolutely necessary.

it can never be used to authenticate other commands, which is why we don't expand on it further.

B.2.3. The authenticatorGetInfo command

This command is invoked to obtain information about the authenticator, such as what its capabilities are and what protocol versions it supports. This information is used to inform the client on what functionalities can be requested from the authenticator, through the parameters sent in the invoked commands. It also allows a client to assess if the relying party's requirements can be met by the target authenticator. For example, if the authenticatorGetInfo method signals that the authenticator does not support discoverable credentials, the client must use that information to decide whether this authenticator can be used to perform a registration, by checking if the relying party imposes discoverable credentials or not.

The authenticatorGetInfo command receives no parameters as input, and requires no permissions to be invoked. The response to this command contains many fields, of which only two are required: versions (0x01) and aaguid (0x03). The versions field is an array of strings of the form "FIDO_2_1" or "FIDO_2_0", which identifies the protocol versions supported by the authenticator, and aaguid is a byte string that refers to the unique authenticator make and model identifier, already mentioned in Appendix A.

The remaining fields are optional (with their absence either resulting in a default value being assumed or in being interpreted as a functionality that is not available), but some include important information that, if excluded from the response, might result in the client having a behaviour that is suboptimal by not taking advantage of the authenticator's full capabilities. Some important fields include: pinUvAuthProtocols (0x06), an array of unsigned integers specifying the supported PIN/UV Auth Protocol instantiations; algorithms (0x0A), an array of PublicKeyCredentialParameters (specified in WebAuthn) that specifies the authenticator's supported algorithms for credential generation; attestationFormats (0x16), an array of strings specifying the supported WebAuthn attestation formats; the aforementioned transports (0x09); and options (0x04), a map with string keys and boolean values where each key signals the presence or absence of a particular functionality. Some of the most important members of options are plat, rk, up, uv, clientPin, pinUvAuthToken and ep, where plat, which defaults to false, specifies if the authenticator is a platform authenticator, rk, which also default to false, signals the support for discoverable credentials and up, which defaults to true, signals the support for testing for user presence.

The clientPin option informs if the authenticator is capable of receiving a PIN from the client. When it is present *and* set to true, the functionality is supported *and* has been set. When it is false, it is supported but has not been set. If clientPin is absent, then the authenticator does not support receiving a PIN from the client. The ep option specifies if the authenticator supports enterprise attestation *and* if it is enabled, following the same logic as clientPin. The uv option refers to the authenticator's built-in user verification capabilities (not input by the client), such as fingerprint reading. If absent, then this functionality is not supported. If present, then it is supported and enabled if set to true and disabled if set to false. The pinUvAuthToken option refers to the capabilities of the authenticator regarding permissions on the pinUvAuthToken. If it is false or absent, then obtaining a pinUvAuthToken with permissions is not supported.

B.2.4. The authenticatorClientPIN command

The main purpose of this command is to obtain the pinUvAuthToken from the authenticator, which is the functionality most relevant for this work, although is is also used to set or change the PIN, as well as getting information on the number of PIN or UV retries left in the authenticator. The multiple functionalities of authenticatorClientPIN can be triggered by including the intended *subcommand* as a parameter, which will inform the authenticator on what operation the client is requesting.

This command receives a maximum of 8 parameters: the client selected pinUvAuthProtocol, the requested subCommand, the client ECDH public key keyAgreement, a tag pinUvAuthParam, the encrypted PIN newPinEnc, the encrypted hashed PIN pinHashEnc, a permissions field permissions and the relying party identifier rpId to which the permissions will be assigned. Of these parameters, which we explain next, only subCommand is always mandatory, since it is essential information the authenticator must receive in order to know which operation is being requested. Other parameters may be mandatory only when calling authenticatorClientPIN with certain subcommands, such as newPinEnc, for example, which is mandatory only when authenticatorClientPIN is being used for changing or setting a new PIN.

We explain the most important operations that can be requested next, determined by the chosen subcommand, and the role each authenticatorClientPIN parameter plays in the process.

Getting a shared secret

Every operation that requires the client to send secret information to the authenticator, or the other way around, must ensure this information is encrypted, because the communication channel between these two parties is not secure by default. To achieve this, the two parties must first perform a key exchange to derive a shared symmetric secret key, and then use this key to encrypt (and in some cases also authenticate) the information to be sent.

The subcommands used to set or change the PIN, and for obtaining the pinUvAuthToken, must always be preceeded by this subcommand. When setting (or changing) the PIN, the shared secret is used to encrypt and authenticate the new PIN (and the hash of the old PIN). When obtaining the pinUvAuthToken, the hash of the PIN may be sent to the authenticator (if the user verification mechanism chosen is the PIN) and, if that is the case, it must be encrypted with a shared secret key. The authenticator also needs this key to encrypt and send the pinUvAuthToken back to the client.

Because these subcommands require a shared secret key, whenever the client wants to send one to the authenticator, it must first send a authenticatorClientPIN command with the subCommand (0x02) parameter set as getKeyAgreement (subcommand code is also 0x02) to the authenticator. When obtaining a shared secret, the authenticatorClientPIN command does not require any other parameters, and the authenticator's response will include only the KeyAgreement field (0x01), with its own COSE_Key encoded Elliptic-Curve Diffie-Hellman public key. The client, upon receiving the authenticator's response, runs encapsulate on the received ECDH public key to obtain the shared secret key and its own ECDH private key.

Note that this subcommand is only used to obtain the authenticator's public key, and that at this point the client's public key has not yet been sent.

Setting or changing the PIN

The first step the client performs when setting or changing the PIN is actually collecting the PIN from the user, even before obtaining the shared secret. When in possession of both, the client can invoke authenticatorClientPIN to set or change the PIN by setting the subCommand parameter with setPIN (0x03) or changePIN (0x04), respectively. For both subcommands, the client must also set the pinUvAuthProtocol parameter to the same value used when obtaining the shared secret key, the keyAgreement parameter with its

ECDH public key derived in the previous step, the newPinEnc parameter with the encrypted PIN (padded to 64 bytes) and the pinUvAuthParam parameter with the authentication tag calculated on the newPinEnc ciphertext with the shared secret key[8]. For changePIN, the client must additionally include the pinHashEnc, which is the result of encrypting the 16 most significant bytes of the SHA-256 digest of the PIN with the shared secret key, and the authentication tag is calculated on the concatenation of newPinEnc and pinHashEnc.

When the authenticator receives this command with setPIN, provided the parameters are correct and the PIN has not yet been set, it runs decapsulate to obtain the shared secret key from the client's ECDH public key, and uses the key to verify the newPinEnc ciphertext against the pinUvAuthParam tag. If successful, it decrypts newPinEnc, unpadds the PIN and verifies if restrictions like length are respected. If everything is correct, the authenticator stores the 16 most significant bytes of the SHA-256 digest of the PIN and sets the pinRetries counter (which keeps track of how many incorrect PIN entries may be attempted before canceling access completely) to its maximum value[9], before signaling success to the client.

When the authenticator receives changePIN, the process is identical until the verification of pinUvAuthParam on the concatenation of newPinEnc and pinHashEnc. On success, the authenticator decrements pinRetries by 1, decrypts pinHashEnc, and checks if it matches with the stored hashed PIN previousy set. If it does, as with setPIN, the newPinEnc ciphertext is decrypted, hashed, the 16 most significant bytes of the digest are stored as the new current hashed PIN and the pinRetries counter is reset. Additionally, every existing pinUvAuthToken is also invalidated. If it does not, the authenticator calls regenerate, and then checks if pinRetries is 0 and if this is the 3rd consecutive PIN mismatch. The former permanently blocks the authenticator (which can only be unblocked by resetting the authenticator, wiping all stored credentials), while the latter forces a reboot (an interruption in power, such as removing the USB authenticator from the computer).

Obtaining the pinUvAuthToken

The client can obtain the pinUvAuthToken from the authenticator by invoking authenticatorClientPIN using one of three different subcommands: getPinToken (0x05), getPinUvAuthTokenUsingUvWithPermissions (0x06) or getPinUvAuthTokenUsingPinWithPermissions

[8] If operating with PIN/UV Auth Protocol 1, this is the same key used for encryption. If using PIN/UV Auth Protocol 2, encryption and authentication are performed with the different parts of the 64-byte shared secret key.

[9] The pinRetries counter may have, at most, a value of 8, but manufacturers may decide on a lower limit. The pinRetries counter is always reset to its maximum value after a successful PIN entry.

(0x09). The first subcommand is a legacy operation for obtaining the pinToken introduced in CTAP 2.0, while the other two are new operations introduced in CTAP 2.1 to obtain a pinUvAuthToken, with permissions, using biometrics or a PIN as the user verification mechanism, respectively.

As mentioned in Subsection B.2.3, the authenticatorGetInfo command may return three options: clientPin, uv and pinUvAuthToken. The client uses this information to figure out which subcommands the authenticator supports[10]. If pinUvAuthToken is present and true, then both CTAP 2.1 subcommands are supported as long as uv is present and true (for getPinUvAuthTokenUsingUvWithPermissions) or clientPin is present and true (for getPinUvAuthTokenUsingPinWithPermissions). If pinUvAuthToken is absent or false, then the client may only use getPinToken, provided clientPin is present and true.

When using the getPinToken subcommand, just as with changePIN, the client first collects the PIN and obtains the shared secret, and then invokes authenticatorClientPIN with the pinUvAuthProtocol, keyAgreement and pinHashEnc parameters. Upon receiving the request, the authenticator proceeds identically to the changePIN subcommand, running decapsulate to obtain the shared secret, decrypting the pinHashEnc ciphertext to obtain the hashed PIN and comparing it to the stored hashed PIN (while also decrementing the pinRetries counter before this verification). If the verification is successful, the authenticator resets the pinRetries counter and creates a new pinUvAuthToken by running resetPinUvAuthToken for every instantiation of PIN/UV Auth Protocol it supports. The pinUvAuthToken for the current pinUvAuthProtocol is then assigned the default permissions mc and ga (this happens only in getPinToken), activated (the in use flag is set to true), encrypted with the same shared secret used to encrypt the hashed PIN (independently of the pinUvAuthProtocol) and returned to the client. If there is a mismatch between the stored hashed PIN and the one received by the client, the error procedure is identical to changePIN.

The getPinUvAuthTokenUsingPinWithPermissions operation is similar to getPinToken, with the main difference being the possibility for the client to request different permissions for the requested pinUvAuthToken. To do this, it passes the additional permissions and rpId parameters. If the verification process is successful, the authenticator assigns the requested permissions to the newly generated pinUvAuthToken, and associates these permissions with the rpId, before encrypting it and sending it back to the client.

[10]When an authenticator supports the two CTAP 2.1 subcommands, the client must first attempt to use getPinUvAuthTokenUsingUvWithPermissions and fallback to getPinUvAuthTokenUsingPinWithPermissions if unsuccessful.

Finally, the getPinUvAuthTokenUsingUvWithPermissions subcommand works in a similar way to getPinUvAuthTokenUsingPinWithPermissions, but uses a built-in user verification mechanism instead of the PIN, which means the client does not include the pinHashEnc parameter. Instead, the authenticator attempts to verify the user via an internal procedure (e.g. fingerprint scan). The procedure has some nuances regarding how many failed verification attempts may be performed before returning to the client, and also how many times the client may attempt to invoke getPinUvAuthTokenUsingUvWithPermissions before falling back to another subcommand, but the most important aspect to note is the uvRetries counter, which plays a similar role to pinRetries. This counter has a maximum possible value of 25, and is decremented by 1 for every failed attempt, just like pinRetries. If it reaches 0, the client must use another subcommand that requires PIN verification. Contrary to the other two subcommands, consecutive failed verification attempts do not trigger a call to regenerate, and do not require a reboot.

One aspect worth noting is that the first two subcommands do not require the user to physically interact with the authenticator, and thus the pinUvAuthToken is activated with the userIsPresent flag set to false. However, when the pinUvAuthToken is obtained through built-in user verification, if that verification involves the user interacting with the authenticator, such as touching the fingerprint scanner, then proof of user presence is provided in this step and the userIsPresent flag is set to true.

B.2.5. The authenticatorMakeCredential command

This command is invoked by the client to create a new credential on the authenticator, as part of the process to register a user with a relying party, and corresponds to the authenticatorMakeCredential method mentioned in the WebAuthn specification [9].

Parameters

When invoking authenticatorMakeCredential, the client includes some or all of the following parameters: clientDataHash (0x01), rp (0x02), user (0x03), pubKeyCredParams (0x04), excludeList (0x05), extensions (0x06), options (0x07), pinUvAuthParam (0x08), pinUvAuthProtocol (0x09), enterpriseAttestation (0x0A) and attestationFormatsPreference (0x0B).

Note that most parameters match perfectly with the parameters specified in Section 6.3.2 of the WebAuthn specification, including in their typing, with the exceptions being options, pinUvAuthParam and pinUvAuthProtocol, which have no direct mapping. The pinUvAuthProtocol parameter refers to the chosen instantiation of the PIN/UV Auth Protocol. The

pinUvAuthParam parameter contains a tag that authenticates the command. Both parameters are therefore specific to CTAP2.

The options parameter is a map containing the values of the three WebAuthn flags requireUserPresence, requireUserVerification and requireResidentKey in options.up, options.uv and options.rk, respectively. However, there is a significant difference on how a client platform must behave depending on the authenticator CTAP2 version. On CTAP 2.0, for example, the options.up flag does not exist, because CTAP 2.0 authenticators are always expected to test for user presence. If this flag is provided in options, the authenticator returns an error. On the other hand, CTAP 2.1 authenticators may receive this flag on a authenticatorMakeCredential command, but will return an error if it is set to false. Regarding options.uv, it has been deprecated since CTAP 2.1, in favor of using pinUvAuthParam as a form of checking that user verification took place. Therefore, if the relying party requires user verification, the client targeting a CTAP 2.1 authenticator *should* obtain a pinUvAuthToken and use it to authorize the authenticatorMakeCredential command (which is the FIDO2 use case we consider in this thesis). If no user verification is required, the options.uv flag is set to false, and pinUvAuthParam is not sent to authenticatorMakeCredential.

Usage scenarios

The authenticatorMakeCredential command is fairly complex. Depending on authenticator and client functionalities, version differences and relying party preferences, it may successfully create a new credential without being authorized with a pinUvAuthToken. For example, an authenticator may not support user verification (authenticatorGetInfo returns neither uv nor clientPin as true). In this scenario, if the relying party does not require user verification, the client may set options.uv to false and still succeed in creating a credential. Another scenario is when an authenticator supports user verification, but returns the makeCredUvNotRqd option set to true in the authenticatorGetInfo response, which allows the creation of a new non-discoverable credential without performing user verification, if that is requested by the client platform (attempting to create a discoverable credential when user verification is supported with this option enabled causes an error). It is also possible for an authenticator to provide user verification during the authenticatorMakeCredential command itself, and not via the authenticatorClientPIN command, by performing the built-in user verification procedure (with fingerprint scanning for example) when the options.uv flag is set to true and no pinUvAuthParam is provided. This is a backwards

compatibility feature for CTAP 2.0 client platforms, and is not the preferred form of user verification for CTAP 2.1 or higher client platforms.

We don't expand on these use cases here, but redirect to Sections 6.1.1 and 6.1.2 of the specification [10] for a thorough description of client and authenticator procedures when invoking this command. Instead, we consider scenarios where: (i) an authenticator "is protected by some form of user verification" (i.e., has either uv or clientPin set to true in the authenticatorGetInfo response); (ii) the relying party sets the user verification as "required" or "preferred"; (iii) an authenticator performs user verification through the authenticatorClientPIN command by generating a pinUvAuthToken. In sum, we consider scenarios where authenticatorMakeCredential is authenticated with a pinUvAuthToken.

Before calling authenticatorMakeCredential

Even in scenarios where the relying party sets user verification as "required" or "preferred", and a user verification mechanism is enabled (a PIN has been set or fingerprint information has been enrolled), the client will perform differently depending on the concrete options provided by the authenticator in authenticatorGetInfo.

If the uv and pinUvAuthToken options are true, then the client invokes the getPinUvAuthTokenUsingUvWithPermissions subcommand. If uv is not true, but clientPin is, then the client will try to obtain it via getPinUvAuthTokenUsingPinWithPermissions. If the pinUvAuthToken option is not true, the client will either attempt to perform the user verification in the authenticatorMakeCredential command by setting the options.uv flag to true (which we don't consider here), or use the getPinToken subcommand to obtain the pinUvAuthToken, assuming clientPin is true.

Once the pinUvAuthToken is obtained, with the mc permission (and optionally with ga too), the client creates the pinUvAuthParam parameter by calling authenticate on clientDataHash with the pinUvAuthToken, authorizing the command to be invoked on the authenticator. The PIN/UV Auth Protocol instantiation that implements authenticate is the same that was used to obtain the pinUvAuthToken, and is passed as a parameter to authenticatorMakeCredential.

Authenticator operations

As was previously explained, we assume in this description that user verification is required and supported, and that it was performed in authenticatorClientPIN. Therefore, the client invokes authenticatorMakeCredential with a pinUvAuthParam parameter.

After receiving the request, the authenticator first checks that the pinUvAuthProtocol parameter is present and supported, returning an error otherwise, and then iterates through the pubKeyCredParams list, validating *every* algorithm present and choosing the first that is supported by the authenticator.

Next, it creates the response structure which includes, among other data, the attestation object (Appendix A, Section A.2), and initializes the UV and UP bits in the attestation object's authData to 0.

The authenticator then starts going through the options parameter. Assuming pinUvAuthParam is present as a parameter, the options.uv flag is always treated as being false, since the presence of pinUvAuthParam takes precedence. The options.rk option is assumed to be false when absent, meaning that unless this option is specifically set by the client as true, the authenticator will create a non-discoverable credential. Finally, the options.up is always assumed to be true when absent and, as mentioned before, if it is present and set to false the authenticator returns with an error.

Next, the authenticator checks the pinUvAuthParam parameter by calling the verify function from the selected pinUvAuthProtocol on the clientDataHash parameter using its pinUvAuthToken, and returns an error if the procedure fails. This is perhaps the most important step in the context of this work, since it is where the authenticator validates that this command to create a new credential was indeed invoked by an authorized client that underwent some form of user verification (but not necessarily user presence yet). If the validation succeeds, the authenticator also checks if the pinUvAuthToken has the mc permission, and if the RP ID associated with this permission (if any) is the same as the rp.id received as a parameter. If everything is successful, the UV bit in authData is set to 1.

The authenticator then checks the excludeList, and aborts if any of the listed credentials matches one of its own credentials, for the specified rp.id.

Next, it performs a test of user presence. When user verification is performed through authenticatorClientPIN, the pinUvAuthToken may have the userPresent flag set to true, if it was obtained by performing built-in user verification. In these scenarios, the user has already provided proof of user presence. On the other hand, if the client obtained a pinUvAuthToken using a PIN, for example, then proof of user presence has not been provided yet, and therefore must be provided here by, for example, touching or pressing a button on the authenticator. In both cases, a successful proof of user presence set the UP bit in

authData to 1.

As was mentioned before, the process of testing for user presence clears all permissions[11] from the authenticator, while also setting both userPresent and userVerified flags to false. This renders the pinUvAuthToken effectively useless, requiring the client to obtain a new one for authorizing new commands.

The authenticator then creates a new credential key pair, using the previously chosen algorithm from pubKeyCredParams, which will always be non-discoverable unless options.rk is set to true (making it necessary to have an explicit request from the client to create a discoverable credential). If the new credential is discoverable, any discoverable credential previously stored in the authenticator for the same rp.id and user.id will be replaced.

Finally, the authenticator checks the attestationFormatsPreference list. If there is only one entry with the ”none” format (i.e., the relying party did not require or prefer attestation), then no attestation is performed. Otherwise, the authenticator generates the attestation statement attStmt using the first format on the list that is supported, or a format of its choosing if no supported format is found. Recall that, if performed, the attestation signature is calculated on the authData structure and the clientDataHash parameter. Therefore, the relying party can trust, depending on its policy on types of attestation, that user presence and verification procedures were properly followed.

A successful authenticatorMakeCredential command ends with the authenticator returning the attestation object.

B.2.6. The authenticatorGetAssertion command

This command is invoked by the client to obtain an assertion signature from the authenticator, as part of the process for authenticating the user with a relying party, and corresponds to the authenticatorGetAssertion method mentioned in the WebAuthn specification [9].

Parameters

The client invokes authenticatorGetAssertion using some or all of the following parameters, which are also mostly the same as the ones defined in WebAuthn: rpId (0x01), clientDataHash (0x02), allowList (0x03), extensions (0x04), options (0x05), pinUvAuthParam (0x06) and

[11] To be more precise, it clears all permissions *except* for lbw, which is a permission necessary for invoking authenticatorLargeBlobs, a command used for storing additional data on the authenticator that is out of scope for this work.

pinUvAuthProtocol (0x07).

The pinUvAuthParam and pinUvAuthProtocol parameters are exactly the same as in authenticatorMakeCredential, and options is different only in that it no longer includes the rk flag and the options.up flag can be false, although it is true by default.

Usage scenarios

We consider the same kind of scenarios as in authenticatorMakeCredential, i.e., where the client authenticates a authenticatorGetAssertion command with a pinUvAuthToken, previously obtained using some form of user verification. However, there exists a ”special” way to use authenticatorGetAssertion which is specific to this command.

When performing a registration or authentication, the relying party always expects the test of user presence to be performed, and thus only terminates successfully if the authData UP bit is set to 1. In CTAP2, this is enforced in the authenticatorMakeCredential command, which returns an error if options.up is false, but is not enforced in authenticatorGetAssertion. The reason for this difference has to do with the so-called ”pre-flight” commands, which are authenticatorGetAssertion commands that have the options.up flag set to false and are used by a client to check the existence of specific credentials in an authenticator. Before invoking authenticatorMakeCredential, a client can invoke ”pre-flight” commands to check if any of the excludeList credentials exist in the authenticator. The same can be done before the authenticatorGetAssertion command with regards to the credentials in allowList. These assertions are never meant to be returned to the relying party, because they have the UP bit set to 0, but allow the client to figure out if the restrictions from excludeList or allowList are respected, and if the respective command has conditions to proceed.

The ”pre-flight” commands are also the reason why a client may request the ga permission for a pinUvAuthToken that will be used to authorize a authenticatorMakeCredential command.

Before calling authenticatorGetAssertion

The procedure the client performs to obtain the pinUvAuthToken and produce the pinUvAuthParam parameter before calling authenticatorGetAssertion is identical to authenticatorMakeCredential, with the exception that it now requests it with the ga permission.

Authenticator operations

The first steps when running the authenticatorGetAssertion algorithm are identical to authenticatorMakeCredential. The authenticator first checks if the received pinUvAuthProtocol parameter is valid (i.e., is present and is supported) and then creates a new assertion response structure, which contains *at least* the identifier of the credential used to performs the assertion, an authData data structure and the assertion signature itself. The UV and UP in authData are set to 0. Then it checks the options parameter to validate the received flags. If options.rk is present, it returns an error and terminates the process.

Next, it performs user verification, by calling verify to validate pinUvAuthParam, checking if the pinUvAuthToken has the ga permission and, if successful, setting the authData UV bit to 1. It also performs the test of user presence (if this is not a "pre-flight" command) and sets the UP flag to 1 if successful, before removing the pinUvAuthToken's permissions.

The next step involves the selection of the credentials in the authenticator that may be used to produce the assertion signature, which depends on allowList. If this parameter was included, and has at least one credential, then the authenticator fetches all the stored credentials that match the ones on the list and that are bound to the received rpId, and then selects one of them as the selected credential for performing the assertion signature. If it doesn't, it instead fetches all *discoverable* credentials, also bound to the same rpId, and the process for selecting the credential is more complicated.

If the authenticator has a display, it shows each credential to the user and its details such as the user account with which they are associated, and the user can choose which credential to use. If it does not have a display, then the authenticator selects the most recent credential, but prepares to produce additional assertion signatures at the client's request through a special command called authenticatorGetNextAssertion.

This stateful command is called by the client when it receives the numberOfCredentials field in the authenticatorGetAssertion response with a value greater than 1. It receives no parameters, and expects the authenticator to remember the state of the previous authenticatorGetAssertion or authenticatorGetNextAssertion commands, including an index to the position for the next credential. The idea of this command is to allow the client to produce an assertion signature for each discoverable credential associated with the received rpId, and then present all the options in its user interface for the user to select the one it wants to use. It is worth noting that with discoverable credentials, since only one can be stored

per relying party and per user handle, this process of selecting which credential to use in the client platform effectively selects which account the user will log into.

Regardless of the selection process, when the credential is chosen the authenticator uses it to produce the assertion signature over the clientDataHash and the authData structure, and returns the response to the client.

B.3. Differences between versions

As was previously mentioned, there currently exist three versions of CTAP2: CTAP 2.0, CTAP 2.1 and CTAP 2.2.

The core logic of the protocol was already well defined in CTAP 2.0, and the major use case we consider where a pinUvAuthToken (pinToken in CTAP 2.0) is used to authenticate a command is conceptually the same across versions. When CTAP 2.1 was introduced, it refined the protocol, adding new features which further cemented how the protocol should operate. The CTAP 2.2 version added minor changes to CTAP 2.1, none of which affect the results obtained in this work.

The three most important differences between CTAP 2.0 and the later versions are: (i) CTAP 2.0 pinTokens do not regenerate when a client requests a new one via getPinToken, instead only regenerating when a reboot occurs; (ii) CTAP 2.0 only has one PIN/UV Auth Protocol available, which basically corresponds to PIN/UV Auth Protocol 1 from CTAP 2.1 and later; (iii) and CTAP 2.0 does not support permissions, which means the getPinUvAuthTokenUsingPinWithPermissions and getPinUvAuthTokenUsingUvWithPermissions subcommands do not exist in this version, and that a pinUvAuthToken is not really bound to a specific command it can authorize.

Other minor changes from CTAP 2.0 include: (i) the deprecation of the options.uv flag in commands like authenticatorMakeCredential or authenticatorGetAssertion as a form of user verification, in favor of obtaining a pinUvAuthToken; (ii) the addition of new commands such as authenticatorCredentialManagement for managing credentials; (iii) the fixed length of the pinUvAuthToken to 16 or 32 bytes, whereas in CTAP 2.0 it has no limit; and (iv) a new AlwaysUV option that enforces user verification, and always flips the UV bit to 1 in the end, at the authenticator level, independently of the relying party's request.

Regarding CTAP 2.2, the only notable addition is the introduction of the persistentPinUvAuthToken and the related pcmr permission.

Feature	CTAP 2.0	CTAP 2.1	CTAP 2.2
pinUvAuthToken regenerates	Only on reboot	**Yes**	**Yes**
PIN/UV Auth Protocol	1	**1 and 2**	**1 and 2**
Permissions support	No	**Yes**	**Yes**
options.uv flag	**Yes**	**Deprecated**	**Deprecated**
Authenticator Credential Management	No	**Yes**	**Yes**
Limited pinUvAuthToken length	No	**Yes**	**Yes**
AlwaysUV support	No	**Yes**	**Yes**
Persistent pinUvAuthToken	No	No	**Yes**

Figure B.3: Key differences between the different versions of CTAP2.

C. CTAP 2.1 proof for PIN/UV Auth Protocol 1

We present a brief overview of the CTAP 2.1 security proof when instantiated with PIN/UV Auth Protocol 1, by outlining its differences from the CTAP 2.1 proof when using PIN/UV Auth Protocol 2. Most games remain unchanged, since they are not affected by which protocol is instantiated. Therefore, we focus only on Games 5, 9 and 10, which differ when CTAP 2.1 is instantiated with PIN/UV Auth Protocol 1.

Game 5 This game replaces all encrypted pins on Setup with constant values, and then also replaces all pinUvAuthTokens sent during Bind with random values only when using the same symmetric key used on Setup. When considering PIN/UV Auth Protocol 1, we can no longer reduce to the IND-1$PA-LHPC security of the underlying encryption scheme SKE, because during Setup the same symmetric key K is used to encrypt $\mathsf{H}(0000)$ into c_p and then to authenticate c_p by producing a tag t_p. Therefore, we must reduce to a variant of this security definition, which we call IND-1$PA-LHPC-M (defined in Chapter 2).

Game 9 This game eliminates collisions between pinUvAuthTokens that are sent to $\mathcal{A}$ in Execute or Send-Bind-T queries. When considering PIN/UV Auth Protocol 1, each pinUvAuthToken has size $\mu\lambda$, for $\mu \in \{1,2\}$, which means the upper bound for the probability that a collision between two pinUvAuthTokens occurs is $1/2^{\lambda}$. Therefore, we have $|\mathsf{Pr}_8 - \mathsf{Pr}_9| \leq (q_{\mathsf{E}} + q_{\mathsf{Send}})^2/2^{\lambda+1}$.

Game 10 In this game, each tag t corresponds to the leftmost λ bits from the 2λ-bit output of MAC, instantiated as HMAC-SHA-256. The reduction is the same, under the assumption that the truncated output of HMAC-SHA-256 is still a secure MAC.

D. IND-1$PA-LHPC Proof for CBC

Here we show our code-based proof of the IND-1$PA-LHPC security of CBC via a sequence of games. Our proof is based on the IND-1$PA-LPC proof for CBC from [2], with a few modifications to accomodate the differences in the two security definitions. Therefore, much like the authors in [2], we define CBC as an instance of a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, seen in D.1. We also define F as an invertible pseudo-random permutation $\mathsf{F} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$, with $\mathcal{K} := \{0,1\}^{f_1(\lambda)}$, $\mathcal{M} := \{0,1\}^{f_2(\lambda)}$ and $\mathcal{C} := \{0,1\}^{f_2(\lambda)}$ for polynomials f_1 and f_2. We model H as a random oracle $\mathcal{H}_1 : \{0,1\}^* \to \{0,1\}^{l_1}$, available to $\mathcal{A}$.

Theorem D.1. *For every efficient adversary $\mathcal{A}$ that makes at most q_{LHPC} and q_{RAND} queries to $\mathcal{O}_{LHPC}$ and $\mathcal{O}_{RAND}$, there exist adversaries $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$ such that:*

$$\begin{aligned} Adv^{IND\text{-}1\$PA\text{-}LHPC}_{CBC}(\mathcal{A}) \leq\ & Adv^{prp}_{F}(\mathcal{B}_1) \\ & + 1/2^{l_1} \\ & + \frac{q_{LHPC}}{2^{f_2(\lambda)} - 1 - q_{LHPC}} + q_{RAND} \lceil \frac{l_{max}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \\ & + Adv^{prp}_{F}(\mathcal{B}_2) \\ & + Adv^{IND\text{-}1\$PA}_{CBC}(\mathcal{B}_3) \end{aligned}$$

KG(1^λ):
1: $K \xleftarrow{\$} \mathcal{K}$
2: **return** K

Enc(K, m):
1: $x_1 \,||\, ... \,||\, x_n \leftarrow m$ s.t. $|\, x_i \,| = f_2(\lambda)\ \forall i \in \{1, ..., n\}$
2: $y_0 \leftarrow \mathsf{SetIV}()$
3: **for all** $i \in \{1, ..., n\}$ **do**
4: $\quad y_i \leftarrow \mathsf{F}(K, y_{i-1} \oplus x_i)$
5: $y \leftarrow y_0 \,||\, ... \,||\, y_n$
6: **return** y

Dec(K, c):
1: $y_0 \,||\, ... \,||\, y_n \leftarrow c$ s.t. $|\, y_i \,| = f_2(\lambda)\ \forall i \in \{0, ..., n\}$
2: **for all** $i \in \{1, ..., n\}$ **do**
3: $\quad x_i \leftarrow y_{i-1} \oplus \mathsf{F}^{-1}(K, y_i)$
4: $m \leftarrow x_1 \,||\, ... \,||\, x_n$
5: **return** m

Figure D.1: CBC instantiated as a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$, as defined in [2].

Let $\Pr_g$ denote the probability that game g outputs 1. Modifications applied to the game are showcased in orange. Specific events that may cause the challenger to abort if some

bad event happens are showcased in red, and are relevant only for the game where they are defined. Code that is specific to reductions is showcased in green.

Game 0

This is the original $\text{Expt}^{\text{IND-1\$PA-LHPC}}_{\text{CBC}}$. Therefore,

$$\text{Pr}_0 = \text{Adv}^{\text{IND-1\$PA-LHPC}}_{\text{CBC}}(\mathcal{A}) \tag{D.1}$$

Game 1

Let $\mathcal{F}$ be the set of all invertible random permutations $f : \{0,1\}^{\text{f}_2(\lambda)} \to \{0,1\}^{\text{f}_2(\lambda)}$.

In this game, the challenger uniformly samples a invertible random permutation in the beginning of the experiment. Then, whenever it executes $\text{Enc}(K, m)$ or $\text{Dec}(K, c)$, the computation of $\text{F}(K, \cdot)$ and $\text{F}^{-1}(K, \cdot)$ is replaced by $f(\cdot)$ and $f^{-1}(\cdot)$, respectively.

Game1($\mathcal{A}$):

$b \xleftarrow{\$} \{0,1\}$
$K \xleftarrow{\$} \text{CBC.KG}()$
$\text{state} \leftarrow 0$
$(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
$c_{\text{chal}} \xleftarrow{\$} \mathcal{O}_{\text{LR}}(m_0, m_1)$
$f \xleftarrow{\$} \mathcal{F}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\text{LR}}, \mathcal{O}_{\text{RAND}}, \mathcal{O}_{\text{LHPC}}}(c_{\text{chal}})$
return $(b = b')$

Figure D.2: Game 1.

Enc(K, m):

1: $x_1 \,||\, ... \,||\, x_n \leftarrow m$ s.t. $|\, x_i \,| = f_2(\lambda)\ \forall i \in \{1, ..., n\}$
2: $y_0 \leftarrow \text{SetIV}()$
3: **for all** $i \in \{1, ..., n\}$ **do**
4: $\quad y_i \leftarrow f(y_{i-1} \oplus x_i)$
5: $y \leftarrow y_0 \,||\, ... \,||\, y_n$
6: **return** y

Dec(K, c):

1: $y_0 \,||\, ... \,||\, y_n \leftarrow c$ s.t. $|\, y_i \,| = f_2(\lambda)\ \forall i \in \{0, ..., n\}$
2: **for all** $i \in \{1, ..., n\}$ **do**
3: $\quad x_i \leftarrow y_{i-1} \oplus f^{-1}(y_i)$
4: $m \leftarrow x_1 \,||\, ... \,||\, x_n$
5: **return** m

Figure D.3: Modifications to CBC - Game 1.

Advantage measure

The only modification in game 1 corresponds to the replacement of the underlying invertible pseudo-random permutation in the encryption scheme with an invertible, truly random permutation. Therefore, we can bound the advantage of $\mathcal{A}$ by considering an adversary $\mathcal{B}_1$ that can distinguish between f and $\mathsf{F}(K,\cdot)$ by invoking $\mathcal{A}$. Therefore:

$$\mathsf{Pr}_0 - \mathsf{Pr}_1 \leq \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{F}}(\mathcal{B}_1) \tag{D.2}$$

Game 2

In this game, the challenger aborts if $\mathcal{H}_1(m_b)$ is a prefix of, but not equal to, m_b.

Game2($\mathcal{A}$):
- $b \xleftarrow{\$} \{0,1\}$
- $K \xleftarrow{\$} \mathsf{CBC.KG}()$
- $\mathsf{state} \leftarrow 0$
- $(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
- $\mathsf{bad} \leftarrow \mathsf{false}$
- $t \leftarrow \mathcal{H}_1(m_b)$
- ▷ $\lceil \frac{|x|}{f_2(\lambda)} \rceil$ *is the number of blocks of size* $f_2(\lambda)$ *for some string* x.
- $n \leftarrow \lceil \frac{|t|}{f_2(\lambda)} \rceil$, $t_1 \,||\, .. \,||\, t_n \leftarrow t$
- **if** $n < \lceil \frac{|m_b|}{f_2(\lambda)} \rceil$ **then**
  - **if** $(t_1 \,||\, .. \,||\, t_n) = (x_1 \,||\, .. \,||\, x_n)$ **then**
    - $\mathsf{bad} \leftarrow \mathsf{true}$
    - abort
- $c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
- $f \leftarrow \mathcal{F}$
- $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}},\mathcal{O}_{\mathsf{RAND}},\mathcal{O}_{\mathsf{LHPC}}}(c_{\mathsf{chal}})$
- **return** $(b = b')$

Figure D.4: Game 2.

Advantage measure

We bound the advantage of $\mathcal{A}$ against game 2 by considering the bad event bad which happens exactly when m_b is at least one block longer than $\mathcal{H}_1(m_b)$ and $\mathcal{H}_1(m_b)$ is a prefix of m_b.

Since m_b's blocks are fixed in the beginning, for bad to occur, $\mathcal{H}_1$ must output exactly $x_1 \,||\, .. \,||\, x_n$, which happens with probability $1/2^{l_1}$.

Since games 1 and 2 proceed identically until bad, through the difference lemma, we have:

$$| \Pr_1 - \Pr_2 | \leq \Pr_2[\text{bad}] \tag{D.3}$$

$$\Pr_2[\text{bad}] \leq 1/2^{l_1} \tag{D.4}$$

Therefore,

$$| \Pr_1 - \Pr_2 | \leq 1/2^{l_1} \tag{D.5}$$

Game 3

In this game, when $b = 0$ and $\mathcal{A}$ delivers a ciphertext c to $\mathcal{O}_{\text{LHPC}}$, with $c \neq c_{\text{chal}}$, and $\mathcal{H}_1(m_0) = \text{Dec}(K, c)$, the challenger returns 0.

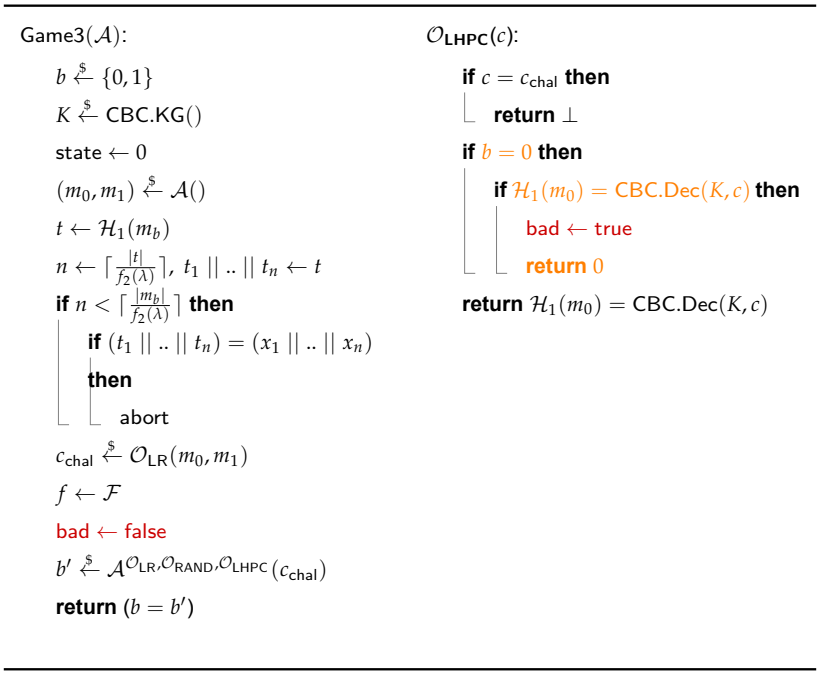


Figure D.5: Game 3.

Advantage measure

We bound the advantage of $\mathcal{A}$ against game 3 by considering the bad event bad which happens exactly when $b = 0$ and $\mathcal{A}$ delivers a ciphertext c to $\mathcal{O}_{\text{LHPC}}$ which correctly decrypts into $\mathcal{H}_1(m_0)$ and is different from the challenge ciphertext c_{chal}.

We know from the definition of $\mathcal{H}_1$ that the target ciphertext c must have length *exactly* $\lceil \frac{l_1}{f_2(\lambda)} \rceil + 1$. Let $n = \lceil \frac{l_1}{f_2(\lambda)} \rceil + 1$.

In order for $\mathcal{A}$ to produce $c = y_0 \,||\, ... \,||\, y_n$, for a message $m = \mathcal{H}_1(m_0) = x_1 \,||\, ... \,||\, x_n$, it must be able to produce, depending on m_0, one or more of its blocks without having access to f or f^{-1}.

If $m_0 \neq \mathcal{H}_1(m_0)$, and m_0 and $\mathcal{H}_1(m_0)$ share the first p blocks, for $0 \leq p < n$, then $\mathcal{A}$ must be able to generate all $n - p$ blocks of the target ciphertext c. The best possible case for $\mathcal{A}$ is when $p = n - 1$, i.e., when $\mathcal{H}_1(m_0)$ shares all of its blocks with m_0 except for the last one. Therefore, $\mathcal{A}$ only has to produce the last block y_n for c to decrypt into m_0.

However, since $y_n = f(y_{n-1} \oplus x_n)$, and $\mathcal{A}$ does not have access to f or f^{-1}, $\mathcal{A}$ has no information about y_n, and therefore has only two options: either guess y_n directly, or obtain it through the $\mathcal{O}_{\mathsf{RAND}}$ oracle.

The probability that $\mathcal{A}$ guesses y_n directly can be upper bounded by considering q_{LHPC} as the maximum number of queries to $\mathcal{O}_{\mathsf{LHPC}}$, and that for the first $\mathcal{O}_{\mathsf{LHPC}}$ query, $\mathcal{A}$ can guess y_n with probability at most $\frac{1}{2^{f_2(\lambda)}-1}$ (in case m_0 has more blocks than $\mathcal{H}_1(m_0)$, in which case the last block of c must be different from the first distinct block of c_{chal}), on the second attempt with probability $\frac{1}{2^{f_2(\lambda)}-2}$ and so on. Each query attempted by $\mathcal{A}$ effectively reduces the probability space to guess the correct block y_n. Therefore, $\mathcal{A}$ can guess y_n with probability at most

$$\sum_{i=0}^{q_{\mathsf{LHPC}}} \frac{1}{2^{f_2(\lambda)} - 1 - i} \leq \frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} \tag{D.6}$$

The probability that $\mathcal{O}_{\mathsf{RAND}}$ inputs $y_{n-1} \oplus x_n$ to f is, at most, $2^{-f_2(\lambda)}$ per block per query, since each query to $\mathcal{O}_{\mathsf{RAND}}$ can produce the same input to f more than once. Therefore, we can upper bound the probability of $f(y_{n-1} \oplus x_n)$ occuring throughout the game as

$$q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.7}$$

, where l_{max} is the largest length l given to $\mathcal{O}_{\mathsf{RAND}}$ by $\mathcal{A}$, and $\lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil$ is the maximum number of times each query to $\mathcal{O}_{\mathsf{RAND}}$ calls f.

In these circunstances, $\mathcal{A}$ can produce c such that $\mathcal{H}_1(m_0) = \mathsf{Dec}(K, c)$ with a maximum probability of

$$\frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.8}$$

If $m_0 = \mathcal{H}_1(m_0)$, then $\mathcal{A}$ must produce a target ciphertext c that is different from c_{chal}. However, in order to produce this c, $\mathcal{A}$ must use a new IV.

Recall that $\mathcal{A}$ must produce $c = y_0 \mid\mid ... \mid\mid y_n$ such that $\mathsf{Dec}(K, c) = \mathsf{Dec}(K, c_{\mathsf{chal}})$ but c cannot share all of its blocks with c_{chal}, and that in CBC mode, each ciphertext block y_i is calculated as $y_i \leftarrow f(y_{i-1} \oplus x_i), \forall i \in \{1, ..., n\}$. If c and c_{chal} share the same first block y_0 (the IV), then $y_1 = f(y_0 \oplus x_1)$ will always be the same when calculated with the same random permutation f over the same message $m = x_1 \mid\mid ... \mid\mid x_n$. This follows for all y_i, which always depends on the previous block y_{i-1}. Therefore, by sharing the first block, c necessarily shares all of its blocks with c_{chal}, and thus is either the same as or a prefix of c_{chal}.

After generating a new IV, which is block y_0, $\mathcal{A}$ must be able to create all remaining $\lceil \frac{l_1}{f_2(\lambda)} \rceil$ blocks, which are calculated by $f(y_{i-1} \oplus x_i), \forall i \in 1, .., n$, without having access to f or f^{-1}. The best possible case for $\mathcal{A}$ is when $\lceil \frac{l_1}{f_2(\lambda)} \rceil = 1$, and thus $n = 2$, which means $\mathcal{A}$ must only produce y_1, which is also the last block. From the previous case, we know that $\mathcal{A}$ can produce y_1 with probability at most

$$\frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.9}$$

Notice that each of these cases is mutually exclusive, but that the absolute best case scenario for $\mathcal{A}$ in either case can be upper bounded by equations D.8/D.9.

Since games 2 and 3 proceed identically until bad happens, through the difference lemma, we have:

$$\mid \mathsf{Pr}_2 - \mathsf{Pr}_3 \mid \leq \mathsf{Pr}_3[\mathsf{bad}] \tag{D.10}$$

Since bad can only happen if $b = 0 \wedge \mathcal{H}_1(m_0) = \mathsf{Dec}(K, c)$, we have:

$$\mathsf{Pr}_3[\mathsf{bad}] \leq 2^{-1} \, (\frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)}) \tag{D.11}$$

Therefore,

$$\mid \mathsf{Pr}_2 - \mathsf{Pr}_3 \mid \leq 2^{-1} \, \frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + 2^{-1} \, q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.12}$$

Game 4

In this game, when $b = 1$ and $\mathcal{A}$ delivers a ciphertext c to $\mathcal{O}_{\mathsf{LHPC}}$, with $c \neq c_{\mathsf{chal}}$, and $\mathcal{H}_1(m_0) = \mathsf{Dec}(K, c)$, the challenger returns 0.

Game4($\mathcal{A}$):

$b \xleftarrow{\$} \{0,1\}$
$K \xleftarrow{\$} \mathsf{CBC.KG}()$
$\mathsf{state} \leftarrow 0$
$(m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
$t \leftarrow \mathcal{H}_1(m_b)$
$n \leftarrow \lceil \frac{|t|}{f_2(\lambda)} \rceil$, $t_1 \,||\, .. \,||\, t_n \leftarrow t$
if $n < \lceil \frac{|m_b|}{f_2(\lambda)} \rceil$ **then**
  if $(t_1 \,||\, .. \,||\, t_n) = (x_1 \,||\, .. \,||\, x_n)$ **then**
    abort
$c_{\mathsf{chal}} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(m_0, m_1)$
$f \leftarrow \mathcal{F}$
$\mathsf{bad} \leftarrow \mathsf{false}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}, \mathcal{O}_{\mathsf{LHPC}}}(c_{\mathsf{chal}})$
return $(b = b')$

$\mathcal{O}_{\mathsf{LHPC}}(c)$:

if $c = c_{\mathsf{chal}}$ **then**
  return $\perp$
if $b = 0$ **then**
  if $\mathcal{H}_1(m_0) = \mathsf{CBC.Dec}(K, c)$ **then**
    return 0
if $b = 1$ **then**
  if $\mathcal{H}_1(m_0) = \mathsf{CBC.Dec}(K, c)$ **then**
    $\mathsf{bad} \leftarrow \mathsf{true}$
    return 0
return $\mathcal{H}_1(m_0) = \mathsf{CBC.Dec}(K, c)$

Figure D.6: Game 4.

Advantage measure

We bound the advantage of $\mathcal{A}$ against game 4 by considering the bad event bad which happens exactly when $b = 1$ and $\mathcal{A}$ delivers a ciphertext c to $\mathcal{O}_{\mathsf{LHPC}}$ which correctly decrypts into $\mathcal{H}_1(m_0)$.

The analysis is identical to game 3.

Let $n = \lceil \frac{l_1}{f_2(\lambda)} \rceil + 1$. Unless $\mathcal{A}$ provides a message m_1 such that $m_1 = \mathcal{H}_1(m_0)$, which results in $\mathcal{A}$ having to generate all n blocks of ciphertext c, $\mathcal{A}$ must generate, at least, one block y_n of ciphertext c. This happens with probability at most

$$\frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.13}$$

Since games 3 and 4 proceed identically until bad happens, through the difference lemma, we have:

$$| \Pr_3 - \Pr_4 | \leq \Pr_4[\mathsf{bad}] \tag{D.14}$$

Since bad can only happen if $b = 1 \wedge \mathcal{H}_1(m_0) = \mathsf{Dec}(K, c)$, we have:

$$\mathsf{Pr}_4[\mathsf{bad}] \leq 2^{-1} \left(\frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)}\right) \tag{D.15}$$

Therefore,

$$\mid \mathsf{Pr}_3 - \mathsf{Pr}_4 \mid \leq 2^{-1} \, \frac{q_{\mathsf{LHPC}}}{2^{f_2(\lambda)} - 1 - q_{\mathsf{LHPC}}} + 2^{-1} \, q_{\mathsf{RAND}} \lceil \frac{l_{\mathsf{max}}}{f_2(\lambda)} \rceil \, 2^{-f_2(\lambda)} \tag{D.16}$$

Game 5

In this game, whenever $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{LHPC}}$ with a ciphertext $c \neq c_{\mathsf{chal}}$, the oracle returns 0.

```
O_LHPC(c):
    if c = c_chal then
        return ⊥
    return 0
```

Figure D.7: Game 5.

Advantage measure

In the previous game, the oracle $\mathcal{O}_{\mathsf{LHPC}}$ always returns 0 to $\mathcal{A}$ when $c \neq c_{\mathsf{chal}}$, regardless of b. Therefore, this game is functionally identical to game 4, and we have:

$$\mathsf{Pr}_4 = \mathsf{Pr}_5 \tag{D.17}$$

Game 6

In this game, whenever the challenger executes $\mathsf{Enc}(K, m)$ or $\mathsf{Dec}(K, c)$, the computation of $f(\cdot)$ and $f^{-1}(\cdot)$ is replaced by $\mathsf{F}(K, \cdot)$ and $\mathsf{F}^{-1}(K, \cdot)$, respectively.

Advantage measure

The only modification in game 6 corresponds to the replacement of the underlying invertible truly random permutation in the encryption scheme with an invertible, pseudo-random permutation. Therefore, we can bound the advantage of $\mathcal{A}$ by considering an adversary $\mathcal{B}_2$ that can distinguish between $\mathsf{F}(K, \cdot)$ and f by invoking $\mathcal{A}$. Therefore:

Enc(K, m):

1: $x_1 \,||\, ... \,||\, x_n \leftarrow m$ s.t. $|\, x_i \,| = f_2(\lambda)\ \forall i \in \{1, ..., n\}$
2: $y_0 \leftarrow \mathsf{SetIV}()$
3: **for all** $i \in \{1, ..., n\}$ **do**
4: $\quad y_i \leftarrow \mathsf{F}(K, y_{i-1} \oplus x_i)$
5: $y \leftarrow y_0 \,||\, ... \,||\, y_n$
6: **return** y

Dec(K, c):

1: $y_0 \,||\, ... \,||\, y_n \leftarrow c$ s.t. $|\, y_i \,| = f_2(\lambda)\ \forall i \in \{0, ..., n\}$
2: **for all** $i \in \{1, ..., n\}$ **do**
3: $\quad x_i \leftarrow y_{i-1} \oplus \mathsf{F}^{-1}(K, y_i)$
4: $m \leftarrow x_1 \,||\, ... \,||\, x_n$
5: **return** m

Figure D.8: Modifications to CBC - Game 6.

$$\mathsf{Pr}_5 - \mathsf{Pr}_6 \leq \mathsf{Adv}^{\mathsf{prp}}_{\mathsf{F}}(\mathcal{B}_2) \tag{D.18}$$

Final Analysis

We can now calculate an upper bound for the advantage of $\mathcal{A}$ against game 6 by reduction to the IND-1$PA security of CBC.

Let $\mathcal{B}_3$ be an adversary against the IND-1$PA security of CBC. We show that if $\mathcal{A}$ wins game 6, then $\mathcal{B}_3$ wins the IND-1$PA game.

$\mathcal{B}_3(\mathcal{A}, \mathcal{O}^{\text{IND-1\$PA}}_{\text{LR}}, \mathcal{O}^{\text{IND-1\$PA}}_{\text{RAND}})$:
$\quad (m_0, m_1) \xleftarrow{\$} \mathcal{A}()$
$\quad t \leftarrow \mathcal{H}_1(m_b)$
$\quad n \leftarrow \lceil \frac{|t|}{f_2(\lambda)} \rceil,\ t_1 \,||\, .. \,||\, t_n \leftarrow t$
$\quad$ **if** $n < \lceil \frac{|m_b|}{f_2(\lambda)} \rceil$ **then**
$\quad\quad$ **if** $(t_1 \,||\, .. \,||\, t_n) = (x_1 \,||\, .. \,||\, x_n)$ **then**
$\quad\quad\quad$ abort
$\quad c_{\text{chal}} \xleftarrow{\$} \mathcal{O}_{\text{LR}}(m_0, m_1)$
$\quad b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\text{LR}}, \mathcal{O}_{\text{RAND}}, \mathcal{O}_{\text{LHPC}}}(c_{\text{chal}})$
$\quad$ **return** (b')

$\mathcal{O}_{\textbf{LR}}(m_0, m_1)$:
$\quad c_{\text{chal}} \leftarrow \mathcal{O}^{\text{IND-1\$PA}}_{\text{LR}}(m_0, m_1)$
$\quad$ **return** c_{chal}

$\mathcal{O}_{\textbf{LHPC}}(c)$:
$\quad$ **if** $c = c_{\text{chal}}$ **then**
$\quad\quad$ **return** $\perp$
$\quad$ **return** 0

$\mathcal{O}_{\textbf{RAND}}(l)$:
$\quad (m'_0, m'_1, c') \leftarrow \mathcal{O}^{\text{IND-1\$PA}}_{\text{RAND}}(l)$
$\quad$ **return** (m'_0, m'_1, c')

Figure D.9: Game 6 simulated by $\mathcal{B}_3$, with corresponding Left-Right oracle $\mathcal{O}_{\text{LR}}$, RAND oracle $\mathcal{O}_{\text{RAND}}$ and LHPC oracle $\mathcal{O}_{\text{LHPC}}$.

Advantage measure

From the definition of $\mathcal{B}_3$, if $\mathcal{A}$ wins game 6, then $\mathcal{B}_3$ wins against the IND-1$PA security of CBC.

$$\mathsf{Pr}_6 \leq \mathsf{Pr}^{\text{IND-1\$PA}}_{\text{CBC}}(\mathcal{B}_3) \tag{D.19}$$

Note that the running times of all adversaries are close to that of $\mathcal{A}$.

This concludes the proof.

E. CTAP 2.1+ SUF-t Authentication Proof

In this Appendix, we present the complete pseudocode proof for the SUF-t authentication security of CTAP 2.1+, as an instance of mPACA, with a sequence of games. We base our modifications for each game on the mPACA model oracles and CTAP 2.1+ syntax as presented in Chapter 5.

Let $\Pr_g$ denote the probability that game g outputs 1. Each game's modifications that are *persistent* throughout the proof are showcased in orange. Modifications that exist only when employing a hybrid argument or a reduction, and therefore are only relevant to a specific game, are shown in green. Finally, bad events and the end of a reduction are marked in red and are also relevant only to their respective game.

Game 0

This game is the original $\mathrm{Expt}^{\text{SUF-t}}_{\text{CTAP 2.1+}}(\mathcal{A})$ security experiment. Therefore,

$$\Pr_0 = \mathrm{Adv}^{\text{SUF-t}}_{\text{CTAP 2.1+}}(\mathcal{A}) \tag{E.1}$$

Game 1

In this game we replace every key K_1 and K_2 derived from Z by all passive ECDH key exchange sessions with two independent and uniformly random session keys $\tilde{K}_1$ and $\tilde{K}_2$, and bound the advantage of $\mathcal{A}$ by reducing to the sCDH security of ECDH.

We maintain two structures $\mathcal{L}_{\text{sCDH}}$ and $\mathcal{L}_{\mathcal{H}_2}$ respectively to save ocurrences of client and token public keys in order to maintain consistency throughout the game and to save adversary queries to random oracle $\mathcal{H}_2$. Every execution of encapsulate, which happens only on clients, will first check if the current pair of public keys has appeared before. If it hasn't, the experiment samples the two session keys $\tilde{K}_1$ and $\tilde{K}_2$ and saves them in $\mathcal{L}_{\text{sCDH}}$. Otherwise, the challenger uses the $\tilde{K}_1$ and $\tilde{K}_2$ which were previously stored in $\mathcal{L}_{\text{sCDH}}$. On the token side, every time decapsulate executes, it performs the same check. If the pair does not exist, then the public key received must not have been generated by an honest client, since $\mathcal{L}_{\text{sCDH}}$ is only ever set by the client in encapsulate. In that situation, the regular ECDH key derivation is used. If it does exist, then both keys have already appeared in the experiment before, in a previous execution of encapsulate, which means the same $\tilde{K}_1$ and $\tilde{K}_2$

must be used, in order to maintain consistency. It is clear that, unless $\mathcal{A}$ ever queried $\mathcal{H}_2$ with the shared secret Z derived in a Setup or Execute query, Game 1 proceeds identically to Game 0.

```
Game1(A):
    L_authC, L_authT ← ∅, L_valid ← ∅, L_corrupt ← ∅
    L_sCDH, L_H2 ← ∅
    win-SUF-t ← 0
    () ←$ A^O(1^λ)
    return win-SUF-t

AdvH2 (u1, u2):
    v ← H2(u1, u2)
    L_H2[u1, u2] ← v
    return v
```

Figure E.1: Game 1 on the left, and $\mathcal{H}_2$ wrapper Adv$\mathcal{H}_2$ on the right - Game 1. Adv$\mathcal{H}_2$ is a wrapper that $\mathcal{A}$ can use to call the random oracle, so that we can distinguish calls to $\mathcal{H}_2$ made specifically by $\mathcal{A}$.

```
encapsulate (pk'):
    if L_sCDH[pk', pk] = ⊥ then
        K̃1 ←$ {0,1}^2λ, K̃2 ←$ {0,1}^2λ
        K̃ ← (K̃1, K̃2)
        L_sCDH[pk', pk] ← K̃
    else
        K̃ ← L_sCDH[pk', pk]
    K ← K̃
    c ← pk
    return (c, K)

decapsulate (c):
    K̃ ← L_sCDH[pk, c]
    if K̃ ≠ ⊥ then
        K ← K̃
        return K
    Z ← XCoordinateOf(sk · c)
    K1 ← H2(Z, "CTAP2 HMAC KEY")
    K2 ← H2(Z, "CTAP2 AES KEY")
    K ← (K1, K2)
    return K
```

Figure E.2: Modifications to PIN/UV Auth Protocol functions encapsulate and decapsulate - Game 1. Variables sk and pk refer to the stored keys being used by the client session or the token session.

Hybrid Argument

We bound the advantage of $\mathcal{A}$ against Game 1 by creating n_{TokenPK} hybrid games between Game 0 and Game 1, where n_{TokenPK} is the total number of token public keys sampled throughout the experiment that are output to $\mathcal{A}$ in Setup, Execute and Send-Bind-T queries, via the obtainSharedSecret-T function. We then define a bad event bad on game $0.i$, where the game aborts and $\mathcal{A}$ loses if it queries $\mathcal{H}_2$ with $u_1 = Z = \text{XCoordinateOf}(sk \cdot pk^i_{\text{token}})$, and pk^i_{token} and c exist in $\mathcal{L}_{\text{sCDH}}$ at any point in the game, with pk^i_{token} as the i^{th} token public key and sk and c any client private and public key generated by a client session that received pk^i_{token}.

We use the notation from regular Diffie-Hellman instead of Elliptic Curve Diffie-Hellman to simplify the code and make it more legible. For example, instead of $Z \leftarrow \text{XCoordinateOf}(sk \cdot pk')$, we use $Z \leftarrow (pk')^{sk}$ during this hybrid and the reduction to sCDH. However, every operation used in this proof uses ECDH.

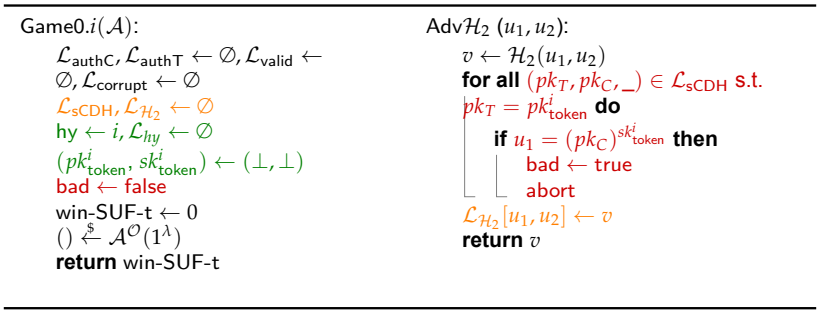


Figure E.3: Hybrid game $0.i$ on the left and random oracle wrapper Adv$\mathcal{H}_2$ on the right.

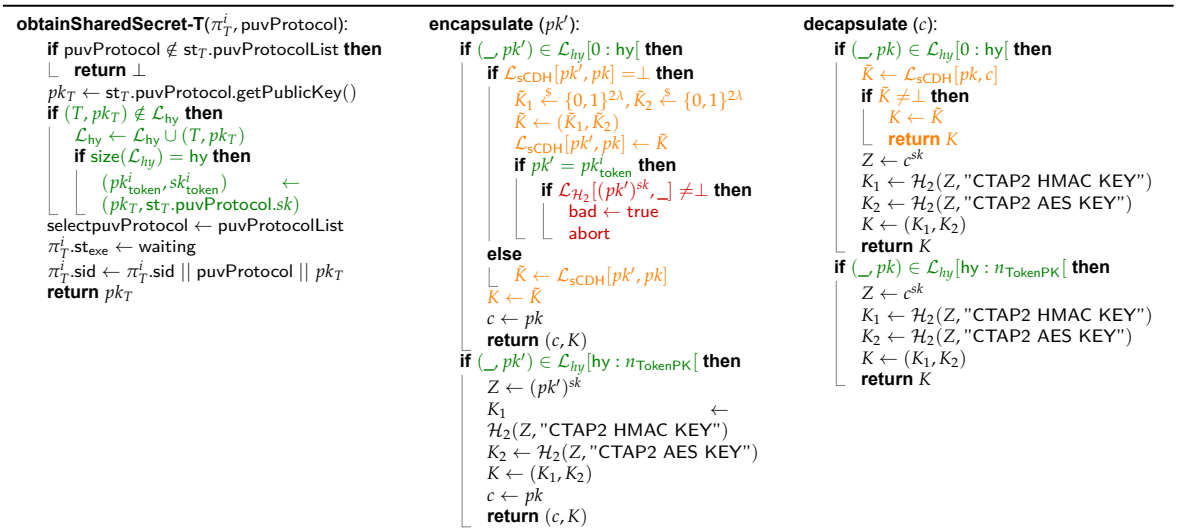


Figure E.4: obtainSharedSecret-T on the left and PIN/UV Auth Protocol encapsulate and decapsulate functions on the right, from hybrid game $0.i$.

Reduction to the sCDH security of ECDH

We now construct an adversary $\mathcal{B}^i_1$ that simulates game $0.i$ to $\mathcal{A}$, in such a way that when bad happens, $\mathcal{B}^i_1$ wins against the sCDH security of ECDH.

Throughout the simulation of game $0.i$, $\mathcal{B}^i_1$ sets the i^{th} token public key as the sCDH challenge public key g^a and any client public key generated by a client session that receives g^a to be g^{b+r_w}, for $r_w \leftarrow \mathbb{Z}_q$ and $w \geq 1$. In such client sessions, $\mathcal{B}^i_1$ checks if $u_1 = Z$ is in $\mathcal{L}_{\text{sCDH}}$ using its $\mathcal{O}_a$ oracle. Additionally, every time $\mathcal{A}$ calls random oracle $\mathcal{H}_2$ (always through Adv$\mathcal{H}_2$) $\mathcal{B}^i_1$ also checks if for any public key pair in $\mathcal{L}_{\text{sCDH}}$ that includes g^a, $u_1 = Z$.

It is not hard to see that, whenever bad happens during $\mathcal{B}^i_1$'s simulation, $\mathcal{B}^i_1$ wins by using $\mathcal{A}$'s input to the random oracle $\mathcal{H}_2$. Indeed, if $\mathcal{A}$ ever calls $\mathcal{H}_2$ with input $u_1 = Z = (g^{b+r_w})^a, \forall r_w \in \mathcal{L}_{r_w}$, then $\mathcal{B}^i_1$ can output $Z' \leftarrow u_1 \cdot (g^a)^{-r_w} \equiv (g^{b+r_w})^a \cdot (g^a)^{-r_w} \equiv g^{ab}$ and win the sCDH game.

Challenger$^{\text{sCDH}}_{\text{ECDH}}$:
$(a, g^a) \xleftarrow{\$}$ ECDH.KG()
$(b, g^b) \xleftarrow{\$}$ ECDH.KG()
$Z' \leftarrow \mathcal{A}^{\mathcal{O}_a}(g, g^a, g^b)$
$Z \leftarrow (g^b)^a$
win-sCDH $\leftarrow (Z = Z')$
return win-sCDH

$\mathcal{O}_a(c, Z)$:
$Z' \leftarrow c^a$
return $(Z = Z')$

$\mathcal{B}_1^i(g, g^a, g^b, \mathcal{A}, \mathcal{O}_a)$:
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
hy $\leftarrow i, \mathcal{L}_{hy} \leftarrow \emptyset$
$(pk^i_{\text{token}}, sk^i_{\text{token}}) \leftarrow (\perp, \perp)$
$\mathcal{L}_{r_w}, \mathcal{L}_{\text{Fake}\mathcal{H}_2} \leftarrow \emptyset$
$Z' \xleftarrow{\$} \mathbb{Z}_q$
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
▷ *If bad ever happened, Z' has the correct value because it was set during the game simulation from $\mathcal{B}_1^i$.*
return Z'

Figure E.5: sCDH Challenger and oracle on the left and adversary $\mathcal{B}_1^i$'s simulation of game $0.i$ on the right.

$\mathcal{H}_2\ (u_1, u_2)$:
if $t_1[u_1, u_2] \neq \perp$ **then**
  return $t_1[u_1, u_2]$
else
  for $(c, (\tilde{K}_1, \tilde{K}_2)) \in \mathcal{L}_{\text{Fake}\mathcal{H}_2}$ **do**
    if $\mathcal{O}_a(c, u_1) =$ true **then**
      if $u_2 =$ "CTAP2 HMAC KEY" **then**
        $t_1[u_1, u_2] \leftarrow \tilde{K}_1$
        return $\tilde{K}_1$
      else if $u_2 =$ "CTAP2 AES KEY" **then**
        $t_1[u_1, u_2] \leftarrow \tilde{K}_2$
        return $\tilde{K}_2$
  $v \xleftarrow{\$} \{0,1\}^{2\lambda}$
  $t_1[u_1, u_2] \leftarrow v$
  return v

Adv$\mathcal{H}_2\ (u_1, u_2)$:
$v \leftarrow \mathcal{H}_2(u_1, u_2)$
for all $(r_w, g^b \cdot g^{r_w}) \in \mathcal{L}_{r_w}$ **do**
  if $\mathcal{L}_{\text{sCDH}}[g^a, g^b \cdot g^{r_w}] \neq \perp$ **then**
    if $\mathcal{O}_a(g^b \cdot g^{r_w}, u_1) =$ true **then**
      $Z' = u_1 \cdot (g^a)^{-r_w}$
      win-sCDH
$\mathcal{L}_{\mathcal{H}_2}[u_1, u_2] \leftarrow v$
return v

Figure E.6: Random Oracle $\mathcal{H}_2$ and wrapper Adv$\mathcal{H}_2$ simulated by $\mathcal{B}_1^i$.

Advantage measure

By the definition of $\mathcal{B}_1^i$, if bad happens, then $\mathcal{A}$ found the solution to the problem given by the sCDH challenger to $\mathcal{B}_1^i$, i.e., g^{ab}. Therefore:

$$\Pr_{0.i}[\text{bad}] \leq \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1^i) \tag{E.2}$$

Since games $0.i$ and $0.(i-1)$ are identical until bad happens, through the difference lemma, we have:

$$\Pr_{0.i}[\text{bad}] \geq |\ \Pr_{0.(i-1)} - \Pr_{0.i}\ |, \forall i \in [1, n_{\text{TokenPK}}] \tag{E.3}$$

$$\text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1^i) \geq |\ \Pr_{0.(i-1)} - \Pr_{0.i}\ |, \forall i \in [1, n_{\text{TokenPK}}] \tag{E.4}$$

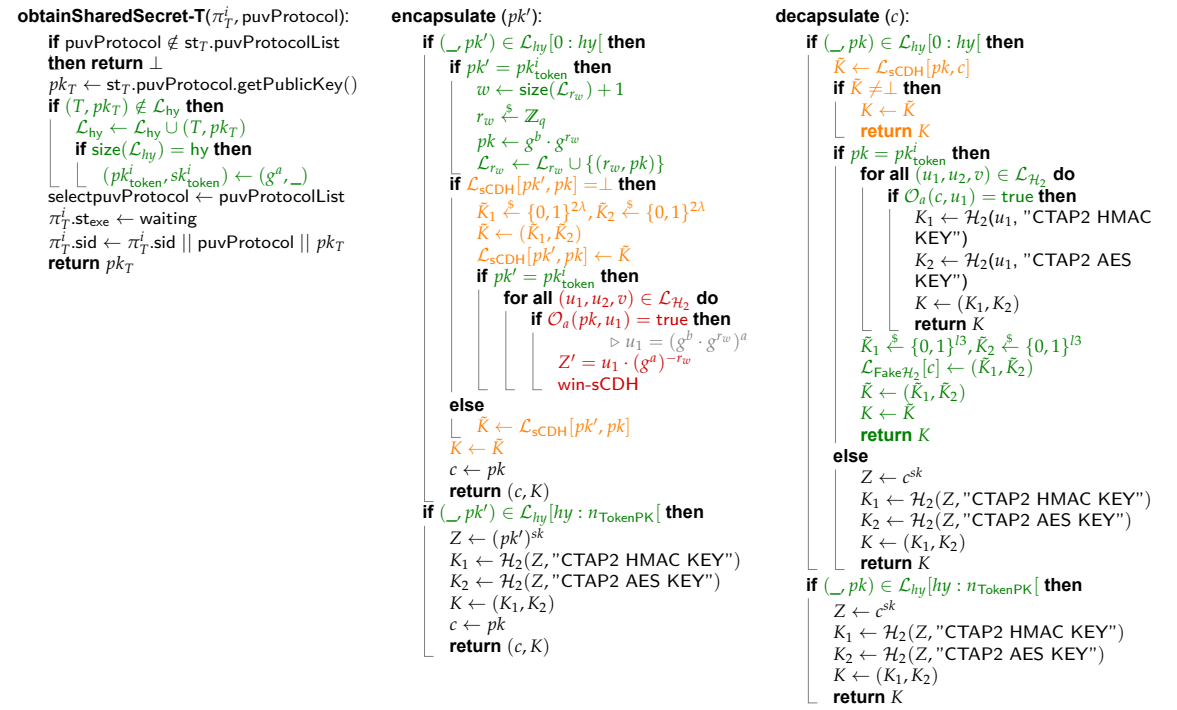


Figure E.7: obtainSharedSecret-T on the left and PIN/UV Auth Protocol encapsulate and decapsulate functions on the right, simulated by $\mathcal{B}_1^i$.

$$\sum_{i=1}^{n_{\text{TokenPK}}} \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1^i) \geq \sum_{i=1}^{n_{\text{TokenPK}}} \left| \text{Pr}_{0.(i-1)} - \text{Pr}_{0.i} \right| \tag{E.5}$$

$$\sum_{i=1}^{n_{\text{TokenPK}}} \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1^i) \geq \left| \text{Pr}_{0.0} - \text{Pr}_{0.n_{\text{TokenPK}}} \right| \tag{E.6}$$

Let $\mathcal{B}_1$ be the adversary with the highest advantage against the sCDH game out of all adversaries $\mathcal{B}_1^i$, $\forall i \in [1, n_{\text{TokenPK}}]$. Then, it is always true that:

$$n_{\text{TokenPK}} \ \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1) \geq \left| \text{Pr}_{0.0} - \text{Pr}_{0.n_{\text{TokenPK}}} \right| \tag{E.7}$$

Since Game 0.0 corresponds to Game 0 and Game $0.n_{\text{TokenPK}}$ corresponds to Game 1, we have:

$$n_{\text{TokenPK}} \ \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1) \geq \left| \text{Pr}_0 - \text{Pr}_1 \right| \tag{E.8}$$

Given that token ECDH public keys can only be given to $\mathcal{A}$ in Setup, Execute or Send-Bind-T queries, we know $n_{\text{TokenPK}} \leq (q_S + q_E + q_{\text{Send}})$. Therefore:

$$(q_S + q_E + q_{Send})\,\mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1) \geq |\,\mathsf{Pr}_0 - \mathsf{Pr}_1\,| \tag{E.9}$$

Game 2

In this game, the pinHash is no longer saved in the token's state, being saved instead on a new structure $\mathcal{L}_{\mathsf{TokenSetup}}$ which is not available to $\mathcal{A}$.

Game2($\mathcal{A}$):
- $\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{valid}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **return** win-SUF-t

Figure E.8: Game 2.

setPIN-T(π^i_T, puvProtocol, c, c_p, t_p):
- **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ $\mathcal{L}_{\mathsf{TokenSetup}}[T] \neq \perp$ **then return** $\perp$
- $K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
- **if** $K = \perp \vee$ st$_T$.puvProtocol.verify(K, c_p, t_p) = false **then return** $\perp$
- pin $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_p)
- **if** pin $\notin \mathcal{P}$ **then return** $\perp$
- $\mathcal{L}_{\mathsf{TokenSetup}}[T] \leftarrow$ H(pin)
- st$_T$.pinRetries $\leftarrow$ pinRetriesMax
- **return** accepted

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
- **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then return** ($\perp$, false)
- $K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
- **if** $K = \perp$ **then return** ($\perp$, false)
- st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
- pinHash $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_{ph})
- pinHash$_T \leftarrow \mathcal{L}_{\mathsf{TokenSetup}}[T]$
- **if** pinHash $\neq$ pinHash$_T$ **then**
  - st$_T$.puvProtocol.regenerate()
  - st$_T.m \leftarrow$ st$_T.m - 1$
  - **if** st$_T.m = 0$ **then**
    - authPowerUp-T(st$_T$)
    - **return** ($\perp$, true)
  - **else return** ($\perp$, false)
- st$_T.m \leftarrow 3$, st$_T$.pinRetries $\leftarrow$ pinRetriesMax
- **for all** puvProtocol' $\in$ st$_T$.puvProtocolList **do**
  - st$_T$.puvProtocol'.resetPuvToken()
- π^i_T.bs $\leftarrow \pi^i_T$.puvProtocol.pt
- $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt(K, π^i_T.bs)
- π^i_T.st$_{\mathsf{exe}} \leftarrow$ bindDone
- π^i_T.canValidate $\leftarrow$ true
- π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || c || c_{ph} || c_{pt} || false
- **return** (c_{pt}, false)

Figure E.9: Modified CTAP 2.1+ setPIN-T and obtainPinUvAuthToken-T functions - Game 2.

Advantage measure

Since this model does not allow tokens to be corrupted to reveal their state st$_T$, Game 2 is functionally identical to Game 1, and we have:

$$\mathsf{Pr}_1 = \mathsf{Pr}_2 \tag{E.10}$$

Game 3

In this game, $\mathcal{A}$ loses by causing a bad event bad if and only if the challenger samples two identical ECDH public keys, in either a token session or a client session. Both client and token sessions will sample a fresh ECDH key pair through the regenerate function.

Game3($\mathcal{A}$):

$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF-t

Figure E.10: Game 3. The flag bad in red is only used in this game, to signal when a collision happens.

regenerate ():

$(pk, sk) \xleftarrow{\$}$ ECDH.KG()
if $pk \in \mathcal{L}_{\text{pk}}$ **then**
    bad $\leftarrow$ true
    abort
$\mathcal{L}_{\text{pk}} \leftarrow \mathcal{L}_{\text{pk}} \cup \{pk\}$

Figure E.11: Modifications to PIN/UV Auth Protocol function regenerate - Game 3.

Advantage measure

Each client-side ECDH public key is generated either on Setup or Execute queries, in the obtainSharedSecret-C-start function. Since each query always generates an ECDH public key, we have at most $q_{\text{S}} + q_{\text{E}}$ client ECDH public keys during the experiment.

Each token-side ECDH public key can be generated either during the creation of the token with the NewT oracle, by rebooting the token via the Reboot oracle, or during the Send-Bind-T oracle, in the obtainPinUvAuthToken-T function, whenever the pinHash verification fails. Since each NewT or Reboot query generates a new ECDH key pair every time, and each Send-Bind-T generates at most 2 ECDH key pairs (recall that obtainPinUvAuthToken-T regenerates whenever the pinHash verification fails but also when $\text{st}_T.m$ reaches 0, which can happen during the same execution), we have at most $q_{\text{NT}} + q_{\text{R}} + 2q_{\text{Send}}$ token ECDH public keys during the experiment.

Let $n_{pk} = q_{\text{S}} + q_{\text{E}} + q_{\text{NT}} + q_{\text{R}} + 2q_{\text{Send}}$ be the total number of ECDH public keys generated through these oracles.

Considering the maximum probability of a collision between two keys as $1/q$, where q is the prime order of the ECDH group used by the challenger throughout the experiment, and $n_{pk}^2/2$ as an upper bound for the amount of ECDH public keys pairs that can exist, we have:

$$\mathsf{Pr}_3[\mathsf{bad}] \leq n_{pk}^2/(2q) \tag{E.11}$$

Since games 2 and 3 are identical until bad, by the difference lemma, we have:

$$\mathsf{Pr}_3[\mathsf{bad}] \geq |\ \mathsf{Pr}_2 - \mathsf{Pr}_3\ | \tag{E.12}$$

And, therefore:

$$n_{pk}^2/(2q) \geq |\ \mathsf{Pr}_2 - \mathsf{Pr}_3\ | \tag{E.13}$$

Finally, from the definition of n_{pk}, we have:

$$\frac{(q_\mathsf{S} + q_\mathsf{E} + q_\mathsf{NT} + q_\mathsf{R} + 2q_\mathsf{Send})^2}{2q} \geq |\ \mathsf{Pr}_2 - \mathsf{Pr}_3\ | \tag{E.14}$$

Game 4

In this game, the game aborts and $\mathcal{A}$ loses if when a collision occurs on some output of H when called by the challenger.

In order to easily represent the changes in code, and ensure every time a call to H is made the game checks for collisions, we create a wrapper function H_w which calls H and performs the collision check, and change every location in the entire experiment where H is called with H_w.

Game4($\mathcal{A}$):
- $\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{valid}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{hash}} \leftarrow \emptyset$
- bad $\leftarrow$ false
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **return** win-SUF-t

Figure E.12: Game 4. The flag bad in red is only used in this game, to signal when a pinHash collision happens.

H_w (pin):
- pinHash $\leftarrow \mathsf{H}(\mathsf{pin})$
- **for all** $(\mathsf{pin}', \mathsf{pinHash}') \in \mathcal{L}_{\mathsf{hash}}$ **do**
  - **if** $\mathsf{pinHash}' = \mathsf{pinHash} \wedge \mathsf{pin}' \neq \mathsf{pin}$ **then**
    - bad $\leftarrow$ true
    - abort
- $\mathcal{L}_{\mathsf{hash}}[\mathsf{pin}] \leftarrow \mathsf{pinHash}$
- **return** pinHash

Figure E.13: New wrapper function H_w which calls H and aborts if a collision is found - Game 4.

obtainPinUvAuthToken-C-start(π_C^j, pin):
pinHash $\leftarrow$ H_w(pin)
$c_{ph} \xleftarrow{\$} \pi_C^j$.selectedPuvProtocol.encrypt(π_C^j.K, pinHash)
π_C^j.$st_{exe} \leftarrow$ bindStart
π_C^j.sid $\leftarrow \pi_C^j$.sid || c_{ph}
return c_{ph}

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p):
if puvProtocol $\notin$ st_T.puvProtocolList $\vee$
$\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then return** $\perp$
$K \leftarrow st_T$.puvProtocol.decapsulate(c)
if $K = \perp \vee st_T$.puvProtocol.verify(K, c_p, t_p) = false **then return** $\perp$
pin $\leftarrow st_T$.puvProtocol.decrypt(K, c_p)
if pin $\notin \mathcal{P}$ **then return** $\perp$
$\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow H_w$(pin)
st_T.pinRetries $\leftarrow$ pinRetriesMax
return accepted

Figure E.14: Modified CTAP 2.1+ functions obtainPinUvAuthToken-C-start and setPIN-T - Game 4.

Advantage measure

Because Games 3 and 4 are identical until bad, through the difference lemma, we have:

$$| \Pr_3 - \Pr_4 | \leq \Pr_4[\text{bad}] \tag{E.15}$$

Since H is assumed to be collision resistant, there exists an adversary $\mathcal{B}_4$ such that:

$$\Pr_4[\text{bad}] \leq \text{Adv}^{\text{coll}}_{\text{H}}(\mathcal{B}_4) \tag{E.16}$$

Therefore:

$$| \Pr_3 - \Pr_4 | \leq \text{Adv}^{\text{coll}}_{\text{H}}(\mathcal{B}_4) \tag{E.17}$$

Game 5

In this game, we replace all pins encrypted by any client session and sent to any token session during Setup with a constant pin 0000. Additionally, on any token session that shares the same session key $\tilde{K}$ with a different token session of the same token involved in a previous Setup query, we replace the encryption of the pinUvAuthToken pt with the encryption of a uniform random value $\tilde{pt}$.

We bound the advantage of $\mathcal{A}$ by reducing to the IND-1$PA-LHPC security of SKE. The reason it does not suffice to reduce to the IND-1CPA security results from the observation that it is not guaranteed that the session key $\tilde{K}$ that encrypts the constant pin will not be used again to create more ciphertexts in the experiment. If $\mathcal{A}$ saves a client public key c used to perform a Setup with a token session π_T^i and then sends the same key c to another session $\pi_T^{i'}$ of the same token during a Bind execution via the Send-Bind-T oracle,

Game5($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
$\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF-t

Setup (T, i, C, j, U):
$\text{pin}_U \leftarrow \mathcal{L}_{\text{valid}}[U]$
if $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  return $\perp$
$\pi_T^i \leftarrow \text{st}_T$
$\mathcal{L}_{\text{pin}}[\pi_T^i] \leftarrow \text{pin}_U$
trans $\xleftarrow{\$}$ Setup$(\pi_T^i, \pi_C^j, \text{pin}_U)$
π_T^i.isValid, π_C^j.isValid $\leftarrow$ false
st_T.user $\leftarrow U$
return trans

Figure E.15: Game 5 and modified Setup oracle.

setPIN-C(π_C^j, pin_U):
if $\text{pin}_U \notin \mathcal{P}$ **then**
  return $\perp$
$c_p \xleftarrow{\$} \pi_C^j$.selectedPuvProtocol.encrypt(π_C^j.K, 0000)
$t_p \xleftarrow{\$} \pi_C^j$.selectedPuvProtocol.authenticate(π_C^j.K, c_p)
return (c_p, t_p)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p):
if puvProtocol $\notin \text{st}_T$.puvProtocolList $\vee$
$\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then return** $\perp$
$K \leftarrow \text{st}_T$.puvProtocol.decapsulate(c)
if $K = \perp \vee \text{st}_T$.puvProtocol.verify$(K, c_p, t_p) =$ false
then return $\perp$
$\text{pin} \leftarrow \mathcal{L}_{\text{pin}}[\pi_T^i]$
if pin $\notin \mathcal{P}$ **then return** $\perp$
$\mathcal{L}_{\text{setupKeys}}[T] \leftarrow (\text{st}_T.\text{puvProtocol}.pk, c)$
$\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow \text{H}_{\text{w}}(\text{pin})$
st_T.pinRetries $\leftarrow$ pinRetriesMax
return accepted

Figure E.16: Modified CTAP 2.1+ functions setPIN-C and setPIN-T - Game 5.

obtainPinUvAuthToken-T(π_T^i, puvProtocol, c, c_{ph}):
if puvProtocol $\notin \text{st}_T$.puvProtocolList $\vee \text{st}_T$.pinRetries $= 0$ **then**
  return $(\perp,$ false$)$
$K \leftarrow \text{st}_T$.puvProtocol.decapsulate(c)
if $K = \perp$ **then**
  return $(\perp,$ false$)$
st_T.pinRetries $\leftarrow$ pinRetries $- 1$
pinHash $\leftarrow \text{st}_T$.puvProtocol.decrypt(K, c_{ph})
$\text{pinHash}_T \leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
if pinHash $\neq \text{pinHash}_T$ **then**
  st_T.puvProtocol.regenerate()
  $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
  if $\text{st}_T.m = 0$ **then**
    authPowerUp-T(st_T)
    return $(\perp,$ true$)$
  else
    return $(\perp,$ false$)$
$\text{st}_T.m \leftarrow 3, \text{st}_T$.pinRetries $\leftarrow$ pinRetriesMax
for all puvProtocol' $\in \text{st}_T$.puvProtocolList **do**
  st_T.puvProtocol'.resetPuvToken()
π_T^i.bs $\leftarrow \pi_T^i$.puvProtocol.pt
▷ *If this happens, then the ECDH key pair used in this execution was already used in a previous Setup query for token T.*
if $(\text{st}_T.\text{puvProtocol}.pk, c) = \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi_T^i.\text{bs})}$
  $c_{pt} \xleftarrow{\$} \text{st}_T$.puvProtocol.encrypt$(K, \tilde{pt})$
else
  $c_{pt} \xleftarrow{\$} \text{st}_T$.puvProtocol.encrypt$(K, \pi_T^i.\text{bs})$
$\pi_T^i.\text{st}_{\text{exe}} \leftarrow$ bindDone
π_T^i.canValidate $\leftarrow$ true
π_T^i.sid $\leftarrow \pi_T^i$.sid $||$ puvProtocol $|| c || c_{ph} || c_{pt} ||$ false
return $(c_{pt},$ false$)$

Figure E.17: Modified CTAP 2.1+ function obtainPinUvAuthToken-T - Game 5.

$\pi_T^{i'}$ might use the same session key $\tilde{K}_2$ to encrypt a pinUvAuthToken, provided $\mathcal{A}$ can pass the pinHash verification step. This is possible because it is *not* guaranteed that a token T will always generate a fresh ECDH key pair for every new key agreement.

In order to know, during the execution of obtainPinUvAuthToken-T, if the token session π_T^i is reusing the same session key used in a previous Setup query, we create a structure $\mathcal{L}_{\text{setupKeys}}$ which, during Setup, will fix the ECDH client and token public keys on token T. Later, if the ECDH public keys used in obtainPinUvAuthToken-T are the same for the given token T, we know it will be reusing the same session key, and thus we must replace the pinUvAuthToken being encrypted.

Hybrid Argument

To bound $\mathcal{A}$'s advantage against Game 5, we begin by creating n_{setup} hybrid games between Game 4 and Game 5, where n_{setup} is the maximum number of (valid) calls to the Setup oracle (valid refers to calls that actually result in executing the Setup between a client session and token session). Then, we replace the pin with the constant value 0000, the pinUvAuthToken from the relevant Send-Bind-T queries with a random value $\tilde{pt}$, and fix the ECDH public keys in $\mathcal{L}_{\text{setupKeys}}$ only until the i^{th} execution of Setup.

Game4.$i(\mathcal{A})$:
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- hy $\leftarrow i$, $\text{count}_{\text{setup}} \leftarrow 0$
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
- **return** win-SUF-t

Setup (T, i, C, j, U):
- $\text{pin}_U \leftarrow \mathcal{L}_{\text{valid}}[U]$
- **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  - **return** $\perp$
- $\pi_T^i \leftarrow \text{st}_T$
- $\text{count}_{\text{setup}} \leftarrow \text{count}_{\text{setup}} + 1$
- $\mathcal{L}_{\text{pin}}[\pi_T^i] \leftarrow \text{pin}_U$
- trans $\xleftarrow{\$}$ Setup$(\pi_T^i, \pi_C^j, \text{pin}_U)$
- π_T^i.isValid, π_C^j.isValid $\leftarrow$ false
- st_T.user $\leftarrow U$
- **return** trans

Figure E.18: Hybrid game 4.i.

Reduction to the IND-1$PA-LHPC security of SKE

We now construct an adversary $\mathcal{B}_5^i$ against the IND-1$PA-LHPC security of SKE that simulates game 4.i to $\mathcal{A}$ perfectly, except when $\mathcal{A}$ performs the i^{th} valid query to Setup. In this case, when executing setPIN-C, $\mathcal{B}_5^i$ sends pin and 0000 to $\mathcal{O}_{\text{LR}}$ and receives a challenge ciphertext c_p, instead of encrypting it normally with the derived session key $\tilde{K}_2$. Then, when running setPIN-T on the corresponding token session, $\mathcal{B}_5^i$ flags the public key pair that was used during this i^{th} Setup, to maintain consistency every time the same public

setPIN-C(π_C^j, pin_U):
- **if** $\text{pin}_U \notin \mathcal{P}$ **then**
  - **return** $\perp$
- **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
  - $c_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.encrypt}(\pi_C^j.\text{K}, 0000)$
  - $t_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.authenticate}(\pi_C^j.\text{K}, c_p)$
  - **return** (c_p, t_p)
- **else**
  - $c_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.encrypt}(\pi_C^j.\text{K}, \text{pin}_U)$
  - $t_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.authenticate}(\pi_C^j.\text{K}, c_p)$
  - **return** (c_p, t_p)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p):
- **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee$
  $\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then return** $\perp$
- $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
- **if** $K = \perp \vee \text{st}_T.\text{puvProtocol.verify}(K, c_p, t_p) = \text{false}$
  then return $\perp$
- **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
  - $\text{pin} \leftarrow \mathcal{L}_{\text{pin}}[\pi_T^i]$
- **else**
  - $\text{pin} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_p)$
- **if** $\text{pin} \notin \mathcal{P}$ **then return** $\perp$
- **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
  - $\mathcal{L}_{\text{setupKeys}}[T] \leftarrow (\text{st}_T.\text{puvProtocol}.pk, c)$
- $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow \text{H}_w(\text{pin})$
- $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
- **return** accepted

Figure E.19: Modified CTAP 2.1+ functions setPIN-C and setPIN-T for hybrid game 4.i.

key pair appears later in the simulation, and sets the pinHash corresponding to the pin that was saved in $\mathcal{L}_{\text{pin}}$ in the beginning of the Setup oracle execution.

If $\mathcal{A}$ later queries Send-Bind-T to the target token session π_T^i that was involved in the i^{th} Setup query with the public key c that was used in that Setup (which will be identifiable by the flag set before in setPIN-T), $\mathcal{B}_5^i$ must perform a query to its LHPC oracle $\mathcal{O}_{\text{LHPC}}$ to check if the correct pinHash is present in the received ciphertext. If it's not, then the pinHash verification fails. Otherwise, the verification succeeds, and $\mathcal{B}_5^i$ must now use the RAND oracle $\mathcal{O}_{\text{RAND}}$ to encrypt a random pinUvAuthToken to send to $\mathcal{A}$.

$\mathcal{B}_5^i(\mathcal{A}, \mathcal{O}_{\text{LR}}, \mathcal{O}_{\text{RAND}}, \mathcal{O}_{\text{LHPC}})$:
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- $\text{hy} \leftarrow i, \text{count}_{\text{setup}} \leftarrow 0$
- $pk_T^i, pk_C^i \leftarrow \perp$
- $c_{\text{chal}} \leftarrow \perp$
- win-SUF-t $\leftarrow 0$
- $b' \leftarrow \perp$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **if** win-SUF-t $= 1$ **then**
  - **return** $b' \leftarrow 0$
- **else**
  - **return** $b' \leftarrow 1$
- **return** b'

Setup (T, i, C, j, U):
- **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  - **return** $\perp$
- $\pi_T^i \leftarrow \text{st}_T$
- $\text{count}_{\text{setup}} \leftarrow \text{count}_{\text{setup}} + 1$
- $\mathcal{L}_{\text{pin}}[\pi_T^i] \leftarrow \text{pin}_U$
- $\text{trans} \xleftarrow{\$} \text{Setup}(\pi_T^i, \pi_C^j, \text{pin}_U)$
- $\pi_T^i.\text{isValid}, \pi_C^j.\text{isValid} \leftarrow \text{false}$
- $\text{st}_T.\text{user} \leftarrow U$
- **return** trans

Figure E.20: Adversary $\mathcal{B}_5^i$'s simulation of game 4.i.

Advantage measure

The advantage of $\mathcal{B}_5^i$ against the IND-1$PA-LHPC security of SKE is:

setPIN-C(π_C^j, pin_U):
- **if** $\text{pin}_U \notin \mathcal{P}$ **then**
  - **return** $\perp$
- **if** $\text{count}_{\text{setup}} = \text{hy}$ **then**
  - $c_p \xleftarrow{\$} \mathcal{O}_{\text{LR}}(\text{pin}_U, 0000)$
  - $c_{\text{chal}} \leftarrow c_p$
  - $t_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.authenticate}(\pi_C^j.\text{K}, c_p)$
- **else if** $\text{count}_{\text{setup}} < \text{hy}$ **then**
  - $c_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.encrypt}(\pi_C^j.\text{K}, 0000)$
  - $t_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.authenticate}(\pi_C^j.\text{K}, c_p)$
- **else**
  - $c_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.encrypt}(\pi_C^j.\text{K}, \text{pin}_U)$
  - $t_p \xleftarrow{\$} \pi_C^j.\text{selectedPuvProtocol.authenticate}(\pi_C^j.\text{K}, c_p)$
- **return** (c_p, t_p)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p):
- **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee$ $\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then return** $\perp$
- $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
- **if** $K = \perp \vee \text{st}_T.\text{puvProtocol.verify}(K, c_p, t_p) = \text{false}$ **then return** $\perp$
- **if** $\text{count}_{\text{setup}} = \text{hy}$ **then**
  - $pk_T^i \leftarrow \pi_T^i.\text{puvProtocol}.pk$
  - $pk_C^i \leftarrow c$
- **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
  - $\text{pin} \leftarrow \mathcal{L}_{\text{pin}}[\pi_T^i]$
- **else**
  - $\text{pin} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_p)$
- **if** $\text{pin} \notin \mathcal{P}$ **then return** $\perp$
- **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
  - $\mathcal{L}_{\text{setupKeys}}[T] \leftarrow (\text{st}_T.\text{puvProtocol}.pk, c)$
- $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow \text{H}_w(\text{pin})$
- $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
- **return** accepted

obtainPinUvAuthToken-T(π_T^i, puvProtocol, c, c_{ph}):
- **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  - **return** $(\perp, \text{false})$
- $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
- **if** $K = \perp$ **then**
  - **return** $(\perp, \text{false})$
- $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
- **if** $\pi_T^i.\text{puvProtocol}.pk = pk_T^i \wedge c = pk_C^i$ **then**
  - **if** $c_{ph} = c_{\text{chal}}$ **then**
    - $\text{pin} \leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
    - **if** $\text{H}(\text{pin}) = \text{pin}$ **then**
      - $\text{pinHash} \leftarrow \text{H}(\text{pin})$
    - **else**
      - $\text{pinHash} \leftarrow \perp$
  - **else if** $\mathcal{O}_{\text{LHPC}}(c_{ph}) = \text{true}$ **then**
    - $\text{pinHash} \leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
  - **else**
    - $\text{pinHash} \leftarrow \perp$
- **else**
  - $\text{pinHash} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_{ph})$
- $\text{pinHash}_T \leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
- **if** $\text{pinHash} \neq \text{pinHash}_T$ **then**
  - $\text{st}_T.\text{puvProtocol.regenerate}()$
  - $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
  - **if** $\text{st}_T.m = 0$ **then**
    - $\text{authPowerUp-T}(\text{st}_T)$
    - **return** $(\perp, \text{true})$
  - **else**
    - **return** $(\perp, \text{false})$
- $\text{st}_T.m \leftarrow 3, \text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
- **for all** puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  - $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
- $\pi_T^i.\text{bs} \leftarrow \pi_T^i.\text{puvProtocol}.pt$
- **if** $(\text{st}_T.\text{puvProtocol}.pk, c) = \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  - **if** $\pi_T^i.\text{puvProtocol}.pk = pk_T^i \wedge c = pk_C^i$ **then**
    - $(_, _, c_{pt}) \leftarrow \mathcal{O}_{\text{RAND}}(\text{len}(\pi_C^j.\text{bs}))$
  - **else**
    - $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi_T^i.\text{bs})}$
    - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \tilde{pt})$
- **else**
  - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \pi_T^i.\text{bs})$
- $\pi_T^i.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
- $\pi_T^i.\text{canValidate} \leftarrow \text{true}$
- $\pi_T^i.\text{sid} \leftarrow \pi_T^i.\text{sid} \,||\, \text{puvProtocol} \,||\, c \,||\, c_{ph} \,||\, c_{pt} \,||\, \text{false}$
- **return** (c_{pt}, false)

Figure E.21: CTAP 2.1+ functions setPIN-C, setPIN-T and obtainPinUvAuthToken-T simulated by $\mathcal{B}_5^i$.

$$\text{Adv}_{\text{SKE}}^{\text{IND-1\$PA-LHPC}}(\mathcal{B}_5^i) = |\ \Pr_{\text{IND-1\$PA-LHPC}}[b' = 0 | b = 1] - \Pr_{\text{IND-1\$PA-LHPC}}[b' = 0 | b = 0]\ | \tag{E.18}$$

When oracle $\mathcal{O}_{\text{LR}}$ encrypts m_0, which corresponds to the real pin, then $\mathcal{B}_5^i$ is simulating Game 4.$(i-1)$ to $\mathcal{A}$. If instead it encrypts m_1, corresponding to the constant pin 0000, then it simulates Game 4.i.

From the definition of $\mathcal{B}_5^i$, it follows that:

$$\mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5^i) = \mid \mathsf{Pr}_{4.i}[\text{win-SUF-t} = 1] - \mathsf{Pr}_{4.(i-1)}[\text{win-SUF-t} = 1] \mid, \forall i \in [1, n_{\mathsf{setup}}] \tag{E.19}$$

$$\mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5^i) = \mid \mathsf{Pr}_{4.i} - \mathsf{Pr}_{4.(i-1)} \mid, \forall i \in [1, n_{\mathsf{setup}}] \tag{E.20}$$

$$\sum_{i=1}^{n_{\mathsf{setup}}} \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5^i) = \sum_{i=1}^{n_{\mathsf{setup}}} \mid \mathsf{Pr}_{4.i} - \mathsf{Pr}_{4.(i-1)} \mid \tag{E.21}$$

$$\sum_{i=1}^{n_{\mathsf{setup}}} \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5^i) \geq \mid \mathsf{Pr}_{4.0} - \mathsf{Pr}_{4.n_{\mathsf{setup}}} \mid \tag{E.22}$$

Let $\mathcal{B}_5$ be the adversary with the highest advantage against the IND-1\$PA-LHPC game, $\forall i \in [1, n_{\mathsf{setup}}]$. Then, it is always true that:

$$n_{\mathsf{setup}} \ \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5) \geq \sum_{i=1}^{n_{\mathsf{setup}}} \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5^i) \tag{E.23}$$

And thus,

$$n_{\mathsf{setup}} \ \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5) \geq \mid \mathsf{Pr}_{4.0} - \mathsf{Pr}_{4.n_{\mathsf{setup}}} \mid \tag{E.24}$$

Since Game 4.0 corresponds to Game 4 and Game $4.n_{\mathsf{setup}}$ corresponds to Game 5, we have:

$$n_{\mathsf{setup}} \ \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5) \geq \mid \mathsf{Pr}_4 - \mathsf{Pr}_5 \mid \tag{E.25}$$

And, since n_{setup} is the maximum amount of valid Setup queries made by $\mathcal{A}$, we have that $n_{\mathsf{setup}} \leq q_{\mathsf{S}}$. Therefore:

$$q_{\mathsf{S}} \ \mathsf{Adv}^{\text{IND-1\$PA-LHPC}}_{\mathsf{SKE}}(\mathcal{B}_5) \geq \mid \mathsf{Pr}_4 - \mathsf{Pr}_5 \mid \tag{E.26}$$

Game 6

In this game, we replace all pinHashes encrypted by any client session in Execute with the hash of a constant value 0000. Additionally, for any token session that was not actively attacked by $\mathcal{A}$, we replace the encryption of the pinUvAuthToken pt with the encryption of a uniform random value $\tilde{pt}$. We skip any pinHash verification for any passive attempt (when c and c_{ph} were delivered to the token via an Execute query or when $\mathcal{A}$ delivers some c and c_{ph} to a token via the Send-Bind-T oracle that were output by a client session in a previous Execute, as long as the token is using the same public key). We then bound the advantage of $\mathcal{A}$ by reducing to the IND-1$PA-LPC security of SKE.

Game6 ($\mathcal{A}$):
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
- **return** win-SUF-t

Execute (T, i, C, j):
- $\text{pin}_U \leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
- **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  - **return** $\perp$
- $\pi_T^i \leftarrow \text{st}_T$
- trans, $m_C, m_T \leftarrow \perp$
- ▷ *Each π_C^j and π_T^i can be used only once in each Execute, Send-Bind-T and Send-Bind-C.*
- $\mathcal{L}_{\text{ct}}[\pi_C^j, \pi_T^i] \leftarrow (\perp, \perp, \perp)$
- **while** $\pi_C^j.\text{st}_{\text{exe}} \neq \text{bindDone}$ **do**
  - $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
  - $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
  - trans $\leftarrow$ trans $||\ m_T\ ||\ m_C$
- **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
  - $\pi_T^{i'}.\text{isValid} \leftarrow$ false
- **return** trans

obtainPinUvAuthToken-C-start(π_C^j, pin):
- $\text{pinHash} \leftarrow \text{H}_w(0000)$
- $c_{ph} \xleftarrow{\$} \pi_C^j.\text{puvProtocol.encrypt}(\pi_C^j.\text{K}, \text{pinHash})$
- $\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindStart
- $\pi_C^j.\text{sid} \leftarrow \pi_C^j.\text{sid}\ ||\ c_{ph}$
- $\mathcal{L}_{\text{ct}}[\pi_C^j, _] \leftarrow (\pi_C^j.\text{puvProtocol}.pk, c_{ph}, \perp)$
- **return** c_{ph}

obtainPinUvAuthToken-C-end(π_C^j, c_{pt}):
- $(_, _, pt) \leftarrow \mathcal{L}_{\text{ct}}[\pi_C^j, _]$
- $\pi_C^j.\text{bs} \leftarrow pt$
- $\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindDone
- $\pi_C^j.\text{sid} \leftarrow \pi_C^j.\text{sid}\ ||\ c_{pt}\ ||$ false
- **return**

obtainPinUvAuthToken-T$(\pi_T^i, \text{puvProtocol}, c, c_{ph})$:
- **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  - **return** $(\perp,$ false$)$
- $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
- **if** $K = \perp$ **then**
  - **return** $(\perp,$ false$)$
- $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
- $\text{active}_{\text{key}}, \text{active}_{c_{ph}} \leftarrow$ true
- **for all** $(\pi_{C'}^{j'}, \pi_{T'}^{i'}, c', c'_{ph}, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T \wedge \pi_{T'}^{i'}.\text{puvProtocol}.pk = \pi_T^i.\text{puvProtocol}.pk$ **do**
  - **if** $c' = c$ **then**
    - $\text{active}_{\text{key}} \leftarrow$ false
  - **if** $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    - $\text{active}_{c_{ph}} \leftarrow$ false
    - **break**
- $\text{verifyPin} \leftarrow \text{active}_{\text{key}} \vee \text{active}_{c_{ph}}$
- **if** verifyPin $=$ true **then**
  - $\text{pinHash} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_{ph})$
  - $\text{pinHash}_T \leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
  - **if** $\text{pinHash} \neq \text{pinHash}_T$ **then**
    - $\text{st}_T.\text{puvProtocol.regenerate}()$
    - $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
    - **if** $\text{st}_T.m = 0$ **then**
      - authPowerUp-T(st_T)
      - **return** $(\perp,$ true$)$
    - **else**
      - **return** $(\perp,$ false$)$
- $\text{st}_T.m \leftarrow 3, \text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
- **for all** puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  - $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
- $\pi_T^i.\text{bs} \leftarrow \pi_T^i.\text{puvProtocol}.pt$
- **if** $\text{active}_{\text{key}} = \text{true} \wedge (\text{st}_T.\text{puvProtocol}.pk, c) \neq \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \pi_T^i.\text{bs})$
- **else**
  - $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi_T^i.\text{bs})}$
  - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \tilde{pt})$
  - $\mathcal{L}_{\text{ct}}[_, \pi_T^i] \leftarrow (c', c'_{ph}, \pi_T^i.\text{bs})$
- $\pi_T^i.\text{st}_{\text{exe}} \leftarrow$ bindDone
- $\pi_T^i.\text{canValidate} \leftarrow$ true
- $\pi_T^i.\text{sid} \leftarrow \pi_T^i.\text{sid}\ ||\ \text{puvProtocol}\ ||\ c\ ||\ c_{ph}\ ||\ c_{pt}\ ||$ false
- **return** $(c_{pt},$ false$)$

Figure E.22: Game 6, modified Execute oracle and modified CTAP 2.1+ functions obtainPinUvAuthToken-C-start, obtainPinUvAuthToken-C-end and obtainPinUvAuthToken-T - Game 6.

We create a new structure $\mathcal{L}_{\text{ct}}$ which saves important information from the client session, namely its public key c and ciphertext c_{ph} containing the constant pinHash. Then, the game behaves differently depending on which oracle called obtainPinUvAuthToken-T.

If c and c_{ph} are transmitted to a token session π_T^i during an Execute query by a client session π_C^j, which is guaranteed to be passive, then the pinHash verification is skipped, and a new random pinUvAuthToken is encrypted instead of the real pinUvAuthToken generated by π_T^i that is set as the binding state π_T^i.bs and saved in $\mathcal{L}_{\text{ct}}$. The ciphertext c_{pt} is then sent to π_C^j, which simply discards it and sets the same binding state as π_T^i.

If c and c_{ph} are sent by $\mathcal{A}$ to a token session π_T^i via a Send-Bind-T query, the received c and c_{ph} are treated differently in obtainPinUvAuthToken-T depending on their origin. If both c and c_{ph} were created by a client session $\pi_C^{j'}$ that completed a Bind with a different token session $\pi_T^{i'}$ via an Execute query, and $\pi_T^{i'}$ and π_T^i share the same ECDH public key, then the experiment skips the pinHash verification (we say that c and c_{ph} are passive). Recall that any Bind session from the Execute oracle is passive, which means that client sessions involved in an Execute query for some token T must have not only received T's public key, but also be the only client sessions to have received that key (collisions between public keys were removed in a previous game, ruling out different tokens with the same public key). Any one of those client sessions, as a result, encrypts the same pin in the original experiment, in such a way that any of T's sessions can always correctly decrypt into the pinHash if they derive the same session key K. Therefore, a token session π_T^i can always determine that, as long as $\mathcal{A}$ delivers a c and c_{ph} that were created in a client session that received π_T^i's ECDH public key, c_{ph} contains the correct pinHash.

If c has never appeared in an Execute query with token T before, then π_T^i must derive the session key normally, since it means that $\mathcal{A}$ has started a Bind session with π_T^i without any client session being involved, and decrypt c_{ph} normally. If c has appeared in a client session π_C^j before, and π_T^i's ECDH public key is the same that π_C^j received, but c_{ph} is different, then it must be decrypted normally in order to verify the decrypted pinHash.

After the verification step, if c is passive, a new random pinUvAuthToken $\tilde{pt}$ is encrypted instead of the real pinUvAuthToken pt that was set as the binding state, resulting in a ciphertext c_{pt}. The resulting binding state is stored in $\mathcal{L}_{\text{ct}}$, and c_{pt} is sent to $\mathcal{A}$.

Hybrid Argument

To bound $\mathcal{A}$'s advantage against Game 6, we begin by creating n_{execute} hybrid games, where n_{execute} is the maximum number of (valid) calls to the Execute oracle (valid refers to calls that actually result in executing the Execute between a client session and token session). We replace the hashed pin with the hash of the constant value 0000 in client sessions and stop pinHash verification and replace pinUvAuthToken encryption with the encryption of a random value in token sessions only until the i^{th} execution of Execute.

Additionally, we also skip verifications of the pinHash in any token session that receives passive c and c_{ph} that were involved in the first i^{th} Execute queries, as long as the token session is reusing the same ECDH public key, and replace all pinUvAuthTokens with random values in any token session that receives a passive c and passes pinHash verification, as long as c came from one of the first i^{th} Execute queries and the token session is reusing the same ECDH public key.

A new temporary structure $\mathcal{L}_{\text{executePk}}$ is created so that each client and token session can be tracked throughout the experiment, by linking each client and token ECDH public key pair with a specific Execute query. This allows us to maintain consistency with $\mathcal{A}$ whenever it queries Send-Bind-T with an ECDH public key c that was used by a previous client session during Execute.

Reduction to the IND-1$PA-LPC security of SKE

We now construct an adversary $\mathcal{B}_6^i$ against the IND-1$PA-LPC security of SKE that simulates game $5.i$ to $\mathcal{A}$ perfectly, except when $\mathcal{A}$ performs the i^{th} valid query to Execute. In this case, $\mathcal{B}_6^i$ sends $\mathsf{H}(\mathsf{pin})$ and $\mathsf{H}(0000)$ to $\mathcal{O}_{\mathsf{LR}}$ and receives a challenge ciphertext c_{ph}, instead of encrypting it normally with the derived session key $\tilde{K}_2$. When these are delivered to the token session π_T^i, verification always succeeds when performed during Execute. $\mathcal{B}_6^i$ calls $\mathcal{O}_{\mathsf{RAND}}$ to produce ciphertext c_{pt} and saves pt_0 as the binding state. Since this was all during Execute, c_{ph} is delivered to the client session π_C^j, which simply keeps the correct binding state pt_0.

If $\mathcal{A}$ calls Send-Bind-T on a token session π_T^i such that the provided public key c and π_T^i's public key pk were involved in the i^{th} Execute query, but c_{ph} was not the output ciphertext (c_{ph} is not the challenge ciphertext output by $\mathcal{O}_{\mathsf{LR}}$), then $\mathcal{B}_6^i$ must use the $\mathcal{O}_{\mathsf{LPC}}$ oracle to check if pinHash is in the received ciphertext c_{ph}. In this situation, if the verification

Game5.i ($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
$\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
$\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
$\text{hy} \leftarrow i, \text{count}_{\text{execute}} \leftarrow 0, \mathcal{L}_{\text{executePk}} \leftarrow \emptyset$
$\pi^*_C, \pi^*_T \leftarrow \perp$
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
return win-SUF-t

Execute (T, i, C, j):
$\text{pin}_U \leftarrow \mathcal{L}_{\text{valid}}[U]$
if $\text{st}_T = \perp$ **or** $\pi^i_T \neq \perp$ **or** $\pi^j_C \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  return $\perp$
$\pi^i_T \leftarrow \text{st}_T$
trans, $m_C, m_T \leftarrow \perp$
$\mathcal{L}_{\text{ct}}[\pi^j_C, \pi^i_T] \leftarrow (\perp, \perp, \perp)$
$\text{count}_{\text{execute}} \leftarrow \text{count}_{\text{execute}} + 1$
$\pi^*_C \leftarrow \pi^j_C, \pi^*_T \leftarrow \pi^i_T$
while $\pi^j_C.\text{st}_{\text{exe}} \neq \text{bindDone}$ **do**
  $m_T \xleftarrow{\$} \text{Bind-T}(\pi^i_T, m_C)$
  $m_C \xleftarrow{\$} \text{Bind-C}(\pi^j_C, U, m_T)$
  trans $\leftarrow$ trans $||\ m_T\ ||\ m_C$
for all $i' \neq i$ **and** $\pi^{i'}_T \neq \perp$ **do**
  $\pi^{i'}_T.\text{isValid} \leftarrow$ false
return trans

obtainPinUvAuthToken-C-start(π^j_C, pin):
if $\text{count}_{\text{execute}} \leq \text{hy}$ **then**
  $\text{pinHash} \leftarrow \text{H}_w(0000)$
else
  $\text{pinHash} \leftarrow \text{H}_w(\text{pin})$
$pk_{\text{client}} \leftarrow \pi^*_C.\text{puvProtocol}.pk$
$pk_{\text{token}} \leftarrow \pi^*_T.\text{puvProtocol}.pk$
$\mathcal{L}_{\text{executePk}}[pk_{\text{client}}, pk_{\text{token}}] \leftarrow \text{count}_{\text{execute}}$
$c_{ph} \xleftarrow{\$} \pi^j_C.\text{puvProtocol.encrypt}(\pi^j_C.\text{K}, \text{pinHash})$
$\pi^j_C.\text{st}_{\text{exe}} \leftarrow \text{bindStart}$
$\pi^j_C.\text{sid} \leftarrow \pi^j_C.\text{sid}\ ||\ c_{ph}$
$\mathcal{L}_{\text{ct}}[\pi^j_C, _] \leftarrow (\pi^j_C.\text{puvProtocol}.pk, c_{ph}, \perp)$
return c_{ph}

obtainPinUvAuthToken-C-end(π^j_C, c_{pt}):
if $\text{count}_{\text{execute}} \leq \text{hy}$ **then**
  $(_, _, pt) \leftarrow \mathcal{L}_{\text{ct}}[\pi^j_C, _]$
  $\pi^j_C.\text{bs} \leftarrow pt$
else
  $\pi^j_C.\text{bs} \leftarrow \pi^j_C.\text{puvProtocol.decrypt}(\pi^j_C.\text{K}, c_{pt})$
$\pi^j_C.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
$\pi^j_C.\text{sid} \leftarrow \pi^j_C.\text{sid}\ ||\ c_{pt}\ ||$ false
return

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
if puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  return $(\perp,$ false$)$
$K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
if $K = \perp$ **then**
  return $(\perp,$ false$)$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
$\text{active}_{\text{key}}, \text{active}_{c_{ph}} \leftarrow \text{true}$
for all $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T \wedge \pi^{i'}_{T'}.\text{puvProtocol}.pk = \pi^i_T.\text{puvProtocol}.pk$ **do**
  if $c' = c$ **then**
    $\text{active}_{\text{key}} \leftarrow$ false
  if $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    $\text{active}_{c_{ph}} \leftarrow$ false
    break
$\text{verifyPin} \leftarrow \text{active}_{\text{key}} \vee \text{active}_{c_{ph}}$
$\text{count}_{\text{execute}} \leftarrow \mathcal{L}_{\text{executePk}}[c, \pi^i_T.\text{puvProtocol}.pk]$
if $\text{verifyPin} = \text{true} \vee \text{count}_{\text{execute}} > \text{hy}$ **then**
  $\text{pinHash} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_{ph})$
  $\text{pinHash}_T \leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
  if $\text{pinHash} \neq \text{pinHash}_T$ **then**
    $\text{st}_T.\text{puvProtocol.regenerate}()$
    $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
    if $\text{st}_T.m = 0$ **then**
      $\text{authPowerUp-T}(\text{st}_T)$
      return $(\perp,$ true$)$
    else
      return $(\perp,$ false$)$
$\text{st}_T.m \leftarrow 3, \text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
for all puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
$\pi^i_T.\text{bs} \leftarrow \pi^i_T.\text{puvProtocol}.pt$
if $(\text{active}_{\text{key}} = \text{true} \vee \text{count}_{\text{execute}} > \text{hy}) \wedge (\text{st}_T.\text{puvProtocol}.pk, c) \neq \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \pi^i_T.\text{bs})$
else
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\text{bs})}$
  $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \tilde{pt})$
  $\mathcal{L}_{\text{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, \pi^i_T.\text{bs})$
$\pi^i_T.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
$\pi^i_T.\text{canValidate} \leftarrow \text{true}$
$\pi^i_T.\text{sid} \leftarrow \pi^i_T.\text{sid}\ ||$ puvProtocol $||\ c\ ||\ c_{ph}\ ||\ c_{pt}\ ||$ false
return $(c_{pt},$ false$)$

Figure E.23: Hybrid game 5.i, Execute oracle and modified CTAP 2.1+ functions obtainPinUvAuthToken-C-start, obtainPinUvAuthToken-C-end and obtainPinUvAuthToken-T from hybrid game 5.i.

succeeds, then $\mathcal{B}_6^i$ still uses the $\mathcal{O}_{\mathsf{RAND}}$ oracle to encrypt a random pinUvAuthToken, and delivers c_{pt} to $\mathcal{A}$.

$\mathcal{B}_6^i(\mathcal{A}, \mathcal{O}_{\mathsf{LR}}, \mathcal{O}_{\mathsf{RAND}}, \mathcal{O}_{\mathsf{LPC}})$:
- $\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{valid}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pin}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{ct}} \leftarrow \emptyset$
- $\mathsf{hy} \leftarrow i, \mathsf{count}_{\mathsf{executePk}} \leftarrow 0, \mathcal{L}_{\mathsf{execute}} \leftarrow \emptyset$
- $\pi_C^* \leftarrow \perp, \pi_T^* \leftarrow \perp$
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- $b' \leftarrow \perp$
- **if** win-SUF-t $= 1$ **then**
  - **return** $b' \leftarrow 0$
- **else**
  - **return** $b' \leftarrow 1$
- **return** b'

Execute (T, i, C, j):
- $\mathsf{pin}_U \leftarrow \mathcal{L}_{\mathsf{valid}}[U]$
- **if** $\mathsf{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\mathsf{pin}_U = \perp$ **then**
  - **return** $\perp$
- $\pi_T^i \leftarrow \mathsf{st}_T$
- trans, $m_C, m_T \leftarrow \perp$
- $\mathcal{L}_{\mathsf{ct}}[\pi_C^j, \pi_T^i] \leftarrow (\perp, \perp, \perp)$
- $\mathsf{count}_{\mathsf{execute}} \leftarrow \mathsf{count}_{\mathsf{execute}} + 1$
- $\pi_C^* \leftarrow \pi_C^j, \pi_T^* \leftarrow \pi_T^i$
- **while** $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \neq$ bindStart **do**
  - $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
  - $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
  - trans $\leftarrow$ trans $\|\, m_T \,\|\, m_C$
- **return** trans

obtainPinUvAuthToken-C-start(π_C^j, pin):
- **if** $\mathsf{count} = \mathsf{hy}$ **then**
  - $c_{ph} \xleftarrow{\$} \mathcal{O}_{\mathsf{LR}}(\mathsf{H_w}(\mathsf{pin}), \mathsf{H_w}(0000))$
- **else if** $\mathsf{count}_{\mathsf{execute}} \leq \mathsf{hy}$ **then**
  - $\mathsf{pinHash} \leftarrow \mathsf{H_w}(0000)$
  - $c_{ph} \xleftarrow{\$} \pi_C^j.\mathsf{puvProtocol.encrypt}(\pi_C^j.\mathsf{K}, \mathsf{pinHash})$
- **else**
  - $\mathsf{pinHash} \leftarrow \mathsf{H_w}(\mathsf{pin})$
  - $c_{ph} \xleftarrow{\$} \pi_C^j.\mathsf{puvProtocol.encrypt}(\pi_C^j.\mathsf{K}, \mathsf{pinHash})$
- $pk_{\mathsf{client}} \leftarrow \pi_C^*.\mathsf{puvProtocol}.pk$
- $pk_{\mathsf{token}} \leftarrow \pi_T^*.\mathsf{puvProtocol}.pk$
- $\mathcal{L}_{\mathsf{executePk}}[pk_{\mathsf{client}}, pk_{\mathsf{token}}] \leftarrow \mathsf{count}_{\mathsf{execute}}$
- $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindStart
- $\pi_C^j.\mathsf{sid} \leftarrow \pi_C^j.\mathsf{sid} \,\|\, c_{ph}$
- $\mathcal{L}_{\mathsf{ct}}[\pi_C^j, _] \leftarrow (\pi_C^j.\mathsf{puvProtocol}.pk, c_{ph}, \perp)$
- **return** c_{ph}

obtainPinUvAuthToken-C-end(π_C^j, c_{pt}):
- **if** $\mathsf{count}_{\mathsf{execute}} \leq \mathsf{hy}$ **then**
  - $(_, _, pt) \leftarrow \mathcal{L}_{\mathsf{ct}}[\pi_C^j, _]$
  - $\pi_C^j.\mathsf{bs} \leftarrow pt$
- **else**
  - $\pi_C^j.\mathsf{bs} \leftarrow \pi_C^j.\mathsf{puvProtocol.decrypt}(\pi_C^j.\mathsf{K}, c_{pt})$
- $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindDone
- $\pi_C^j.\mathsf{sid} \leftarrow \pi_C^j.\mathsf{sid} \,\|\, c_{pt} \,\|$ false
- **return**

obtainPinUvAuthToken-T$(\pi_T^i, \mathsf{puvProtocol}, c, c_{ph})$:
- **if** $\mathsf{puvProtocol} \notin \mathsf{st}_T.\mathsf{puvProtocolList} \vee \mathsf{st}_T.\mathsf{pinRetries} = 0$ **then**
  - **return** $(\perp,$ false$)$
- $K \leftarrow \mathsf{st}_T.\mathsf{puvProtocol.decapsulate}(c)$
- **if** $K = \perp$ **then**
  - **return** $(\perp,$ false$)$
- $\mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetries} - 1$
- $\mathsf{active}_{\mathsf{key}}, \mathsf{active}_{c_{ph}} \leftarrow$ true
- **for all** $(\pi_{C'}^{j'}, \pi_{T'}^{i'}, c', c'_{ph}, _) \in \mathcal{L}_{\mathsf{ct}}$ s.t. $T' = T \wedge \pi_{T'}^{i'}.\mathsf{puvProtocol}.pk = \pi_T^i.\mathsf{puvProtocol}.pk$ **do**
  - **if** $c' = c$ **then**
    - $\mathsf{active}_{\mathsf{key}} \leftarrow$ false
    - **if** $c'_{ph} = c_{ph} \wedge c' = c$ **then**
      - $\mathsf{active}_{c_{ph}} \leftarrow$ false
      - **break**
- $\mathsf{verifyPin} \leftarrow \mathsf{active}_{\mathsf{key}} \vee \mathsf{active}_{c_{ph}}$
- $\mathsf{count}_{\mathsf{execute}} \leftarrow \mathcal{L}_{\mathsf{executePk}}[c, \pi_T^i.\mathsf{puvProtocol}.pk]$
- **if** $\mathsf{verifyPin} =$ true $\vee\ \mathsf{count}_{\mathsf{execute}} > \mathsf{hy}$ **then**
  - $\mathsf{pinHash}_T \leftarrow \mathcal{L}_{\mathsf{TokenSetup}}[T]$
  - **if** $\mathsf{count}_{\mathsf{execute}} = \mathsf{hy}$ **then**
    - **if** $\mathcal{O}_{\mathsf{LPC}}(c_{ph}) =$ true **then**
      - $\mathsf{pinHash} \leftarrow \mathsf{pinHash}_T$
    - **else**
      - $\mathsf{pinHash} \leftarrow \perp$
  - **else**
    - $\mathsf{pinHash} \leftarrow \mathsf{st}_T.\mathsf{puvProtocol.decrypt}(K, c_{ph})$
  - **if** $\mathsf{pinHash} \neq \mathsf{pinHash}_T$ **then**
    - $\mathsf{st}_T.\mathsf{puvProtocol.regenerate}()$
    - $\mathsf{st}_T.m \leftarrow \mathsf{st}_T.m - 1$
    - **if** $\mathsf{st}_T.m = 0$ **then**
      - authPowerUp-T(st_T)
      - **return** $(\perp,$ true$)$
    - **else**
      - **return** $(\perp,$ false$)$
- $\mathsf{st}_T.m \leftarrow 3, \mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetriesMax}$
- **for all** puvProtocol' $\in \mathsf{st}_T.\mathsf{puvProtocolList}$ **do**
  - $\mathsf{st}_T.\mathsf{puvProtocol'.resetPuvToken}()$
- $\pi_T^i.\mathsf{bs} \leftarrow \pi_T^i.\mathsf{puvProtocol}.pt$
- **if** $(\mathsf{active}_{\mathsf{key}} =$ true $\vee\ \mathsf{count}_{\mathsf{execute}} > \mathsf{hy}) \wedge (\mathsf{st}_T.\mathsf{puvProtocol}.pk, c) \neq \mathcal{L}_{\mathsf{setupKeys}}[T]$ **then**
  - $c_{pt} \xleftarrow{\$} \mathsf{st}_T.\mathsf{puvProtocol.encrypt}(K, \pi_T^i.\mathsf{bs})$
- **else if** $\mathsf{count}_{\mathsf{execute}} = \mathsf{hy}$ **then**
  - $(pt_0, pt_1, c_{pt}) \xleftarrow{\$} \mathcal{O}_{\mathsf{RAND}}(\mathsf{len}(\pi_T^i.\mathsf{bs}))$
  - $\pi_T^i.\mathsf{bs} \leftarrow pt_0$
  - $\mathcal{L}_{\mathsf{ct}}[_, \pi_T^i] \leftarrow (c', c'_{ph}, \pi_T^i.\mathsf{bs})$
- **else**
  - $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi_T^i.\mathsf{bs})}$
  - $c_{pt} \xleftarrow{\$} \mathsf{st}_T.\mathsf{puvProtocol.encrypt}(K, \tilde{pt})$
  - $\mathcal{L}_{\mathsf{ct}}[_, \pi_T^i] \leftarrow (c', c'_{ph}, \pi_T^i.\mathsf{bs})$
- $\pi_T^i.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindDone
- $\pi_T^i.\mathsf{canValidate} \leftarrow$ true
- $\pi_T^i.\mathsf{sid} \leftarrow \pi_T^i.\mathsf{sid} \,\|\, \mathsf{puvProtocol} \,\|\, c \,\|\, c_{ph} \,\|\, c_{pt} \,\|$ false
- **return** $(c_{pt},$ false$)$

Figure E.24: Adversary $\mathcal{B}_6^i$'s simulation of game 5.i, Execute oracle and modified CTAP 2.1+ functions obtainPinUvAuthToken-C-start, obtainPinUvAuthToken-C-end and obtainPinUvAuthToken-T.

Advantage measure

The advantage of $\mathcal{B}_6^i$ against the IND-1$PA-LPC security of SKE during game $5.i$ is:

$$\mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) = \mid \mathsf{Pr}_{\text{IND-1\$PA-LPC}}[b' = 0 | b = 1] - \mathsf{Pr}_{\text{IND-1\$PA-LPC}}[b' = 0 | b = 0] \mid \tag{E.27}$$

When oracle $\mathcal{O}_{\mathsf{LR}}$ encrypts m_0, which corresponds to $\mathsf{H}(\mathsf{pin})$, the oracle $\mathcal{O}_{\mathsf{RAND}}$ encrypts pt_0, which is the pinUvAuthToken being stored as the binding state in π_T^i.bs. Then, $\mathcal{B}_6^i$ is simulating Game $5.(i-1)$ to $\mathcal{A}$, where both c_{ph} and c_{pt} have the correct pinHash and pinUvAuthToken respectively. If $\mathcal{O}_{\mathsf{LR}}$ instead encrypts m_1, which is $\mathsf{H}(0000)$, then $\mathcal{O}_{\mathsf{RAND}}$ encrypts pt_1, and $\mathcal{B}_6^i$ is simulating Game $5.i$.

From the definition of $\mathcal{B}_6^i$, it follows that:

$$\mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) = \mid \mathsf{Pr}_{5.i}[\text{win-SUF-t} = 1] - \mathsf{Pr}_{5.(i-1)}[\text{win-SUF-t} = 1] \mid, \forall i \in [1, n_{\text{execute}}] \tag{E.28}$$

$$\mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) = \mid \mathsf{Pr}_{5.i} - \mathsf{Pr}_{5.(i-1)} \mid, \forall i \in [1, n_{\text{execute}}] \tag{E.29}$$

$$\sum_{i=1}^{n_{\text{execute}}} \mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) = \sum_{i=1}^{n_{\text{execute}}} \mid \mathsf{Pr}_{5.i} - \mathsf{Pr}_{5.(i-1)} \mid \tag{E.30}$$

$$\sum_{i=1}^{n_{\text{execute}}} \mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) \geq \mid \mathsf{Pr}_{5.0} - \mathsf{Pr}_{5.n_{\text{execute}}} \mid \tag{E.31}$$

Let $\mathcal{B}_6$ be the adversary with the highest advantage against the IND-1$PA-LPC game out of all adversaries $\mathcal{B}_6^i$, $\forall i \in [1, n_{\text{execute}}]$. Then, it is always true that:

$$n_{\text{execute}}\ \mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6) \geq \sum_{i=1}^{n_{\text{execute}}} \mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6^i) \tag{E.32}$$

It then follows that:

$$n_{\text{execute}}\ \mathsf{Adv}_{\mathsf{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6) \geq \mid \mathsf{Pr}_{5.0} - \mathsf{Pr}_{5.n_{\text{execute}}} \mid \tag{E.33}$$

Since Game 5.0 corresponds to Game 5 and Game $5.n_{\text{execute}}$ corresponds to Game 6, we have:

$$n_{\text{execute}} \, \text{Adv}_{\text{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6) \geq | \, \text{Pr}_5 - \text{Pr}_6 \, | \tag{E.34}$$

And, since n_{execute} is the maximum amount of valid Execute queries made by $\mathcal{A}$, we have that $n_{\text{execute}} \leq q_{\text{E}}$. Therefore:

$$q_{\text{E}} \, \text{Adv}_{\text{SKE}}^{\text{IND-1\$PA-LPC}}(\mathcal{B}_6) \geq | \, \text{Pr}_5 - \text{Pr}_6 \, | \tag{E.35}$$

Game 7

In this game, no user pins are sampled until the pin is necessary to respond to some interaction with $\mathcal{A}$. This happens *only* when $\mathcal{A}$ corrupts a user U to obtain its pin, or when it provides a pinHash of its own when actively attacking token T during the Bind session by calling Send-Bind-T, where the pinHash must be verified.

Advantage measure

The modifications in Game 7 affect *only* when a pin is sampled from a distribution $\mathscr{D}$ over $\mathcal{P}$, and in such a way that it is identical for an adversary to play Game 6 and Game 7.

In fact, since each sampling of a pin is independent from any other sampling of any other pin, it is irrelevant when that sampling occurs, as long as it occurs before the adversary interacts with said pin. Thus, we can sample the pin *exactly* when one of these events occurs, and the game proceeds identically to Game 6.

Therefore,

$$\text{Pr}_6 = \text{Pr}_7 \tag{E.36}$$

Game 8

In this game, all of $\mathcal{A}'s$ active attempts at guessing the pin for a given token are rejected, unless $\mathcal{A}$ has corrupted the user pin associated with the target token.

Recall that at this point, $\mathcal{A}$ has no information whatsoever about any pin for any token whose user it has not already corrupted. No pin was ever sampled until this point, and all ciphertexts sent by any client session contain constant values.

Game7($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
$\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
$\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF-t

NewU (U):
if $\mathcal{L}_{\text{valid}}[U] = \perp$ **then**
  pin $\leftarrow 0$
  $\mathcal{L}_{\text{valid}}[U] \leftarrow$ pin
  $\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ false
  $\mathcal{L}_{\text{sampled}}[U] \leftarrow$ false
return

CorruptUser (U):
if $U \notin \mathcal{L}_{\text{valid}}$ **then return** $\perp$
if $\mathcal{L}_{\text{sampled}}[U] =$ false **then**
  pin $\xleftarrow{\$} \mathcal{P}$
  $\mathcal{L}_{\text{valid}}[U] \leftarrow$ pin
  $\mathcal{L}_{\text{sampled}}[U] \leftarrow$ true
$\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ true
pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
return pin

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then**
  return ($\perp$, false)
$K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
if $K = \perp$ **then**
  return ($\perp$, false)
st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
active$_{\text{key}}$, active$_{c_{ph}} \leftarrow$ true
for all $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T \wedge \pi^{i'}_{T'}$.puvProtocol.$pk =$ π^i_T.puvProtocol.pk **do**
  if $c' = c$ **then**
    active$_{\text{key}} \leftarrow$ false
  if $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    active$_{c_{ph}} \leftarrow$ false
    break
verifyPin $\leftarrow$ active$_{\text{key}} \vee$ active$_{c_{ph}}$
if verifyPin $=$ true **then**
  pinHash $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_{ph})
  if $\mathcal{L}_{\text{sampled}}[\text{st}_T.\text{user}] =$ false **then**
    pin $\xleftarrow{\$} \mathcal{P}$
    $\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}] \leftarrow$ pin
    $\mathcal{L}_{\text{sampled}}[\text{st}_T.\text{user}] \leftarrow$ true
  pinHash$_T \leftarrow \text{H}_w(\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}])$
  if pinHash $\neq$ pinHash$_T$ **then**
    st$_T$.puvProtocol.regenerate()
    st$_T.m \leftarrow$ st$_T.m - 1$
    if st$_T.m = 0$ **then**
      authPowerUp-T(st$_T$)
      return ($\perp$, true)
    else
      return ($\perp$, false)
st$_T.m \leftarrow 3$, st$_T$.pinRetries $\leftarrow$ pinRetriesMax
for all puvProtocol' $\in$ st$_T$.puvProtocolList **do**
  st$_T$.puvProtocol'.resetPuvToken()
π^i_T.bs $\leftarrow \pi^i_T$.puvProtocol.pt
if active$_{\text{key}} =$ true $\wedge$ (st$_T$.puvProtocol.$pk, c) \neq \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt(K, π^i_T.bs)
else
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\text{bs})}$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt($K, \tilde{pt}$)
  $\mathcal{L}_{\text{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, \pi^i_T.\text{bs})$
π^i_T.st$_{\text{exe}} \leftarrow$ bindDone
π^i_T.canValidate $\leftarrow$ true
π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || c || c_{ph} || c_{pt} || false
return (c_{pt}, false)

Figure E.25: Game 7, with modified NewU and CorruptUser oracles, and modified CTAP 2.1+ function obtainPinUvAuthToken-T - Game 7.

Hybrid Argument

We bound the advantage of $\mathcal{A}$ against Game 8 by considering $q^{\text{act}}_{\text{Send}}$ hybrid games, where $q^{\text{act}}_{\text{Send}}$ represents the total amount of active queries made by $\mathcal{A}$ to Send-Bind-T (where the pin must be verified). On game 7.i, in the first i^{th} active queries to Send-Bind-T, every attempt at guessing the pin is rejected. On the i^{th} attempt, we define a bad event that causes the experiment to abort and $\mathcal{A}$ to lose if $\mathcal{A}$ guesses the correct pin but the challenger still rejects.

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
- **if** puvProtocol $\notin$ st_T.puvProtocolList $\vee$ st_T.pinRetries $= 0$ **then**
  - **return** $(\perp, \text{false})$
- $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
- **if** $K = \perp$ **then**
  - **return** $(\perp, \text{false})$
- $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
- $\text{active}_{\text{key}}, \text{active}_{c_{ph}} \leftarrow \text{true}$
- **for all** $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T \wedge \pi^{i'}_{T'}.\text{puvProtocol}.pk = \pi^i_T.\text{puvProtocol}.pk$ **do**
  - **if** $c' = c$ **then**
    - $\text{active}_{\text{key}} \leftarrow \text{false}$
  - **if** $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    - $\text{active}_{c_{ph}} \leftarrow \text{false}$
    - break
- $\text{verifyPin} \leftarrow \text{active}_{\text{key}} \vee \text{active}_{c_{ph}}$
- **if** verifyPin $=$ true **then**
  - $\text{pinHash} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_{ph})$
  - ▷ *NOTE: The value of $\mathcal{L}_{sampled}[st_T.user]$ will always be true if the User has been corrupted. Therefore, it is kept only for simplifying the hybrid game's code.*
  - **if** $\mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}] = \text{true} \wedge \mathcal{L}_{\text{sampled}}[\text{st}_T.\text{user}] = \text{false}$ **then**
    - $\text{pin} \xleftarrow{\$} \mathcal{P}$
    - $\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}] \leftarrow \text{pin}$
    - $\mathcal{L}_{\text{sampled}}[\text{st}_T.\text{user}] \leftarrow \text{true}$
  - $\text{pinHash}_T \leftarrow \text{H}_w(\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}])$
  - **if** $\mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}] = \text{false} \vee \text{pinHash} \neq \text{pinHash}_T$ **then**
    - $\text{st}_T.\text{puvProtocol.regenerate}()$
    - $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
    - **if** $\text{st}_T.m = 0$ **then**
      - $\text{authPowerUp-T}(\text{st}_T)$
      - **return** $(\perp, \text{true})$
    - **else**
      - **return** $(\perp, \text{false})$
- $\text{st}_T.m \leftarrow 3, \text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
- **for all** puvProtocol' $\in$ st_T.puvProtocolList **do**
  - $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
- $\pi^i_T.\text{bs} \leftarrow \pi^i_T.\text{puvProtocol}.pt$
- **if** $\text{active}_{\text{key}} = \text{true} \wedge (\text{st}_T.\text{puvProtocol}.pk, c) \neq \mathcal{L}_{\text{setupKeys}}[T]$ **then**
  - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \pi^i_T.\text{bs})$
- **else**
  - $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\text{bs})}$
  - $c_{pt} \xleftarrow{\$} \text{st}_T.\text{puvProtocol.encrypt}(K, \tilde{pt})$
  - $\mathcal{L}_{\text{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, \pi^i_T.\text{bs})$
- $\pi^i_T.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
- $\pi^i_T.\text{canValidate} \leftarrow \text{true}$
- $\pi^i_T.\text{sid} \leftarrow \pi^i_T.\text{sid} \,||\, \text{puvProtocol} \,||\, c \,||\, c_{ph} \,||\, c_{pt} \,||\, \text{false}$
- **return** (c_{pt}, false)

Figure E.26: Modified CTAP 2.1+ obtainPinUvAuthToken-T function - Game 8.

Game7.i($\mathcal{A}$):
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
- $\text{hy} \leftarrow i, \text{count}_{\text{ActSendT}} \leftarrow 0$
- $\text{bad} \leftarrow \text{false}$
- $\text{win-SUF-t} \leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **return** win-SUF-t

Figure E.27: Hybrid game 7.i.

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
- **if** puvProtocol $\notin \mathsf{st}_T$.puvProtocolList $\vee$ st_T.pinRetries $= 0$ **then**
  - **return** $(\perp, \text{false})$
- $K \leftarrow \mathsf{st}_T$.puvProtocol.decapsulate$(c)$
- **if** $K = \perp$ **then**
  - **return** $(\perp, \text{false})$
- st_T.pinRetries $\leftarrow$ pinRetries $- 1$
- $\mathsf{active}_{\mathsf{key}}, \mathsf{active}_{c_{ph}} \leftarrow$ true
- **for all** $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _) \in \mathcal{L}_{\mathsf{ct}}$ s.t. $T' = T \wedge \pi^{i'}_{T'}$.puvProtocol.$pk = \pi^i_T$.puvProtocol.$pk$ **do**
  - **if** $c' = c$ **then**
    - $\mathsf{active}_{\mathsf{key}} \leftarrow$ false
  - **if** $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    - $\mathsf{active}_{c_{ph}} \leftarrow$ false
    - break
- verifyPin $\leftarrow \mathsf{active}_{\mathsf{key}} \vee \mathsf{active}_{c_{ph}}$
- **if** verifyPin $=$ true **then**
  - $\mathsf{count}_{\mathsf{ActSendT}} \leftarrow \mathsf{count}_{\mathsf{ActSendT}} + 1$
  - pinHash $\leftarrow \mathsf{st}_T$.puvProtocol.decrypt(K, c_{ph})
  - **if** $(\mathcal{L}_{\mathsf{corrupt}}[\mathsf{st}_T.\mathsf{user}] = \text{true} \vee \mathsf{count}_{\mathsf{ActSendT}} \geq \mathsf{hy}) \wedge \mathcal{L}_{\mathsf{sampled}}[\mathsf{st}_T.\mathsf{user}] = \text{false}$ **then**
    - pin $\xleftarrow{\$} \mathcal{P}$
    - $\mathcal{L}_{\mathsf{valid}}[\mathsf{st}_T.\mathsf{user}] \leftarrow$ pin
    - $\mathcal{L}_{\mathsf{sampled}}[\mathsf{st}_T.\mathsf{user}] \leftarrow$ true
  - $\mathsf{pinHash}_T \leftarrow \mathsf{H_w}(\mathcal{L}_{\mathsf{valid}}[\mathsf{st}_T.\mathsf{user}])$
  - **if** $(\mathcal{L}_{\mathsf{corrupt}}[\mathsf{st}_T.\mathsf{user}] = \text{false} \wedge \mathsf{count}_{\mathsf{ActSendT}} \leq \mathsf{hy}) \vee$ pinHash $\neq \mathsf{pinHash}_T$ **then**
    - **if** $\mathsf{count}_{\mathsf{ActSendT}} = \mathsf{hy}$ **then**
      - **if** pinHash $= \mathsf{pinHash}_T$ **then**
        - bad $\leftarrow$ true
        - abort
    - st_T.puvProtocol.regenerate()
    - $\mathsf{st}_T.m \leftarrow \mathsf{st}_T.m - 1$
    - **if** $\mathsf{st}_T.m = 0$ **then**
      - authPowerUp-T(st_T)
      - **return** $(\perp, \text{true})$
    - **else**
      - **return** $(\perp, \text{false})$
- $\mathsf{st}_T.m \leftarrow 3, \mathsf{st}_T$.pinRetries $\leftarrow$ pinRetriesMax
- **for all** puvProtocol' $\in \mathsf{st}_T$.puvProtocolList **do**
  - st_T.puvProtocol'.resetPuvToken()
- π^i_T.bs $\leftarrow \pi^i_T$.puvProtocol.pt
- **if** $\mathsf{active}_{\mathsf{key}} = \text{true} \wedge (\mathsf{st}_T.\mathsf{puvProtocol}.pk, c) \neq \mathcal{L}_{\mathsf{setupKeys}}[T]$ **then**
  - $c_{pt} \xleftarrow{\$} \mathsf{st}_T$.puvProtocol.encrypt$(K, \pi^i_T.\mathsf{bs})$
- **else**
  - $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\mathsf{bs})}$
  - $c_{pt} \xleftarrow{\$} \mathsf{st}_T$.puvProtocol.encrypt$(K, \tilde{pt})$
  - $\mathcal{L}_{\mathsf{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, \pi^i_T.\mathsf{bs})$
- $\pi^i_T.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindDone
- π^i_T.canValidate $\leftarrow$ true
- π^i_T.sid $\leftarrow \pi^i_T$.sid $||$ puvProtocol $|| c || c_{ph} || c_{pt} ||$ false
- **return** (c_{pt}, false)

Figure E.28: Modified CTAP 2.1+ function obtainPinUvAuthToken-T - Game 7.i.

Advantage measure

Recall that during any game 7.i, no pin was ever sampled for any token before attempt i (except for when $\mathcal{A}$ corrupts a user, which makes it impossible to cause bad). Therefore, it is always true that:

$$\Pr_{7.i}[\mathsf{bad}] = \frac{1}{2^{h_{\mathscr{D}}}}, \forall i \in [1, q^{\mathsf{act}}_{\mathsf{Send}}] \tag{E.37}$$

Since games 7.i and 7.$(i-1)$ are identical until bad, through the difference lemma, we have:

$$\Pr_{7.i}[\mathsf{bad}] \geq |\ \Pr_{7.(i-1)} - \Pr_{7.i}\ |, \forall i \in [1, q^{\mathsf{act}}_{\mathsf{Send}}] \tag{E.38}$$

$$\sum_{i=1}^{q_{\mathsf{Send}}^{\mathsf{act}}} \mathsf{Pr}_{7.i}[\mathsf{bad}] \geq \sum_{i=1}^{q_{\mathsf{Send}}^{\mathsf{act}}} \mid \mathsf{Pr}_{7.(i-1)} - \mathsf{Pr}_{7.i} \mid \tag{E.39}$$

$$\sum_{i=1}^{q_{\mathsf{Send}}^{\mathsf{act}}} \mathsf{Pr}_{7.i}[\mathsf{bad}] \geq \mid \mathsf{Pr}_{7.0} - \mathsf{Pr}_{7.q_{\mathsf{Send}}^{\mathsf{act}}} \mid \tag{E.40}$$

Since bad has the same probability of $1/2^{h_{\mathscr{D}}}$ for every $i \in [1, q_{\mathsf{Send}}^{\mathsf{act}}]$, we can write:

$$q_{\mathsf{Send}}^{\mathsf{act}}/2^{h_{\mathscr{D}}} \geq \mid \mathsf{Pr}_{7.0} - \mathsf{Pr}_{7.q_{\mathsf{Send}}^{\mathsf{act}}} \mid \tag{E.41}$$

Since Game 7.0 corresponds to Game 7 and Game $7.q_{\mathsf{Send}}^{\mathsf{act}}$ corresponds to Game 8, we have:

$$q_{\mathsf{Send}}^{\mathsf{act}}/2^{h_{\mathscr{D}}} \geq \mid \mathsf{Pr}_{7} - \mathsf{Pr}_{8} \mid \tag{E.42}$$

Game 9

In this game, $\mathcal{A}$ loses by causing a bad event if and only if a collision between two pinUvAuthTokens that are sent (encrypted) to $\mathcal{A}$ occurs. More specifically, we are only considering pinUvAuthTokens that i. are generated by an actively attacked token session π_T^i that gives the pinUvAuthToken directly to $\mathcal{A}$, which happens only in Send-Bind-T queries, or ii. are generated by an honest execution of Bind through an Execute query, in which case the pinUvAuthToken being considered is the random value $\tilde{pt}$ that is sent in ciphertext $\tilde{c}_{pt}$ and not the pinUvAuthToken pt set as the binding state (recall that this $\tilde{c}_{pt}$ will be part of the session identifier).

In either case, each query to Execute or Send-Bind-T will always, at most, result in one pinUvAuthToken that is sent (encrypted) to $\mathcal{A}$.

Advantage measure

Since each query to Execute or Send-Bind-T creates and sends to $\mathcal{A}$, at most, one pinUvAuthToken, the maximum amount of such pinUvAuthTokens being created in the experiment is $q_{\mathsf{E}} + q_{\mathsf{Send}}$.

Since the pinUvAuthToken is a random bit string of length 2λ (for the PIN/UV Auth Protocol being analyzed), the maximum number of pinUvAuthTokens that can possibly exist is $2^{2\lambda}$.

Game9($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
$\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
$\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pt}} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF-t

Figure E.29: Game 9. The flag bad in red is only used in this game, to signal when a pinUvAuthToken collision happens.

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then**
  return $(\perp,$ false)
$K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
if $K = \perp$ **then**
  return $(\perp,$ false)
st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
$(c', c'_{ph}, _, _) \leftarrow \mathcal{L}_{\text{ct}}[_, \pi^i_T]$
active$_{\text{key}} \leftarrow c' \neq c$
active$_{c_{ph}} \leftarrow c'_{ph} \neq c_{ph}$
for all $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T$ **do**
  if $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    active$_{c_{ph}} \leftarrow$ false
    active$_{\text{key}} \leftarrow$ false
verifyPin $\leftarrow$ active$_{\text{key}} \vee$ active$_{c_{ph}}$
if verifyPin $=$ true **then**
  pinHash $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_{ph})
  if $\mathcal{L}_{\text{corrupt}}$[st$_T$.user] $=$ true $\wedge$ $\mathcal{L}_{\text{sampled}}$[st$_T$.user] $=$ false **then**
    pin $\xleftarrow{\$} \mathcal{P}$
    $\mathcal{L}_{\text{valid}}$[st$_T$.user] $\leftarrow$ pin
    $\mathcal{L}_{\text{sampled}}$[st$_T$.user] $\leftarrow$ true
  pinHash$_T \leftarrow \text{H}_w(\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}])$
  if $\mathcal{L}_{\text{corrupt}}$[st$_T$.user] $=$ false $\vee$ pinHash $\neq$ pinHash$_T$ **then**
    st$_T$.puvProtocol.regenerate()
    st$_T.m \leftarrow$ st$_T.m - 1$
    if st$_T.m = 0$ **then**
      authPowerUp-T(st$_T$)
      return $(\perp,$ true)
    else
      return $(\perp,$ false)
st$_T.m \leftarrow 3$, st$_T$.pinRetries $\leftarrow$ pinRetriesMax
for all puvProtocol' $\in$ st$_T$.puvProtocolList **do**
  st$_T$.puvProtocol'.resetPuvToken()
π^i_T.bs $\leftarrow \pi^i_T$.puvProtocol.pt
if active$_{\text{key}}$ $=$ true $\wedge$ (st$_T$.puvProtocol.pk, c) $\neq$ $\mathcal{L}_{\text{setupKeys}}[T]$ **then**
  if π^i_T.bs $\in \mathcal{L}_{\text{pt}}$ **then**
    bad $\leftarrow$ true
    abort
  $\mathcal{L}_{\text{pt}} \leftarrow \mathcal{L}_{\text{pt}} \cup \{\pi^i_T.\text{bs}\}$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt(K, π^i_T.bs)
else
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\text{bs})}$
  if $\tilde{pt} \in \mathcal{L}_{\text{pt}}$ **then**
    bad $\leftarrow$ true
    abort
  $\mathcal{L}_{\text{pt}} \leftarrow \mathcal{L}_{\text{pt}} \cup \{\tilde{pt}\}$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt($K, \tilde{pt}$)
  $\mathcal{L}_{\text{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, c_{pt}, \pi^i_T.\text{bs})$
π^i_T.st$_{\text{exe}} \leftarrow$ bindDone
π^i_T.canValidate $\leftarrow$ true
π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || c || c_{ph} || c_{pt} || false
return $(c_{pt},$ false)

Figure E.30: Modified CTAP 2.1+ function obtainPinUvAuthToken-T - Game 9.

Considering the probability of a collision between two pinUvAuthTokens as $1/2^{2\lambda}$, and $(q_{\mathsf{E}} + q_{\mathsf{Send}})^2/2$ as an upper bound for the amount of pinUvAuthToken pairs, we have:

$$\mathsf{Pr}_9[\mathsf{bad}] \leq \frac{(q_{\mathsf{E}} + q_{\mathsf{Send}})^2}{2^{2\lambda+1}} \tag{E.43}$$

Since games 8 and 9 are identical until bad, by the difference lemma, we have:

$$\mathsf{Pr}_9[\mathsf{bad}] \geq |\ \mathsf{Pr}_8 - \mathsf{Pr}_9\ | \tag{E.44}$$

And, therefore:

$$\frac{(q_{\mathsf{E}} + q_{\mathsf{Send}})^2}{2^{2\lambda+1}} \geq |\ \mathsf{Pr}_8 - \mathsf{Pr}_9\ | \tag{E.45}$$

Game 10

In this game, $\mathcal{A}$ loses if, at any point in the game, it causes a bad event, which happens when it queries random oracle $\mathcal{H}_3$ with *any* pinUvAuthToken generated throughout the experiment and set as the binding state of some token or client session, but for which the adversary has no information.

We maintain two new structures $\mathcal{L}_{\mathsf{bs}}$ and $\mathcal{L}_{\mathcal{H}_3}$ to keep track of all binding states about which $\mathcal{A}$ has no information and of all of $\mathcal{A}$'s queries to $\mathcal{H}_3$, respectively.

Advantage measure

At this point in the game, $\mathcal{A}$ has no information about any binding state that results from Execute queries or Send-Bind-T queries where it behaves passively, because in these circumstances the c_{pt} given to $\mathcal{A}$ no longer contains any information about π_T^i.bs. Therefore, we can upper bound the amount of binding states that can cause the bad event as $(q_{\mathsf{E}} + q_{\mathsf{Send}})$.

Let $q_{\mathcal{H}_3}$ be the maximum amount of queries $\mathcal{A}$ performs during game 10. Given that the maximum amount of pinUvAuthTokens that can possibly exist is $2^{2\lambda}$, we know that the probability that $\mathcal{A}$ queries $\mathcal{H}_3$ with one of the binding states about which it has no information is upper bounded by $q_{\mathcal{H}_3}\ (q_{\mathsf{E}} + q_{\mathsf{Send}})/2^{2\lambda}$. Therefore, we have have that:

$$\mathsf{Pr}_{10}[\mathsf{bad}] \leq\ q_{\mathcal{H}_3}\ (q_{\mathsf{E}} + q_{\mathsf{Send}})/2^{2\lambda} \tag{E.46}$$

Game10($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
$\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
$\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF-t $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF-t

Adv$\mathcal{H}_3$ (u_1):
$v \leftarrow \mathcal{H}_3(u_1)$
if $u_1 \in \mathcal{L}_{\text{bs}}$ **then**
  bad $\leftarrow$ true
  abort
$\mathcal{L}_{\mathcal{H}_3}[u_1] \leftarrow v$
return v

obtainPinUvAuthToken-T(π^i_T, puvProtocol, c, c_{ph}):
if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then**
  return ($\perp$, false)
$K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
if $K = \perp$ **then**
  return ($\perp$, false)
st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
$(c', c'_{ph}, _, _) \leftarrow \mathcal{L}_{\text{ct}}[_, \pi^i_T]$
active$_{\text{key}} \leftarrow c' \neq c$
active$_{c_{ph}} \leftarrow c'_{ph} \neq c_{ph}$
for all $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _, _) \in \mathcal{L}_{\text{ct}}$ s.t. $T' = T$ **do**
  if $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    active$_{c_{ph}} \leftarrow$ false
    active$_{\text{key}} \leftarrow$ false
verifyPin $\leftarrow$ active$_{\text{key}} \vee$ active$_{c_{ph}}$
if verifyPin $=$ true **then**
  pinHash $\leftarrow$ st$_T$.puvProtocol.decrypt(K, c_{ph})
  if $\mathcal{L}_{\text{corrupt}}$[st$_T$.user] $=$ true $\wedge$ $\mathcal{L}_{\text{sampled}}$[st$_T$.user] $=$ false **then**
    pin $\xleftarrow{\$} \mathcal{P}$
    $\mathcal{L}_{\text{valid}}$[st$_T$.user] $\leftarrow$ pin
    $\mathcal{L}_{\text{sampled}}$[st$_T$.user] $\leftarrow$ true
  pinHash$_T \leftarrow \text{H}_{\text{w}}(\mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}])$
  if $\mathcal{L}_{\text{corrupt}}$[st$_T$.user] $=$ false $\vee$ pinHash $\neq$ pinHash$_T$ **then**
    st$_T$.puvProtocol.regenerate()
    st$_T$.$m \leftarrow$ st$_T$.$m - 1$
    if st$_T$.$m = 0$ **then**
      authPowerUp-T(st$_T$)
      return ($\perp$, true)
    else
      return ($\perp$, false)
st$_T$.$m \leftarrow 3$, st$_T$.pinRetries $\leftarrow$ pinRetriesMax
for all puvProtocol' $\in$ st$_T$.puvProtocolList **do**
  st$_T$.puvProtocol'.resetPuvToken()
π^i_T.bs $\leftarrow \pi^i_T$.puvProtocol.pt
if active$_{\text{key}} =$ true $\wedge$ (st$_T$.puvProtocol.pk, c) $\neq$ $\mathcal{L}_{\text{setupKeys}}[T]$ **then**
  if π^i_T.bs $\in \mathcal{L}_{\text{pt}}$ **then**
    abort
  $\mathcal{L}_{\text{pt}} \leftarrow \mathcal{L}_{\text{pt}} \cup \{\pi^i_T.\text{bs}\}$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt(K, π^i_T.bs)
else
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\text{bs})}$
  if $\tilde{pt} \in \mathcal{L}_{\text{pt}}$ **then**
    abort
  $\mathcal{L}_{\text{pt}} \leftarrow \mathcal{L}_{\text{pt}} \cup \{\tilde{pt}\}$
  if $\mathcal{L}_{\mathcal{H}_3}[\pi^i_T.\text{bs}] \neq \perp$ **then**
    bad $\leftarrow$ true
    abort
  $\mathcal{L}_{\text{bs}} \leftarrow \mathcal{L}_{\text{bs}} \cup \{\pi^i_T.\text{bs}\}$
  $c_{pt} \xleftarrow{\$}$ st$_T$.puvProtocol.encrypt($K, \tilde{pt}$)
  $\mathcal{L}_{\text{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, c_{pt}, \pi^i_T.\text{bs})$
π^i_T.st$_{\text{exe}} \leftarrow$ bindDone
π^i_T.canValidate $\leftarrow$ true
π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || c || c_{ph} || c_{pt} || false
return (c_{pt}, false)

Figure E.31: Game 10. The flag bad in red is only used in this game. Adv$\mathcal{H}_3$ is a wrapper that $\mathcal{A}$ can use to call the random oracle, so that we can distinguish calls to $\mathcal{H}_3$ made specifically by $\mathcal{A}$.

Since games 9 and 10 are identical until bad, by the difference lemma, we have:

$$\text{Pr}_{10}[\text{bad}] \geq |\ \text{Pr}_9 - \text{Pr}_{10}\ | \tag{E.47}$$

And, therefore:

$$q_{\mathcal{H}_3}\ (q_{\text{E}} + q_{\text{Send}})/2^{2\lambda} \geq |\ \text{Pr}_9 - \text{Pr}_{10}\ | \tag{E.48}$$

Game 11

In this game, when $\mathcal{A}$ delivers a message M and a tag t to a token session π_T^i that finished Bind passively (π_T^i did not finish Bind through an active Send-Bind-T query) by querying oracle Validate-T, the challenger always rejects, provided M and t were not output by any of π_T^i's partner (if it has one), the partner was not compromised and the pin associated with π_T^i was not corrupted. In sum, given M and t that would result in $\mathcal{A}$ winning the experiment, the challenger rejects instead, making it impossible for $\mathcal{A}$ to win by forging a message tag and delivering it to any token session. We flag this event by creating a bad event bad_{11}.

Game11($\mathcal{A}$):
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
- $\text{bad}_{11} \leftarrow$ false
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **return** win-SUF-t

Token-Win-SUF-t (T, i, M, t, d):
- **if** $d \neq$ accepted **then return** 1
- **if** $\exists (C_1, j_1), (C_2, j_2)\ s.t.\ (C_1, j_1) \neq (C_2, j_2)$ **and** $\pi_{C_1}^{j_1}.\text{st}_{\text{exe}} = \pi_{C_2}^{j_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi_{C_1}^{j_1}.\text{sid} = \pi_{C_2}^{j_2}.\text{sid}$ **then return** 1
- **if** $\exists (T_1, i_1), (T_2, i_2)\ s.t.\ (T_1, i_1) \neq (T_2, i_2)$ **and** $\pi_{T_1}^{i_1}.\text{st}_{\text{exe}} = \pi_{T_2}^{i_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi_{T_1}^{i_1}.\text{sid} = \pi_{T_2}^{i_2}.\text{sid}$ **then return** 1
- $(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
- **if** $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
  - **if** $(C, j) = (\perp, \perp)$ **or** $\pi_C^j.\text{compromised} = \text{false}$ **then**
    - **if** $\pi_T^i.\text{pinCorr} = \text{false}$ **then**
      - **return** $1 \wedge \text{bad}_{11} \neq \text{true}$
- **return** 0

Figure E.32: Game 11 and modified function Token-Win-SUF-t.

Auth-C (C, j, M):
- **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi_C^j.\text{isValid} = \text{false}$ **then**
  - **return** $\perp$
- $(M, t) \xleftarrow{\$}$ auth-C(π_C^j, M)
- $\mathcal{L}_{\text{authC}} \leftarrow \mathcal{L}_{\text{authC}} \cup \{(C, j, M, t)\}$
- **return** (M, t)

Validate-T (T, i, M, t, d):
- **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi_T^i.\text{isValid} = \text{false}$ **then**
  - **return** $\perp$
- $(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
- status $\leftarrow$ validate-T(π_T^i, M, t, d)
- **if** $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
  - **if** $(C, j) = (\perp, \perp)$ **or** $\pi_C^j.\text{compromised} = \text{false}$ **then**
    - **if** $\pi_T^i.\text{pinCorr} = \text{false}$ **then**
      - **if** status $=$ accepted **then**
        - $\text{bad}_{11} \leftarrow$ true
        - win-SUF-t $\leftarrow$ Token-Win-SUF-t(T, i, M, t, d)
      - **return** rejected
- **if** status $=$ accepted **then**
  - win-SUF-t $\leftarrow$ Token-Win-SUF-t(T, i, M, t, d)
- **return** status

Figure E.33: Modified Auth-C and Validate-T oracles - Game 11.

Notice that, after Game 9, every token session that finishes Bind has *at most* one partner. Indeed, for two sessions π_C^j and π_T^i to be partnered, they must have finished binding and

have matching sid values ($\pi_C^j.\text{sid} = \pi_T^i.\text{sid}$). The sid, after binding is completed, is mainly a concatenation of the Bind session's client and token ECDH public keys c and pk, the pinHash ciphertext c_{ph} and the pinUvAuthToken ciphertext c_{pt}. Since we have eliminated collisions of client ECDH public keys c in Game 3, we know that every client session sid is unique. For token-side sid values on the other hand, it is still possible that two or more token sessions of the same token T had the same c delivered to them by $\mathcal{A}$ and the same token ECDH public key for multiple Bind sessions (since the same token can reuse the same public key), which means avoiding collisions between token public keys is not enought to guarantee a unique sid for each token session.

However, in Game 9, we have also eliminated collisions between every pinUvAuthToken that is encrypted into a ciphertext c_{pt} and sent to $\mathcal{A}$ in Execute or Send-Bind-T queries, which means that c_{pt} can never repeat for the same c and pk, as long as the underlying symmetric encryption scheme SKE is correct. Therefore, every token session π_T^i also has a unique sid.

This is an important observation, as it gives a guarantee that if a token session π_T^i has a partner π_C^j, then π_C^j is its *unique* partner.

After Game 8, it is also always the case that every token session where bad_{11} can happen must have finished Bind passively, through an Execute query or a passive Send-Bind-T query. Indeed, after Game 8, $\mathcal{A}$ can no longer actively attack a token session π_T^i via the Send-Bind-T oracle without corrupting the pin. However, if $\mathcal{A}$ does corrupt the pin, then she can never cause bad_{11}, i.e., can never win the game via a valid forgery against any session of token T.

Therefore, *only* passive executions of Bind can create token sessions where bad_{11} can happen. This also gives us an additional guarantee that such sessions always have a binding state sampled uniformly at random, since these binding states were never checked for collisions in Game 9 because they are never transmitted to $\mathcal{A}$. [1]

Reduction to the SUF-CMA security of MAC

To bound $\mathcal{A}$'s advantage against Game 11, we consider an adversary $\mathcal{B}_{11}$ simulating Game 11 and fix a value n_{TS} for the maximum amount of token sessions that finished Bind passively. Then, $\mathcal{B}_{11}$ guesses at random in which token session bad_{11} will happen first. We refer to that target session as $\pi_{T,\text{target}}^i$.

[1] This guarantee is especially significant when considering the proof for the original CTAP, since it allows to bound $\mathcal{A}$'s advantage by reducing to the SUF-CMA security of MAC.

If $\mathcal{A}$ ever queries Compromise on the client session that is partnered with $\pi^i_{T,\text{target}}$, then the simulation aborts, since it is now impossible to cause bad_{11} on the target session $\pi^i_{T,\text{target}}$ (recall that if $\pi^j_C.\text{compromised} = \text{true}$, then $\mathcal{A}$ can never win via a valid forgery against its partner token session).

Whenever $\mathcal{A}$ queries the Auth-C oracle with a message M such that π^j_C is the partner of the target session $\pi^i_{T,\text{target}}$, $\mathcal{B}_{11}$ calls the $\mathcal{O}_{\text{Auth}}$ oracle to obtain a challenge tag t for M, and returns (M, t) to $\mathcal{A}$.

Whenever $\mathcal{A}$ queries the Validate-T oracle with message M and tag t such that π^i_T is the target session $\pi^i_{T,\text{target}}$, $\mathcal{B}_{11}$ immediately returns M and t to its challenger and ends the simulation. If π^i_T is not the target session, $\mathcal{B}_{11}$ continues simulating Game 11 until either $\mathcal{A}$ queries Validate-T for session $\pi^i_{T,\text{target}}$ or bad_{11} happens.

It is not hard to see then, that if bad_{11} ever happens in the target session $\pi^i_{T,\text{target}}$, $\mathcal{B}_{11}$ always wins the SUF-CMA game by returning the (M, t) given to $\pi^i_{T,\text{target}}$ by $\mathcal{A}$ in a Validate-T query.

$\mathcal{B}_{11}(\mathcal{A}, \mathcal{O}_{\text{Auth}}, \mathcal{O}_{\text{Ver}})$:
  $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
  $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
  $\mathcal{L}_{\text{pt}} \leftarrow \emptyset$
  $\text{bad}_{11} \leftarrow \text{false}$
  $\text{guess} \xleftarrow{\$} \{0, \ldots, n_{\text{TS}} - 1\}$
  $\mathcal{L}_{\text{TS}} \leftarrow \emptyset, (M_{\text{chal}}, t_{\text{chal}}) \leftarrow (\perp, \perp)$
  win-SUF-t $\leftarrow 0$
  $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
  return $(M_{\text{chal}}, t_{\text{chal}})$

Figure E.34: Adversary $\mathcal{B}_{11}$'s simulation of game 11.

Advantage measure

The advantage of $\mathcal{B}_{11}$ against the SUF-CMA security of MAC can be calculated as:

$$\text{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_{11}) \geq \frac{\Pr_{11}[\text{bad}_{11}]}{n_{\text{TS}}} \tag{E.49}$$

$$n_{\text{TS}}\ \text{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_{11}) \geq \Pr_{11}[\text{bad}_{11}] \tag{E.50}$$

Since games 10 and 11 are identical until bad_{11}, by the difference lemma, we have:

Auth-C (C, j, M):

if $\pi^j_C = \perp$ **or** $\pi^j_C.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **or** $\pi^j_C.\mathsf{isValid} = \mathsf{false}$ **then**
  return $\perp$
$\pi^i_{T,\mathsf{target}} \leftarrow \mathcal{L}_{\mathsf{TS}}[\mathsf{guess} - 1]$
if $\pi^i_{T,\mathsf{target}} \neq \perp \wedge (C, j) = \mathsf{tokenBindPartner}(T, i)$ **then**
  $(M, t) \leftarrow \mathcal{O}_{\mathsf{Auth}}(M)$
else
  $(M, t) \xleftarrow{\$} \text{auth-C}(\pi^j_C, M)$
$\mathcal{L}_{\mathsf{authC}} \leftarrow \mathcal{L}_{\mathsf{authC}} \cup \{(C, j, M, t)\}$
return (M, t)

Compromise (C, j):

if $\pi^j_C = \perp$ **or** $\pi^j_C.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **then**
  return $\perp$
$\pi^i_{T,\mathsf{target}} \leftarrow \mathcal{L}_{\mathsf{TS}}[\mathsf{guess} - 1]$
if $\pi^i_{T,\mathsf{target}} \neq \perp \wedge (C, j) = \mathsf{tokenBindPartner}(T, i)$ **then**
  abort
$\pi^j_C.\mathsf{compromised} = \mathsf{True}$
return $\pi^j_C.\mathsf{bs}$

Validate-T (T, i, M, t, d):

if $\pi^i_T = \perp$ **or** $\pi^i_T.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **or** $\pi^i_T.\mathsf{isValid} = \mathsf{false}$ **then**
  return $\perp$
$(C, j) \leftarrow \mathsf{tokenBindPartner}(T, i)$
if $\pi^i_T = \mathcal{L}_{\mathsf{TS}}[\mathsf{guess} - 1]$ **then**
  $(M_{\mathsf{chal}}, t_{\mathsf{chal}}) \leftarrow (M, t)$
  return
else
  $\mathsf{status} \leftarrow \text{validate-T}(\pi^i_T, M, t, d)$
if $(C, j, M, t) \notin \mathcal{L}_{\mathsf{authC}}$ **then**
  if $(C, j) = (\perp, \perp)$ **or** $\pi^j_C.\mathsf{compromised} = \mathsf{false}$ **then**
    if $\pi^i_T.\mathsf{pinCorr} = \mathsf{false}$ **then**
      if $\mathsf{status} = \mathsf{accepted}$ **then**
        $\mathsf{bad}_{11} \leftarrow \mathsf{true}$
        $(M_{\mathsf{chal}}, t_{\mathsf{chal}}) \leftarrow (M, t)$
        return
      return rejected
if $\mathsf{status} = \mathsf{accepted}$ **then**
  $\text{win-SUF-t} \leftarrow \text{Token-Win-SUF-t}(T, i, M, t, d)$
return status

Figure E.35: Modified Auth-C, Validate-T and Compromise oracles simulated by $\mathcal{B}_{11}$.

obtainPinUvAuthToken-T$(\pi^i_T, \mathsf{puvProtocol}, c, c_{ph})$:

if $\mathsf{puvProtocol} \notin \mathsf{st}_T.\mathsf{puvProtocolList} \vee \mathsf{st}_T.\mathsf{pinRetries} = 0$ **then**
  return $(\perp, \mathsf{false})$
$K \leftarrow \mathsf{st}_T.\mathsf{puvProtocol.decapsulate}(c)$
if $K = \perp$ **then**
  return $(\perp, \mathsf{false})$
$\mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetries} - 1$
$(c', c'_{ph}, _, _) \leftarrow \mathcal{L}_{\mathsf{ct}}[_, \pi^i_T]$
$\mathsf{active}_{\mathsf{key}} \leftarrow c' \neq c$
$\mathsf{active}_{c_{ph}} \leftarrow c'_{ph} \neq c_{ph}$
for all $(\pi^{j'}_{C'}, \pi^{i'}_{T'}, c', c'_{ph}, _, _) \in \mathcal{L}_{\mathsf{ct}}$ s.t. $T' = T$ **do**
  if $c'_{ph} = c_{ph} \wedge c' = c$ **then**
    $\mathsf{active}_{c_{ph}} \leftarrow \mathsf{false}$
    $\mathsf{active}_{\mathsf{key}} \leftarrow \mathsf{false}$
$\mathsf{verifyPin} \leftarrow \mathsf{active}_{\mathsf{key}} \vee \mathsf{active}_{c_{ph}}$
if $\mathsf{verifyPin} = \mathsf{true}$ **then**
  $\mathsf{pinHash} \leftarrow \mathsf{st}_T.\mathsf{puvProtocol.decrypt}(K, c_{ph})$
  if $\mathcal{L}_{\mathsf{corrupt}}[\mathsf{st}_T.\mathsf{user}] = \mathsf{true} \wedge \mathcal{L}_{\mathsf{sampled}}[\mathsf{st}_T.\mathsf{user}] = \mathsf{false}$ **then**
    $\mathsf{pin} \xleftarrow{\$} \mathcal{P}$
    $\mathcal{L}_{\mathsf{valid}}[\mathsf{st}_T.\mathsf{user}] \leftarrow \mathsf{pin}$
    $\mathcal{L}_{\mathsf{sampled}}[\mathsf{st}_T.\mathsf{user}] \leftarrow \mathsf{true}$
  $\mathsf{pinHash}_T \leftarrow \mathsf{H}_w(\mathcal{L}_{\mathsf{valid}}[\mathsf{st}_T.\mathsf{user}])$
  if $\mathcal{L}_{\mathsf{corrupt}}[\mathsf{st}_T.\mathsf{user}] = \mathsf{false} \vee \mathsf{pinHash} \neq \mathsf{pinHash}_T$ **then**
    $\mathsf{st}_T.\mathsf{puvProtocol.regenerate}()$
    $\mathsf{st}_T.m \leftarrow \mathsf{st}_T.m - 1$
    if $\mathsf{st}_T.m = 0$ **then**
      $\text{authPowerUp-T}(\mathsf{st}_T)$
      return $(\perp, \mathsf{true})$
    else
      return $(\perp, \mathsf{false})$
$\mathsf{st}_T.m \leftarrow 3, \mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetriesMax}$
for all $\mathsf{puvProtocol'} \in \mathsf{st}_T.\mathsf{puvProtocolList}$ **do**
  $\mathsf{st}_T.\mathsf{puvProtocol'.resetPuvToken}()$
$\pi^i_T.\mathsf{bs} \leftarrow \pi^i_T.\mathsf{puvProtocol}.pt$
if $\mathsf{active}_{\mathsf{key}} = \mathsf{true} \wedge (\mathsf{st}_T.\mathsf{puvProtocol}.pk, c) \neq \mathcal{L}_{\mathsf{setupKeys}}[T]$ **then**
  if $\pi^i_T.\mathsf{bs} \in \mathcal{L}_{\mathsf{pt}}$ **then**
    abort
  $\mathcal{L}_{\mathsf{pt}} \leftarrow \mathcal{L}_{\mathsf{pt}} \cup \{\pi^i_T.\mathsf{bs}\}$
  $c_{pt} \xleftarrow{\$} \mathsf{st}_T.\mathsf{puvProtocol.encrypt}(K, \pi^i_T.\mathsf{bs})$
else
  $\tilde{pt} \xleftarrow{\$} \{0,1\}^{len(\pi^i_T.\mathsf{bs})}$
  if $\tilde{pt} \in \mathcal{L}_{\mathsf{pt}}$ **then**
    abort
  $\mathcal{L}_{\mathsf{pt}} \leftarrow \mathcal{L}_{\mathsf{pt}} \cup \{\tilde{pt}\}$
  if $\pi^i_T.\mathsf{bs} \in \mathcal{L}_{\mathcal{H}_3}$ **then**
    abort
  $\mathcal{L}_{\mathsf{bs}} \leftarrow \mathcal{L}_{\mathsf{bs}} \cup \{\pi^i_T.\mathsf{bs}\}$
  $c_{pt} \xleftarrow{\$} \mathsf{st}_T.\mathsf{puvProtocol.encrypt}(K, \tilde{pt})$
  $\mathcal{L}_{\mathsf{ct}}[_, \pi^i_T] \leftarrow (c', c'_{ph}, c_{pt}, \pi^i_T.\mathsf{bs})$
  $\mathcal{L}_{\mathsf{TS}} \leftarrow \mathcal{L}_{\mathsf{TS}} \cup \{\pi^i_T\}$
$\pi^i_T.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{bindDone}$
$\pi^i_T.\mathsf{canValidate} \leftarrow \mathsf{true}$
$\pi^i_T.\mathsf{sid} \leftarrow \pi^i_T.\mathsf{sid} \,||\, \mathsf{puvProtocol} \,||\, c \,||\, c_{ph} \,||\, c_{pt} \,||\, \mathsf{false}$
return (c_{pt}, false)

Figure E.36: obtainPinUvAuthToken-T function simulated by $\mathcal{B}_{11}$.

$$\mathsf{Pr}_{11}[\mathsf{bad}_{11}] \geq \mid \mathsf{Pr}_{10} - \mathsf{Pr}_{11} \mid \tag{E.51}$$

Thus, we have:

$$n_{\mathsf{TS}} \ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_{11}) \geq \mid \mathsf{Pr}_{10} - \mathsf{Pr}_{11} \mid \tag{E.52}$$

Since each token session can finish Bind passively only in Execute and Send-Bind-T queries, we know that $n_{\mathsf{TS}} \leq q_{\mathsf{E}} + q_{\mathsf{Send}}$. Therefore:

$$(q_{\mathsf{E}} + q_{\mathsf{Send}}) \ \mathsf{Adv}^{\text{SUF-CMA}}_{\mathsf{MAC}}(\mathcal{B}_{11}) \geq \mid \mathsf{Pr}_{10} - \mathsf{Pr}_{11} \mid \tag{E.53}$$

Game 12

In this game, when $\mathcal{A}$ delivers a message M and a tag t to a client session π^j_C by querying oracle Validate-C, the challenger always rejects, provided M and t were not output by π^j_C's partner, π^j_C was not compromised and the pin associated with π^j_C's partner was not corrupted. In sum, given M and t that would result in $\mathcal{A}$ winning the experiment, the challenger rejects instead, making it impossible for $\mathcal{A}$ to win by forging a message tag and delivering it to any client session. We flag this event by creating a bad event bad_{12}.

Game12($\mathcal{A}$):
- $\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{valid}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pin}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{ct}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{sampled}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{pt}} \leftarrow \emptyset$
- $\mathcal{L}_{\mathsf{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
- $\mathsf{bad}_{11} \leftarrow$ false
- $\mathsf{bad}_{12} \leftarrow$ false
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
- **return** win-SUF-t

Client-Win-SUF-t (C, j, M, t):
- **if** $\exists (C_1, j_1), (C_2, j_2)\ s.t.\ (C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\mathsf{st}_{\mathsf{exe}} = \pi^{j_2}_{C_2}.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$ **and** $\pi^{j_1}_{C_1}.\mathsf{sid} = \pi^{j_2}_{C_2}.\mathsf{sid}$ **then return** 1
- **if** $\exists (T_1, i_1), (T_2, i_2)\ s.t.\ (T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\mathsf{st}_{\mathsf{exe}} = \pi^{i_2}_{T_2}.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindDone}$ **and** $\pi^{i_1}_{T_1}.\mathsf{sid} = \pi^{i_2}_{T_2}.\mathsf{sid}$ **then return** 1
- $(T, i) \leftarrow \mathsf{clientBindPartner}(C, j)$
- **if** $(T, i, M, t) \notin \mathcal{L}_{\mathsf{authT}}$ **then**
  - **if** $\pi^j_C.\mathsf{compromised} = \mathsf{false}$ **then**
    - **if** $\pi^i_T.\mathsf{pinCorr} = \mathsf{false}$ **then**
      - **return** $1 \wedge \mathsf{bad}_{12} \neq \mathsf{true}$
- **return** 0

Figure E.37: Game 12 and function Client-Win-SUF-t modifications.

Reduction to the SUF-CMA security of MAC

As was the case in Game 11, we bound the probability of $\mathcal{A}$ winning against Game 12 by reducing to the SUF-CMA security of the underlying MAC.

```
Validate-C (C, j, M, t):
    if π^j_C = ⊥ or π^j_C.st_exe ≠ bindDone or π^j_C.isValid = false then
        return ⊥
    (T, i) ← clientBindPartner(C, j)
    status ← validate-C(π^j_C, M, t)
    if (T, i, M, t) ∉ L_authT then
        if π^j_C.compromised = false then
            if π^i_T.pinCorr = false then
                if status = accepted then
                    bad_12 ← true
                    win-SUF-t ← Client-Win-SUF-t(C, j, M, t)
                return rejected
    if status = accepted then
        win-SUF-t ← Client-Win-SUF-t(C, j, M, t)
    return status

Auth-T (T, i, M):
    if π^i_T = ⊥ or π^i_T.st_exe ≠ bindDone or π^i_T.isValid = false then
        return ⊥
    (M, t) ←$ auth-T(π^i_T, M)
    L_authT ← L_authT ∪ {(T, i, M, t)}
    return (M, t)
```

Figure E.38: Modified Auth-T and Validate-C oracles - Game 12.

We consider an adversary $\mathcal{B}_{12}$ against the SUF-CMA game that simulates Game 12 and fix a value n_{CS} for the maximum amount of client sessions that finish Bind with a token session. Then, $\mathcal{B}_{12}$ guesses at random in which client session bad_{12} will happen first. We refer to that session as $\pi^j_{C,\mathsf{target}}$.

If $\mathcal{A}$ ever queries Compromise on $\pi^j_{C,\mathsf{target}}$, then the simulation aborts, since it is now impossible to cause bad_{12} on the target session $\pi^j_{C,\mathsf{target}}$ (recall that if $\pi^j_C.\mathsf{compromised} = \mathsf{true}$, then $\mathcal{A}$ can never win via a valid forgery against π^j_C).

Whenever $\mathcal{A}$ queries the Auth-T oracle with a message M such that π^i_T is the partner of the target session $\pi^j_{C,\mathsf{target}}$, $\mathcal{B}_{12}$ calls the $\mathcal{O}_{\mathsf{Auth}}$ oracle to obtain a challenge tag t for M, and returns (M, t) to $\mathcal{A}$.

Whenever $\mathcal{A}$ queries the Validate-C oracle with message M and tag t such that π^j_C is the target session $\pi^j_{C,\mathsf{target}}$, $\mathcal{B}_{12}$ immediately returns M and t to its challenger and ends the simulation. If π^j_C is not the target session, $\mathcal{B}_{12}$ continues simulating Game 12 until either $\mathcal{A}$ queries Validate-C for session $\pi^j_{C,\mathsf{target}}$ or bad_{12} happens.

It is not hard to see then, that if bad_{12} ever happens in the target session $\pi^j_{C,\mathsf{target}}$, $\mathcal{B}_{12}$ always wins the SUF-CMA game by returning the (M, t) given to $\pi^j_{C,\mathsf{target}}$ by $\mathcal{A}$ in a Validate-C query.

Advantage measure

The advantage of $\mathcal{B}_{12}$ against the SUF-CMA security of MAC is upper bounded by:

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_{12}) \geq \frac{\Pr_{12}[\mathsf{bad}_{12}]}{n_{\mathsf{CS}}} \tag{E.54}$$

$\mathcal{B}_{12}(\mathcal{A}, \mathcal{O}_{\text{Auth}}, \mathcal{O}_{\text{Ver}})$:
- $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset, \mathcal{L}_{\text{valid}} \leftarrow \emptyset, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
- $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pk}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pin}} \leftarrow \emptyset, \mathcal{L}_{\text{setupKeys}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{ct}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{hash}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{sampled}} \leftarrow \emptyset$
- $\mathcal{L}_{\text{pt}} \leftarrow \emptyset$
- $\text{bad}_{11} \leftarrow$ false
- $\text{bad}_{12} \leftarrow$ false
- guess $\xleftarrow{\$} \{0, ..., n_{\text{CS}} - 1\}$
- $\mathcal{L}_{\text{CS}} \leftarrow \emptyset, (M_{\text{chal}}, t_{\text{chal}}) \leftarrow (\perp, \perp)$
- win-SUF-t $\leftarrow 0$
- $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
- **return** $(M_{\text{chal}}, t_{\text{chal}})$

Execute (T, i, C, j):
- $\text{pin}_U \leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
- **if** $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** $\text{pin}_U = \perp$ **then**
  - **return** $\perp$
- $\pi_T^i \leftarrow \text{st}_T$
- trans, $m_C, m_T \leftarrow \perp$
- $\mathcal{L}_{\text{ct}}[\pi_C^j, \pi_T^i] \leftarrow (\perp, \perp, \perp)$
- **while** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **do**
  - $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
  - $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
  - trans $\leftarrow$ trans $||\ m_T\ ||\ m_C$
- $\mathcal{L}_{\text{CS}} \leftarrow \mathcal{L}_{\text{CS}} \cup \{\pi_C^j\}$
- **for all** $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
  - $\pi_T^{i'}.\text{isValid} \leftarrow$ false
- **return** trans

Figure E.39: Adversary $\mathcal{B}_{12}$'s simulation of game 12.

Validate-C (C, j, M, t):
- **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **or** $\pi_C^j.\text{isValid} =$ false **then return** $\perp$
- $(T, i) \leftarrow$ clientBindPartner(C, j)
- **if** $\pi_C^j = \mathcal{L}_{\text{CS}}[\text{guess} - 1]$ **then**
  - $(M_{\text{chal}}, t_{\text{chal}}) \leftarrow (M, t)$
  - return
- **else**
  - status $\leftarrow$ validate-C(π_C^j, M, t)
- **if** $(T, i, M, t) \notin \mathcal{L}_{\text{authT}}$ **then**
  - **if** $\pi_C^j.\text{compromised} =$ false **then**
    - **if** $\pi_T^i.\text{pinCorr} =$ false **then**
      - **if** status $=$ accepted **then**
        - $\text{bad}_{12} \leftarrow$ true
        - $(M_{\text{chal}}, t_{\text{chal}}) \leftarrow (M, t)$
        - return
      - **return** rejected
- **if** status $=$ accepted **then**
  - win-SUF-t $\leftarrow$ Client-Win-SUF-t(C, j, M, t)
- **return** status

Auth-T (T, i, M):
- **if** $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq$ bindDone **or** $\pi_T^i.\text{isValid} =$ false **then return** $\perp$
- $\pi_{C,\text{target}}^j \leftarrow \mathcal{L}_{\text{CS}}[\text{guess} - 1]$
- **if** $\pi_{C,\text{target}}^j \neq \perp \wedge (T, i) =$ clientBindPartner(C, j) **then**
  - $(M, t) \leftarrow \mathcal{O}_{\text{Auth}}(M)$
- **else**
  - $(M, t) \xleftarrow{\$}$ auth-T(π_T^i, M)
- $\mathcal{L}_{\text{authT}} \leftarrow \mathcal{L}_{\text{authT}} \cup \{(T, i, M, t)\}$
- **return** (M, t)

Compromise (C, j):
- **if** $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **then**
  - **return** $\perp$
- **if** $\pi_C^j = \mathcal{L}_{\text{CS}}[\text{guess} - 1]$ **then**
  - abort
- $\pi_C^j.\text{compromised} =$ True
- **return** $\pi_C^j.\text{bs}$

Figure E.40: Modified Auth-T, Validate-C and Compromise oracles simulated by $\mathcal{B}_{12}$.

$$n_{\text{CS}}\ \text{Adv}_{\text{MAC}}^{\text{SUF-CMA}}(\mathcal{B}_{12}) \geq \Pr_{12}[\text{bad}_{12}] \tag{E.55}$$

Since games 11 and 12 are identical until bad_{12} happens, by the difference lemma, we have:

$$\Pr_{12}[\text{bad}_{12}] \geq |\ \Pr_{11} - \Pr_{12}\ | \tag{E.56}$$

Thus, we have:

$$n_{\text{CS}}\ \text{Adv}_{\text{MAC}}^{\text{SUF-CMA}}(\mathcal{B}_{12}) \geq |\ \Pr_{11} - \Pr_{12}\ | \tag{E.57}$$

Since each client session can finish Bind only in Execute queries, we know that $n_{\mathsf{CS}} \leq q_{\mathsf{E}}$. Therefore:

$$q_{\mathsf{E}} \, \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_{12}) \geq | \, \mathsf{Pr}_{11} - \mathsf{Pr}_{12} \, | \tag{E.58}$$

Final Analysis

Note that, at this point, $\mathcal{A}$ cannot win against Game 12, since it must be able to win via one of the four conditions in Token-Win-SUF-t or one of the three conditions in Client-Win-SUF-t.

Win via Token-Win-SUF-t

Condition 1 of Token-Win-SUF-t ($d = 0$) is always false, since Token-Win-SUF-t is only called after a validation from the Validate-T oracle is successful, which is dependant on the user decision bit d.

Conditions 2 and 3 of Token-Win-SUF-t will also never be true. Indeed, since collisions between both client and token sid values have been ruled out in Game 9, it can never be the case that two different client or token sessions have the same sid. Thus, these conditions are always false.

Condition 4 of Token-Win-SUF-t is also always false. Recall that, for $\mathcal{A}$ to win, it must give a pair (M, t) to the Validate-T oracle for a token session π_T^i and (M, t) must pass validation. Additionally, π_T^i must not have had its pin corrupted, and either not have a partner or have a unique partner that was not compromised. In the latter case, (M, t) must also not have been output by that partner.

Indeed, if the target π_T^i which is validating (M, t) from $\mathcal{A}$ has a (unique) partner π_C^j, then $\mathcal{A}$ hasn't actively attacked π_T^i during the Bind session, and therefore has no information whatsoever about the pinUvAuthToken, since the experiment no longer transmits the encrypted pinUvAuthToken in passive Bind sessions.

If $\mathcal{A}$ actively attacked π_T^i by performing a Bind session itself, in order to attempt to obtain the pinUvAuthToken, the experiment always rejects, provided the pin was not corrupted. If it was corrupted, then $\mathcal{A}$ can never win against any session of that token. This eliminates the attack where $\mathcal{A}$ simply guesses the pin.

$\mathcal{A}$ can also never get any information whatsoever about the pin without corrupting the user U to which it belongs, since no pin is ever sampled before $\mathcal{A}$ either corrupts the user or

atempts to guess the pin for the first time by querying Send-Bind-T, and all ciphertexts c_p and c_{ph} transmitted by a client to a token have constant values.

Finally, $\mathcal{A}$ can never win by correctly forging a tag t, since the token session always rejects any valid forgery attempt from $\mathcal{A}$.

Win via Client-Win-SUF-t

Conditions 1 and 2 are identical to conditions 2 and 3 from Token-Win-SUF-t, so they are always false.

Condition 3, similarly to condition 4 of Token-Win-SUF-t, is also false, since $\mathcal{A}$ can never learn anything about the pinUvAuthToken used to authenticate messages to any client session without corrupting a pin or compromising the session itself, and can never win by actually submitting a valid forged tag t because the client session always rejects such an attempt.

Advantage measure

Given that $\mathcal{A}$ can no longer win against Game 12, we have:

$$\Pr_{12} = 0 \tag{E.59}$$

Note that the running times of all adversaries are close to that of $\mathcal{A}$.

This completes the proof.

F. sCTAP 2.1+ SUF Authentication Proof

In this Appendix, we present the complete pseudocode proof for the SUF authentication security of sCTAP 2.1+, as an instance of mPACA, with a sequence of games. We base our modifications for each game on the mPACA model oracles and sCTAP 2.1+ syntax as presented in Chapter 6.

Let $\Pr_g$ denote the probability that game g outputs 1. Each game's modifications that are *persistent* throughout the proof are showcased in orange. Modifications that exist only when employing a hybrid argument or a reduction, and therefore are only relevant to a specific game, are shown in green. Finally, bad events and the end of a reduction are marked in red and are also relevant only to their respective game.

Game 0

This game is the original $\mathrm{Expt}^{\mathsf{SUF}}_{\mathsf{sCTAP\ 2.1+}}(\mathcal{A})$ security experiment. Therefore,

$$\Pr_0 = \mathrm{Adv}^{\mathsf{SUF}}_{\mathsf{sCTAP\ 2.1+}}(\mathcal{A}) \tag{F.1}$$

Game 1

In this game we replace every key K_1 and K_2 derived from Z by all ECDH key exchange sessions with two independent and uniformly random session keys $\tilde{K}_1$ and $\tilde{K}_2$, and bound the advantage of $\mathcal{A}$ by reducing to the sCDH security of ECDH. Note that ECDH key exchanges only occur during Setup queries, so by the model's definition they are all passive.

We maintain two structures $\mathcal{L}_{\mathsf{sCDH}}$ and $\mathcal{L}_{\mathcal{H}_2}$ respectively to save ocurrences of client and token public keys in order to maintain consistency throughout the game and to save adversary queries to random oracle $\mathcal{H}_2$. Every execution of encapsulate, which happens only on clients, will first check if the current pair of public keys has appeared before. If it hasn't, the experiment samples the two session keys $\tilde{K}_1$ and $\tilde{K}_2$ and saves them in $\mathcal{L}_{\mathsf{sCDH}}$. Otherwise, the challenger uses the $\tilde{K}_1$ and $\tilde{K}_2$ which were previously stored in $\mathcal{L}_{\mathsf{sCDH}}$. On the token side, every time decapsulate executes, it simply uses the corresponding $\tilde{K}_1$ and $\tilde{K}_2$ from $\mathcal{L}_{\mathsf{sCDH}}$. This is always consistent because $\mathcal{A}$ can never deliver a public key of its own to any token session.

It is clear that, unless $\mathcal{A}$ ever queried $\mathcal{H}_2$ with the shared secret Z derived in a Setup query, Game 1 proceeds identically to Game 0.

Game1($\mathcal{A}$):
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: win-SUF $\leftarrow 0$
5: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
6: **return** win-SUF

Adv$\mathcal{H}_2$ (u_1, u_2):
1: $v \leftarrow \mathcal{H}_2(u_1, u_2)$
2: $\mathcal{L}_{\mathcal{H}_2}[u_1, u_2] \leftarrow v$
3: **return** v

Figure F.1: Game 1 on the left, and $\mathcal{H}_2$ wrapper Adv$\mathcal{H}_2$ on the right - Game 1. Adv$\mathcal{H}_2$ is a wrapper that $\mathcal{A}$ can use to call the random oracle, so that we can distinguish calls to $\mathcal{H}_2$ made specifically by $\mathcal{A}$.

encapsulate (pk'):
1: **if** $\mathcal{L}_{\text{sCDH}}[pk', pk] = \perp$ **then**
2: $\quad \tilde{K}_1 \xleftarrow{\$} \{0,1\}^{2\lambda}, \tilde{K}_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$
3: $\quad \tilde{K} \leftarrow (\tilde{K}_1, \tilde{K}_2)$
4: $\quad \mathcal{L}_{\text{sCDH}}[pk', pk] \leftarrow \tilde{K}$
5: **else**
6: $\quad \tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk', pk]$
7: $K \leftarrow \tilde{K}$
8: $c \leftarrow pk$
9: **return** (c, K)

decapsulate (c):
1: $\tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk, c]$
2: $K \leftarrow \tilde{K}$
3: **return** K

Figure F.2: Modifications to PIN/UV Auth Protocol functions encapsulate and decapsulate - Game 1.

Hybrid Argument

We bound the advantage of $\mathcal{A}$ against Game 1 by creating n_{TokenPK} hybrid games between Game 0 and Game 1, where n_{TokenPK} is the total number of token public keys sampled throughout the experiment that are output to $\mathcal{A}$ in Setup queries, via the obtainSharedSecret-T function. We then define a bad event bad on game $0.i$, where the game aborts and $\mathcal{A}$ loses if it queries $\mathcal{H}_2$ with $u_1 = Z = \text{XCoordinateOf}(sk \cdot pk^i_{\text{token}})$, and pk^i_{token} and c exist in $\mathcal{L}_{\text{sCDH}}$ at any point in the game, with pk^i_{token} as the i^{th} token public key and sk and c any client private and public key generated by a client session that received pk^i_{token}.

We use the notation from regular Diffie-Hellman instead of Elliptic Curve Diffie-Hellman to simplify the code and make it more legible. For example, instead of $Z \leftarrow \text{XCoordinateOf}(sk \cdot pk')$, we use $Z \leftarrow (pk')^{sk}$ during this hybrid and the reduction to sCDH. However, every operation used in this proof uses ECDH.

Reduction to the sCDH security of ECDH

We now construct an adversary $\mathcal{B}^i_1$ that simulates game $0.i$ to $\mathcal{A}$, in such a way that when bad happens, $\mathcal{B}^i_1$ wins against the sCDH security of ECDH.

Game0.$i(\mathcal{A})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: hy $\leftarrow i, \mathcal{L}_{hy} \leftarrow \emptyset$
5: $(pk^i_{\text{token}}, sk^i_{\text{token}}) \leftarrow (\perp, \perp)$
6: bad $\leftarrow$ false
7: win-SUF $\leftarrow 0$
8: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
9: **return** win-SUF

Adv$\mathcal{H}_2$ (u_1, u_2):
1: $v \leftarrow \mathcal{H}_2(u_1, u_2)$
2: **for all** $(pk_T, pk_C, _) \in \mathcal{L}_{\text{sCDH}}$ s.t. $pk_T = pk^i_{\text{token}}$ **do**
3: **if** $u_1 = (pk_C)^{sk^i_{\text{token}}}$ **then**
4: bad $\leftarrow$ true
5: abort
6: $\mathcal{L}_{\mathcal{H}_2}[u_1, u_2] \leftarrow v$
7: **return** v

Figure F.3: Hybrid game $0.i$ on the left and random oracle wrapper Adv$\mathcal{H}_2$ on the right.

obtainSharedSecret-T(π^i_T, puvProtocol):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList **then**
2: **return** $\perp$
3: $pk_T \leftarrow$ st$_T$.puvProtocol.getPublicKey()
4: **if** $(T, pk_T) \notin \mathcal{L}_{\text{hy}}$ **then**
5: $\mathcal{L}_{\text{hy}} \leftarrow \mathcal{L}_{\text{hy}} \cup (T, pk_T)$
6: **if** size$(\mathcal{L}_{hy})$ = hy **then**
7: $(pk^i_{\text{token}}, sk^i_{\text{token}}) \leftarrow (pk_T, \text{st}_T.\text{puvProtocol}.sk)$
8: selectpuvProtocol $\leftarrow$ puvProtocolList
9: π^i_T.st$_{\text{exe}} \leftarrow$ waiting
10: π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || pk_T
11: **return** pk_T

encapsulate (pk'):
1: **if** $(_, pk') \in \mathcal{L}_{hy}[0 : \text{hy}[$ **then**
2: **if** $\mathcal{L}_{\text{sCDH}}[pk', pk] = \perp$ **then**
3: $\tilde{K}_1 \xleftarrow{\$} \{0,1\}^{2\lambda}, \tilde{K}_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$
4: $\tilde{K} \leftarrow (\tilde{K}_1, \tilde{K}_2)$
5: $\mathcal{L}_{\text{sCDH}}[pk', pk] \leftarrow \tilde{K}$
6: **if** $pk' = pk^i_{\text{token}}$ **then**
7: **if** $\mathcal{L}_{\mathcal{H}_2}[(pk')^{sk}, _] \neq \perp$ **then**
8: bad $\leftarrow$ true
9: abort
10: **else**
11: $\tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk', pk]$
12: $K \leftarrow \tilde{K}$
13: $c \leftarrow pk$
14: **return** (c, K)
15: **if** $(_, pk') \in \mathcal{L}_{hy}[\text{hy} : n_{\text{TokenPK}}[$ **then**
16: $Z \leftarrow (pk')^{sk}$
17: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
18: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
19: $K \leftarrow (K_1, K_2)$
20: $c \leftarrow pk$
21: **return** (c, K)

decapsulate (c):
1: **if** $(_, pk) \in \mathcal{L}_{hy}[0 : \text{hy}[$ **then**
2: $\tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk, c]$
3: $K \leftarrow \tilde{K}$
4: **return** K
5: **if** $(_, pk) \in \mathcal{L}_{hy}[\text{hy} : n_{\text{TokenPK}}[$ **then**
6: $Z \leftarrow c^{sk}$
7: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
8: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
9: $K \leftarrow (K_1, K_2)$
10: **return** K

Figure F.4: obtainSharedSecret-T on the left and PIN/UV Auth Protocol encapsulate and decapsulate functions on the right, from hybrid game $0.i$.

Throughout the simulation of game $0.i$, $\mathcal{B}^i_1$ sets the i^{th} token public key as the sCDH challenge public key g^a and any client public key generated by a client session that receives g^a to be g^{b+r_w}, for $r_w \leftarrow \mathbb{Z}_q$ and $w \geq 1$. In such client sessions, $\mathcal{B}^i_1$ checks if $u_1 = Z$ is in $\mathcal{L}_{\text{sCDH}}$ using its $\mathcal{O}_a$ oracle. Additionally, every time $\mathcal{A}$ calls random oracle $\mathcal{H}_2$ (always through Adv$\mathcal{H}_2$) $\mathcal{B}^i_1$ also checks if for any public key pair in $\mathcal{L}_{\text{sCDH}}$ that includes g^a, $u_1 = Z$.

It is not hard to see that, whenever bad happens during $\mathcal{B}^i_1$'s simulation, $\mathcal{B}^i_1$ wins by using $\mathcal{A}$'s input to the random oracle $\mathcal{H}_2$. Indeed, if $\mathcal{A}$ ever calls $\mathcal{H}_2$ with input $u_1 = Z = (g^{b+r_w})^a, \forall r_w \in \mathcal{L}_{r_w}$, then $\mathcal{B}^i_1$ can output $Z' \leftarrow u_1 \cdot (g^a)^{-r_w} \equiv (g^{b+r_w})^a \cdot (g^a)^{-r_w} \equiv g^{ab}$ and win the sCDH game.

Advantage measure

By the definition of $\mathcal{B}^i_1$, if bad happens, then $\mathcal{A}$ found the solution to the problem given by the sCDH challenger to $\mathcal{B}^i_1$, i.e., g^{ab}. Therefore:

$\mathcal{B}_1^i(g, g^a, g^b, \mathcal{A}, \mathcal{O}_a)$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: hy $\leftarrow i, \mathcal{L}_{hy} \leftarrow \emptyset$
5: $(pk^i_{\text{token}}, sk^i_{\text{token}}) \leftarrow (\perp, \perp)$
6: $\mathcal{L}_{r_w}, \mathcal{L}_{\text{Fake}\mathcal{H}_2} \leftarrow \emptyset$
7: $Z' \xleftarrow{\$} \mathbb{Z}_q$
8: win-SUF $\leftarrow 0$
9: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
10: ▷ *If bad ever happened, Z' has the correct value because it was set during the game simulation from $\mathcal{B}_1^i$.*
11: **return** Z'

Adv$\mathcal{H}_2$ (u_1, u_2):
1: $v \leftarrow \mathcal{H}_2(u_1, u_2)$
2: **for all** $(r_w, g^b \cdot g^{r_w}) \in \mathcal{L}_{r_w}$ **do**
3: **if** $\mathcal{L}_{\text{sCDH}}[g^a, g^b \cdot g^{r_w}] \neq \perp$ **then**
4: **if** $\mathcal{O}_a(g^b \cdot g^{r_w}, u_1) =$ true **then**
5: $Z' = u_1 \cdot (g^a)^{-r_w}$
6: win-sCDH
7: $\mathcal{L}_{\mathcal{H}_2}[u_1, u_2] \leftarrow v$
8: **return** v

Figure F.5: $\mathcal{B}_1^i$'s simulation of game $0.i$ on the right.

encapsulate (pk'):
1: **if** $(_, pk') \in \mathcal{L}_{hy}[0 : hy[$ **then**
2: **if** $pk' = pk^i_{\text{token}}$ **then**
3: $w \leftarrow \text{size}(\mathcal{L}_{r_w}) + 1$
4: $r_w \xleftarrow{\$} \mathbb{Z}_q$
5: $pk \leftarrow g^b \cdot g^{r_w}$
6: $\mathcal{L}_{r_w} \leftarrow \mathcal{L}_{r_w} \cup \{(r_w, pk)\}$
7: **if** $\mathcal{L}_{\text{sCDH}}[pk', pk] = \perp$ **then**
8: $\tilde{K}_1 \xleftarrow{\$} \{0,1\}^{2\lambda}, \tilde{K}_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$
9: $\tilde{K} \leftarrow (\tilde{K}_1, \tilde{K}_2)$
10: $\mathcal{L}_{\text{sCDH}}[pk', pk] \leftarrow \tilde{K}$
11: **if** $pk' = pk^i_{\text{token}}$ **then**
12: **for all** $(u_1, u_2, v) \in \mathcal{L}_{\mathcal{H}_2}$ **do**
13: **if** $\mathcal{O}_a(pk, u_1) =$ true **then**
14: ▷ $u_1 = (g^b \cdot g^{r_w})^a$
15: $Z' = u_1 \cdot (g^a)^{-r_w}$
16: win-sCDH
17: **else**
18: $\tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk', pk]$
19: $K \leftarrow \tilde{K}$
20: $c \leftarrow pk$
21: **return** (c, K)
22: **if** $(_, pk') \in \mathcal{L}_{hy}[hy : n_{\text{TokenPK}}[$ **then**
23: $Z \leftarrow (pk')^{sk}$
24: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
25: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
26: $K \leftarrow (K_1, K_2)$
27: $c \leftarrow pk$
28: **return** (c, K)

decapsulate (c):
1: **if** $(_, pk) \in \mathcal{L}_{hy}[0 : hy[$ **then**
2: $\tilde{K} \leftarrow \mathcal{L}_{\text{sCDH}}[pk, c]$
3: **if** $\tilde{K} \neq \perp$ **then**
4: $K \leftarrow \tilde{K}$
5: **return** K
6: **if** $(_, pk) \in \mathcal{L}_{hy}[hy : n_{\text{TokenPK}}[$ **then**
7: $Z \leftarrow c^{sk}$
8: $K_1 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 HMAC KEY"})$
9: $K_2 \leftarrow \mathcal{H}_2(Z, \text{"CTAP2 AES KEY"})$
10: $K \leftarrow (K_1, K_2)$
11: **return** K

obtainSharedSecret-T(π^i_T, puvProtocol):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList **then**
2: **return** $\perp$
3: $pk_T \leftarrow$ st$_T$.puvProtocol.getPublicKey()
4: **if** $(T, pk_T) \notin \mathcal{L}_{\text{hy}}$ **then**
5: $\mathcal{L}_{\text{hy}} \leftarrow \mathcal{L}_{\text{hy}} \cup (T, pk_T)$
6: **if** size$(\mathcal{L}_{hy}) =$ hy **then**
7: $(pk^i_{\text{token}}, sk^i_{\text{token}}) \leftarrow (g^a, _)$
8: selectpuvProtocol $\leftarrow$ puvProtocolList
9: π^i_T.st$_{\text{exe}} \leftarrow$ waiting
10: π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || pk_T
11: **return** pk_T

Figure F.6: $\mathcal{B}_1^i$ simulations of encapsulate, on the left, and decapsulate and obtainSharedSecret-T on the right.

$$\Pr_{0.i}[\text{bad}] \leq \text{Adv}^{\text{sCDH}}_{\text{ECDH}}(\mathcal{B}_1^i) \tag{F.2}$$

Since games $0.i$ and $0.(i-1)$ are identical until bad happens, through the difference lemma, we have:

$$\Pr_{0.i}[\text{bad}] \geq \mid \Pr_{0.(i-1)} - \Pr_{0.i} \mid, \forall i \in [1, n_{\text{TokenPK}}] \tag{F.3}$$

$$\mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1^i) \geq \mid \mathsf{Pr}_{0.(i-1)} - \mathsf{Pr}_{0.i} \mid, \forall i \in [1, n_{\mathsf{TokenPK}}] \tag{F.4}$$

$$\sum_{i=1}^{n_{\mathsf{TokenPK}}} \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1^i) \geq \sum_{i=1}^{n_{\mathsf{TokenPK}}} \mid \mathsf{Pr}_{0.(i-1)} - \mathsf{Pr}_{0.i} \mid \tag{F.5}$$

$$\sum_{i=1}^{n_{\mathsf{TokenPK}}} \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1^i) \geq \mid \mathsf{Pr}_{0.0} - \mathsf{Pr}_{0.n_{\mathsf{TokenPK}}} \mid \tag{F.6}$$

Let $\mathcal{B}_1$ be the adversary with the highest advantage against the sCDH game out of all adversaries $\mathcal{B}_1^i$, $\forall i \in [1, n_{\mathsf{TokenPK}}]$. Then, it is always true that:

$$n_{\mathsf{TokenPK}}\ \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1) \geq \mid \mathsf{Pr}_{0.0} - \mathsf{Pr}_{0.n_{\mathsf{TokenPK}}} \mid \tag{F.7}$$

Since Game 0.0 corresponds to Game 0 and Game $0.n_{\mathsf{TokenPK}}$ corresponds to Game 1, we have:

$$n_{\mathsf{TokenPK}}\ \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1) \geq \mid \mathsf{Pr}_0 - \mathsf{Pr}_1 \mid \tag{F.8}$$

Given that token ECDH public keys can only be given to $\mathcal{A}$ in Setup queries, we know $n_{\mathsf{TokenPK}} \leq q_{\mathsf{S}}$. Therefore:

$$q_{\mathsf{S}}\ \mathsf{Adv}^{\mathsf{sCDH}}_{\mathsf{ECDH}}(\mathcal{B}_1) \geq \mid \mathsf{Pr}_0 - \mathsf{Pr}_1 \mid \tag{F.9}$$

Game 2

In this game, the pin is no longer saved in the token's state, and is stored instead on a new structure $\mathcal{L}_{\mathsf{TokenSetup}}$ which is not available to $\mathcal{A}$.

Game2($\mathcal{A}$):
1: $\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\mathsf{valid}}, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
5: win-SUF $\leftarrow 0$
6: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
7: **return** win-SUF

Figure F.7: Game 2.

setPIN-T(π^i_T, puvProtocol, c, c_p, t_p, uName):
1: **if** puvProtocol $\notin$ st_T.puvProtocolList $\vee$ $\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then**
2: **return** $\perp$
3: $K \leftarrow \text{st}_T$.puvProtocol.decapsulate(c)
4: $m \leftarrow c_p$ || uName
5: **if** $K = \perp \vee \text{st}_T$.puvProtocol.verify$(K, m, t_p) =$ false **then**
6: **return** $\perp$
7: pin $\leftarrow \text{st}_T$.puvProtocol.decrypt(K, c_p)
8: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
9: $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow$ pin
10: $\text{st}_T.\text{id}_U =$ uName || 0
11: $\text{st}_T.\text{id}_T =$ uName || 1
12: st_T.pinRetries $\leftarrow$ pinRetriesMax
13: **return** accepted

obtainSessionKey-T-start(π^i_T, puvProtocol, initMsg):
1: **if** puvProtocol $\notin$ st_T.puvProtocolList **then return** $\perp$
2: pin $\leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
3: st_T.puvProtocol.regenerate(initMsg, pin, $\text{st}_T.\text{id}_U, \text{st}_T.\text{id}_T$)
4: $pk_T \leftarrow \text{st}_T$.puvProtocol.getPublicKey()
5: $\pi^i_T.\text{st}_{\text{exe}} \leftarrow$ waiting
6: π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || initMsg || pk_T
7: **return** pk_T

Figure F.8: Modified sCTAP 2.1+ setPIN-T and obtainSessionKey-T-start functions - Game 2.

Advantage measure

Since this model does not allow tokens to be corrupted to reveal their state st_T, Game 2 is functionally identical to Game 1, and we have:

$$\text{Pr}_1 = \text{Pr}_2 \tag{F.10}$$

Game 3

In this game, we replace all pins encrypted by any client session and sent to any token session during Setup with a constant pin 0000, and bound the advantage of $\mathcal{A}$ by reducing to the IND-1CPA security of SKE.

Recall that in the sCTAP 2.1+ protocol, each ephemeral session key K derived from the ECDH key exchange on Setup is *only* used once to encrypt the transmission of the pin.

Game2($\mathcal{A}$):
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: win-SUF $\leftarrow 0$
7: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
8: **return** win-SUF

Setup (T, i, C, j, U):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
2: **if** $\text{st}_T = \perp$ **or** $\pi^i_T \neq \perp$ **or** $\pi^j_C \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
3: $\pi^i_T \leftarrow \text{st}_T$
4: π^j_C.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
5: $\mathcal{L}_{\text{pin}}[\pi^i_T] \leftarrow$ pin
6: trans $\xleftarrow{\$}$ Setup$(\pi^i_T, \pi^j_C,$ pin)
7: π^i_T.isValid, π^j_C.isValid $\leftarrow$ false
8: st_T.user $\leftarrow U$
9: **return** trans

Figure F.9: Game 3 and modified Setup oracle.

setPIN-C(π_C^j, pin, uName):

1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: $c_p \xleftarrow{\$} \pi_C^j$.puvProtocol.encrypt($\pi_C^j$.K, 0000)
3: $m \leftarrow c_p \,||$ uName
4: $t_p \xleftarrow{\$} \pi_C^j$.puvProtocol.authenticate(π_C^j.K, m)
5: **return** (c_p, t_p, uName)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p, uName):

1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ $\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then**
2: **return** $\perp$
3: $K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
4: $m \leftarrow c_p \,||$ uName
5: **if** $K = \perp \vee$ st$_T$.puvProtocol.verify(K, m, t_p) = false **then**
6: **return** $\perp$
7: pin $\leftarrow \mathcal{L}_{\text{pin}}[\pi_T^i]$
8: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
9: $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow$ pin
10: st$_T$.id$_U$ = uName $|| \, 0$
11: st$_T$.id$_T$ = uName $|| \, 1$
12: st$_T$.pinRetries $\leftarrow$ pinRetriesMax
13: **return** accepted

Figure F.10: Modified sCTAP 2.1+ functions setPIN-C and setPIN-T - Game 3.

Hybrid Argument

To bound $\mathcal{A}$'s advantage against Game 3, we begin by creating n_{setup} hybrid games between Game 2 and Game 3, where n_{setup} is the maximum number of (valid) calls to the Setup oracle (valid refers to calls that actually result in executing the Setup between a client session and token session). Then, we replace the pin with the constant value 0000 only until the i^{th} execution of Setup.

Game2.i($\mathcal{A}$):

1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: hy $\leftarrow i$, count$_{\text{setup}} \leftarrow 0$
7: win-SUF $\leftarrow 0$
8: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
9: **return** win-SUF

Setup (T, i, C, j, U):

1: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
2: **if** st$_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
3: $\pi_T^i \leftarrow$ st$_T$
4: π_C^j.pinCorr $\leftarrow \mathcal{L}_{\text{corrupt}}$[st$_T$.user]
5: count$_{\text{setup}} \leftarrow$ count$_{\text{setup}} + 1$
6: $\mathcal{L}_{\text{pin}}[\pi_T^i] \leftarrow$ pin
7: trans $\xleftarrow{\$}$ Setup(π_T^i, π_C^j, pin)
8: π_T^i.isValid, π_C^j.isValid $\leftarrow$ false
9: st$_T$.user $\leftarrow U$
10: **return** trans

Figure F.11: Hybrid game 2.i.

Reduction to the IND-1CPA security of SKE

We now construct an adversary $\mathcal{B}_3^i$ against the IND-1CPA security of SKE that simulates game 2.i to $\mathcal{A}$ perfectly, except when $\mathcal{A}$ performs the i^{th} valid query to Setup. In this case, when executing setPIN-C, $\mathcal{B}_3^i$ sends pin and 0000 to $\mathcal{O}_{\text{LR}}$ and receives a challenge ciphertext c_p, instead of encrypting it normally with the derived session key $\tilde{K}_2$. Then, when

setPIN-C(π^j_C, pin, uName):
1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
3: $\quad c_p \xleftarrow{\$} \pi^j_C.\text{puvProtocol.encrypt}(\pi^j_C.\text{K}, 0000)$
4: **else**
5: $\quad c_p \xleftarrow{\$} \pi^j_C.\text{puvProtocol.encrypt}(\pi^j_C.\text{K}, \text{pin})$
6: $m \leftarrow c_p \,||\, \text{uName}$
7: $t_p \xleftarrow{\$} \pi^j_C.\text{puvProtocol.authenticate}(\pi^j_C.\text{K}, m)$
8: **return** (c_p, t_p, uName)

setPIN-T(π^i_T, puvProtocol, c, c_p, t_p, uName):
1: **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee$
$\quad \mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then**
2: $\quad$ **return** $\perp$
3: $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
4: $m \leftarrow c_p \,||\, \text{uName}$
5: **if** $K = \perp \vee \text{st}_T.\text{puvProtocol.verify}(K, m, t_p) = \text{false}$ **then**
6: $\quad$ **return** $\perp$
7: **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
8: $\quad \text{pin} \leftarrow \mathcal{L}_{\text{pin}}[\pi^i_T]$
9: **else**
10: $\quad \text{pin} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_p)$
11: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
12: $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow \text{pin}$
13: $\text{st}_T.\text{id}_U = \text{uName} \,||\, 0$
14: $\text{st}_T.\text{id}_T = \text{uName} \,||\, 1$
15: $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
16: **return** accepted

Figure F.12: Modified sCTAP 2.1+ functions setPIN-C and setPIN-T for hybrid game $2.i$.

running setPIN-T on the corresponding token session, $\mathcal{B}^i_3$ sets the pin that was saved in $\mathcal{L}_{\text{pin}}$ in the beginning of the Setup oracle execution.

$\mathcal{B}^i_3(\mathcal{A}, \mathcal{O}_{\text{LR}})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: $\text{hy} \leftarrow i, \text{count}_{\text{setup}} \leftarrow 0$
7: $c_{\text{chal}} \leftarrow \perp$
8: win-SUF $\leftarrow 0$
9: $b' \leftarrow \perp$
10: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
11: **if** win-SUF $= 1$ **then**
12: $\quad$ **return** $b' \leftarrow 0$
13: **else**
14: $\quad$ **return** $b' \leftarrow 1$
15: **return** b'

Setup (T, i, C, j, U):
1: pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
2: **if** $\text{st}_T = \perp$ **or** $\pi^i_T \neq \perp$ **or** $\pi^j_C \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
3: $\pi^i_T \leftarrow \text{st}_T$
4: $\pi^j_C.\text{pinCorr} \leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
5: $\text{count}_{\text{setup}} \leftarrow \text{count}_{\text{setup}} + 1$
6: $\mathcal{L}_{\text{pin}}[\pi^i_T] \leftarrow \text{pin}$
7: trans $\xleftarrow{\$}$ Setup$(\pi^i_T, \pi^j_C, \text{pin})$
8: $\pi^i_T.\text{isValid}, \pi^j_C.\text{isValid} \leftarrow \text{false}$
9: $\text{st}_T.\text{user} \leftarrow U$
10: **return** trans

Figure F.13: $\mathcal{B}^i_3$'s simulation of game $2.i$.

Advantage measure

The advantage of $\mathcal{B}^i_3$ against the IND-1CPA security of SKE is:

$$\text{Adv}^{\text{IND-1CPA}}_{\text{SKE}}(\mathcal{B}^i_3) = |\ \Pr_{\text{IND-1CPA}}[b' = 0 | b = 1] - \Pr_{\text{IND-1CPA}}[b' = 0 | b = 0]\ | \qquad \text{(F.11)}$$

When oracle $\mathcal{O}_{\text{LR}}$ encrypts m_0, which corresponds to the real pin, then $\mathcal{B}^i_3$ is simulating Game $2.(i-1)$ to $\mathcal{A}$. If instead it encrypts m_1, corresponding to the constant pin 0000, then it simulates Game $2.i$.

From the definition of $\mathcal{B}^i_3$, it follows that:

setPIN-C(π_C^j, pin, uName):
1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
3: **if** $\text{count}_{\text{setup}} = \text{hy}$ **then**
4: $c_p \leftarrow \mathcal{O}_{\text{LR}}(\text{pin}, 0000)$
5: **else**
6: $c_p \xleftarrow{\$} \pi_C^j.\text{puvProtocol.encrypt}(\pi_C^j.\text{K}, 0000)$
7: **else**
8: $c_p \xleftarrow{\$} \pi_C^j.\text{puvProtocol.encrypt}(\pi_C^j.\text{K}, \text{pin})$
9: $m \leftarrow c_p \,||\, \text{uName}$
10: $t_p \xleftarrow{\$} \pi_C^j.\text{puvProtocol.authenticate}(\pi_C^j.\text{K}, m)$
11: **return** (c_p, t_p, uName)

setPIN-T(π_T^i, puvProtocol, c, c_p, t_p, uName):
1: **if** puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee$
$\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then**
2: **return** $\perp$
3: $K \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
4: $m \leftarrow c_p \,||\, \text{uName}$
5: **if** $K = \perp \vee \text{st}_T.\text{puvProtocol.verify}(K, m, t_p) = \text{false}$
then
6: **return** $\perp$
7: **if** $\text{count}_{\text{setup}} \leq \text{hy}$ **then**
8: $\text{pin} \leftarrow \mathcal{L}_{\text{pin}}[\pi_T^i]$
9: **else**
10: $\text{pin} \leftarrow \text{st}_T.\text{puvProtocol.decrypt}(K, c_p)$
11: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
12: $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow \text{pin}$
13: $\text{st}_T.\text{id}_U = \text{uName} \,||\, 0$
14: $\text{st}_T.\text{id}_T = \text{uName} \,||\, 1$
15: $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
16: **return** accepted

Figure F.14: $\mathcal{B}_3^i$'s simulation of sCTAP 2.1+ functions setPIN-C and setPIN-T.

$$\text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3^i) = \mid \text{Pr}_{2.i}[\text{win-SUF} = 1] - \text{Pr}_{2.(i-1)}[\text{win-SUF} = 1] \mid, \forall i \in [1, n_{\text{setup}}] \quad \text{(F.12)}$$

$$\text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3^i) = \mid \text{Pr}_{2.i} - \text{Pr}_{2.(i-1)} \mid, \forall i \in [1, n_{\text{setup}}] \quad \text{(F.13)}$$

$$\sum_{i=1}^{n_{\text{setup}}} \text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3^i) = \sum_{i=1}^{n_{\text{setup}}} \mid \text{Pr}_{2.i} - \text{Pr}_{2.(i-1)} \mid \quad \text{(F.14)}$$

$$\sum_{i=1}^{n_{\text{setup}}} \text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3^i) \geq \mid \text{Pr}_{2.0} - \text{Pr}_{2.n_{\text{setup}}} \mid \quad \text{(F.15)}$$

Let $\mathcal{B}_3$ be the adversary with the highest advantage against the IND-1CPA game, $\forall i \in [1, n_{\text{setup}}]$. Then, it is always true that:

$$n_{\text{setup}} \ \text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3) \geq \sum_{i=1}^{n_{\text{setup}}} \text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3^i) \quad \text{(F.16)}$$

And thus,

$$n_{\text{setup}} \ \text{Adv}_{\text{SKE}}^{\text{IND-1CPA}}(\mathcal{B}_3) \geq \mid \text{Pr}_{2.0} - \text{Pr}_{2.n_{\text{setup}}} \mid \quad \text{(F.17)}$$

Since Game 2.0 corresponds to Game 2 and Game $2.n_{\mathsf{setup}}$ corresponds to Game 3, we have:

$$n_{\mathsf{setup}} \, \mathsf{Adv}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{B}_3) \geq \mid \mathsf{Pr}_2 - \mathsf{Pr}_3 \mid \tag{F.18}$$

And, since n_{setup} is the maximum amount of valid Setup queries made by $\mathcal{A}$, we have that $n_{\mathsf{setup}} \leq q_{\mathsf{S}}$. Therefore:

$$q_{\mathsf{S}} \, \mathsf{Adv}^{\mathsf{IND\text{-}1CPA}}_{\mathsf{SKE}}(\mathcal{B}_3) \geq \mid \mathsf{Pr}_2 - \mathsf{Pr}_3 \mid \tag{F.19}$$

Game 4

In this game, the game aborts and $\mathcal{A}$ loses when causing a bad event by breaking the explicit mutual authentication guarantee of the underlying PAKE construction, i.e., if it manages to make any token session π_T^i or client session π_C^j successfully terminate Bind without having a unique PAKE partner (recall that two PAKE instances are partnered if and only if both have reached *accept*, agree on the same sid, and have derived the same session key).

We consider the PAKE sid to be the trace of communications between the two PAKE instances, set when each instance *accepts*. Therefore, we maintain two structures $\mathcal{L}_{\mathsf{cTrace}}$ and $\mathcal{L}_{\mathsf{tTrace}}$ to keep track of the trace of communications between client and token sessions. Then, for both sides, bad occurs if (i) a session *terminates*, which happens in $\mathsf{obtainSessionKey\text{-}T\text{-}end}$ for token sessions (note that in this function the token session reaches both *accept* and *terminate*) and $\mathsf{obtainSessionKey\text{-}C\text{-}end}$ for client sessions, and (ii) either $\mathcal{A}$ has not corrupted the pin and the session has no PAKE partner, or the session has more than one PAKE partner. Note that, in $\mathsf{obtainSessionKey\text{-}T\text{-}end}$, the client session has already accepted, and thus its sid is set when the token session terminates. Likewise, in $\mathsf{obtainSessionKey\text{-}C\text{-}end}$, when the client session terminates, the token has also already terminated (and thus accepted). This matches the definition of explicit mutual authentication introduced in Chapter 2.

Therefore, we bound the advantage of $\mathcal{A}$ by reducing to the ma security of the underlying PAKE.

Game4 ($\mathcal{A}$):
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: $\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
7: bad $\leftarrow$ false
8: win-SUF $\leftarrow 0$
9: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
10: **return** win-SUF

obtainSessionKey-C-start(π_C^j, info, pin, uName):
1: Parse(version, puvProtocolList) $\leftarrow$ info
2: **if** version $= 2.0$ **then return** $\perp$
3: select puvProtocol $\leftarrow$ puvProtocolList
4: π_C^j.puvProtocol $\leftarrow$ puvProtocol
5: $\text{id}_U =$ uName $||\ 0$
6: $\text{id}_T =$ uName $||\ 1$
7: initMsg $\xleftarrow{\$}$ π_C^j.puvProtocol.initialize-C(pin, id_U, id_T)
8: π_C^j.$\text{st}_{\text{exe}} \leftarrow$ waiting
9: π_C^j.sid $\leftarrow \pi_C^j$.sid $||$ info $||$ puvProtocol $||$ initMsg
10: $\mathcal{L}_{\text{cTrace}}[\pi_C^j] \leftarrow$ initMsg
11: **return** (puvProtocol, initMsg)

obtainSessionKey-C-accept(π_C^j, pk_T):
1: $c \leftarrow \pi_C^j$.puvProtocol.encapsulate(pk_T)
2: π_C^j.$\text{st}_{\text{exe}} \leftarrow$ bindStart
3: π_C^j.sid $\leftarrow \pi_C^j$.sid $||\ pk_T ||$ puvProtocol $||\ c$
4: $\mathcal{L}_{\text{cTrace}}[\pi_C^j] \leftarrow \mathcal{L}_{\text{cTrace}}[\pi_C^j] ||\ pk_T ||\ c$
5: **return** c

obtainSessionKey-C-end(π_C^j, t):
1: **if** π_C^j.$\text{st}_{\text{exe}} =$ bindError **then**
2: **return** $\perp$
3: $(K, \text{valid}) \leftarrow \pi_C^j$.puvProtocol.finalizeBind(t)
4: **if** valid $=$ false **then**
5: π_C^j.$\text{st}_{\text{exe}} \leftarrow$ bindError
6: **return** $\perp$
7: ▷ $\mathcal{A}$ *did not corrupt the* pin *but managed to make the client accept without a (PAKE) partner*
8: **if** π_C^j.pinCorr $=$ false $\wedge$ $|\text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| = 0$ **then**
9: bad $\leftarrow$ true
10: abort
11: ▷ $\mathcal{A}$ *managed to make at least 2 token sessions agree on trace. Even if the* pin *is corrupted,* $\mathcal{A}$ *should not be able to do this, given the randomness created by the token in respStart.*
12: **if** $|\text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| > 1$ **then**
13: bad $\leftarrow$ true
14: abort
15: π_C^j.bs $\leftarrow K$
16: π_C^j.$\text{st}_{\text{exe}} \leftarrow$ bindDone
17: π_C^j.sid $\leftarrow \pi_C^j$.sid $||\ t ||$ false
18: **return** accepted

obtainSessionKey-T-start(π_T^i, puvProtocol, initMsg):
1: **if** puvProtocol $\notin \text{st}_T$.puvProtocolList **then return** $\perp$
2: pin $\leftarrow \mathcal{L}_{\text{TokenSetup}}[T]$
3: st_T.puvProtocol.regenerate(initMsg, pin, $\text{st}_T.\text{id}_U, \text{st}_T.\text{id}_T$)
4: $pk_T \leftarrow \text{st}_T$.puvProtocol.getPublicKey()
5: π_T^i.$\text{st}_{\text{exe}} \leftarrow$ waiting
6: π_T^i.sid $\leftarrow \pi_T^i$.sid $||$ puvProtocol $||$ initMsg $||\ pk_T$
7: $\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow$ initMsg $||\ pk_T$
8: **return** pk_T

obtainSessionKey-T-end(π_T^i, puvProtocol, c):
1: **if** puvProtocol $\notin \text{st}_T$.puvProtocolList $\vee\ \text{st}_T$.pinRetries $= 0$ **then**
2: **return** $(\perp, \text{false})$
3: st_T.pinRetries $\leftarrow$ pinRetries $- 1$
4: $(K, t) \leftarrow \text{st}_T$.puvProtocol.decapsulate(c)
5: **if** $(K, t) = \perp$ **then**
6: st_T.puvProtocol.initialize-T()
7: $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
8: **if** $\text{st}_T.m = 0$ **then**
9: authPowerUp-T(st_T)
10: **return** $(\perp, \text{true})$
11: **else**
12: **return** $(\perp, \text{false})$
13: $\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi_T^i] ||\ c$
14: ▷ $\mathcal{A}$ *did not corrupt the* pin *but managed to make the token accept without a (PAKE) partner*
15: **if** π_T^i.pinCorr $=$ false $\wedge$ $|\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| = 0$ **then**
16: bad $\leftarrow$ true
17: abort
18: ▷ $\mathcal{A}$ *managed to make at least 2 client sessions agree on trace so far. Even if the* pin *is corrupted,* $\mathcal{A}$ *should not be able to do this, given the randomness created by the Client in initStart and initAccept.*
19: **if** $|\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| > 1$ **then**
20: bad $\leftarrow$ true
21: abort
22: $\text{st}_T.m \leftarrow 3$
23: st_T.pinRetries $\leftarrow$ pinRetriesMax
24: **for all** puvProtocol' $\in \text{st}_T$.puvProtocolList **do**
25: st_T.puvProtocol'.resetPuvToken()
26: π_T^i.bs $\leftarrow \text{st}_T$.puvProtocol.pt
27: π_T^i.$\text{st}_{\text{exe}} \leftarrow$ bindDone
28: π_T^i.canValidate $\leftarrow$ true
29: π_T^i.sid $\leftarrow \pi_T^i$.sid $||$ puvProtocol $||\ c ||\ t ||$ false
30: **return** (t, false)

Figure F.15: Game 4.

Reduction to the ma security of the PAKE

We construct an adversary $\mathcal{B}_4$ against the ma security of the underlying PAKE that perfectly simulates Game 4 to $\mathcal{A}$, such that when bad happens, $\mathcal{B}_4$ wins the ma game.

The ma challenger first runs $\mathcal{O}^{\text{Initialization}}$ to sample passwords from $\mathcal{P}$ following distribution $\mathscr{D}$ for each pair of parties (in this PAKE model, each party is identified by a unique string that matches the unique identifiers we defined for sCTAP 2.1+). $\mathcal{B}_4$ simulates NewU by sampling a constant pin 0 and deriving the two user and token identites id_U and id_T from U.uName, which are saved so that they can be used in the Setup oracle later. CorruptUser can be simulated by calling the $\mathcal{O}^{\text{Corrupt}}$ oracle on U's identity id_U and its token's identity id_T. The Setup oracle does not need any PAKE oracles to be simulated, and uses the pair $(\text{id}_U, \text{id}_T)$ obtained in NewU as the identities that must be saved in the token. Execute is simulated perfectly by calling $\mathcal{O}^{\text{Execute}}$, saving the trace of communications and plugging each message as the next message to be sent by either π_C^j or π_T^i. Send-Bind-C and Send-Bind-T can be simulated using $\mathcal{O}^{\text{Send}^i}$. The session key K, set as the binding state, can be obtained by calling $\mathcal{O}^{\text{Reveal}}$, either by querying it eagerly on the last step of Bind for (obtainSessionKey-T-end for π_T^i and obtainSessionKey-C-end for π_C^j) or lazily when Compromise or one of the authentication/verification oracles is called.

Advantage measure

Because Games 3 and 4 are identical until bad, through the difference lemma, we have:

$$\mid \Pr_3 - \Pr_4 \mid \leq \Pr_4[\text{bad}] \tag{F.20}$$

Since bad only occurs when the explicit mutual authentication (ma) guarantee of PAKE is broken, we have:

$$\Pr_4[\text{bad}] \leq \text{Adv}^{\text{ma}}_{\text{PAKE}}(\mathcal{B}_4) \tag{F.21}$$

Therefore:

$$\mid \Pr_3 - \Pr_4 \mid \leq \text{Adv}^{\text{ma}}_{\text{PAKE}}(\mathcal{B}_4) \tag{F.22}$$

$\mathcal{B}_4(\mathcal{A}, \mathcal{O}^{\text{Send}^i}, \mathcal{O}^{\text{Execute}}, \mathcal{O}^{\text{Test}}, \mathcal{O}^{\text{Corrupt}}, \mathcal{O}^{\text{Reveal}})$:
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
bad $\leftarrow$ false
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
if bad $=$ true **then**
  win-pake-ma
else
  lose-pake-ma

Execute (T, i, C, j):
pin $\leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
if $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then return** $\perp$
$\pi_T^i \leftarrow \text{st}_T$
$\pi_C^j.\text{pinCorr} \leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
trans, $m_C, m_T \leftarrow \perp$
$(\text{msg}_1, \text{msg}_2, \text{msg}_3, \text{msg}_4, \text{msg}_5) \xleftarrow{\$} \mathcal{O}^{\text{Execute}}(\text{st}_T.\text{id}_T, i, \text{st}_T.\text{id}_U, j)$
$\pi_C^j.\text{msg}_1 \leftarrow \text{msg}_1$, $\pi_C^j.\text{msg}_3 \leftarrow \text{msg}_3$, $\pi_C^j.\text{msg}_5 \leftarrow \text{msg}_5$
$\pi_T^i.\text{msg}_2 \leftarrow \text{msg}_2$, $\pi_T^i.\text{msg}_4 \leftarrow \text{msg}_4$
while $\pi_C^j.\text{st}_{\text{exe}} \neq$ bindDone **do**
  $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
  $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
  trans $\leftarrow$ trans $\|\, m_T \,\|\, m_C$
for all $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
  $\pi_T^{i'}.\text{isValid} \leftarrow$ false
return trans

CorruptUser (U):
if $\mathcal{L}_{\text{valid}}[U] = \perp$ **then return** $\perp$
$\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ true
$(\text{id}_U, \text{id}_T) \leftarrow \mathcal{L}_{\text{ids}}[U]$
pin $\leftarrow \mathcal{O}^{\text{Corrupt}}(\text{id}_U, \text{id}_T)$
return pin

NewU (U):
if $\mathcal{L}_{\text{valid}}[U] = \perp$ **then**
  pin $\leftarrow 0$
  $\mathcal{L}_{\text{valid}}[U] \leftarrow$ pin
  $\mathcal{L}_{\text{corrupt}}[U] \leftarrow$ false
  $\text{id}_U \leftarrow U.\text{uName} \,\|\, 0$
  $\text{id}_T \leftarrow U.\text{uName} \,\|\, 1$
  $\mathcal{L}_{\text{ids}}[U] \leftarrow (\text{id}_U, \text{id}_T)$
return

Setup (T, i, C, j, U):
pin $\leftarrow \mathcal{L}_{\text{valid}}[U]$
if $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then**
  return $\perp$
$\pi_T^i \leftarrow \text{st}_T$
$\pi_C^j.\text{pinCorr} \leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
$\mathcal{L}_{\text{pin}}[\pi_T^i] \leftarrow$ pin
$\pi_T^i.\text{ids} \leftarrow \mathcal{L}_{\text{ids}}[U]$
trans $\xleftarrow{\$}$ Setup$(\pi_T^i, \pi_C^j, \text{pin})$
$\pi_T^i.\text{isValid}, \pi_C^j.\text{isValid} \leftarrow$ false
$\text{st}_T.\text{user} \leftarrow U$
return trans

Send-Bind-C (C, j, m, U):
if $\pi_C^j = \perp$ **then**
  $\pi_C^j.\text{pinCorr} \leftarrow \mathcal{L}_{\text{corrupt}}[U]$
if $\pi_C^j.\text{st}_{\text{exe}} =$ bindDone **then return** $\perp$
if $\pi_C^j.\text{isValid} =$ false **then return** $\perp$
if $\mathcal{L}_{\text{valid}}[U] = \perp$ **then return** $\perp$
$\pi_C^j.\text{ids} \leftarrow \mathcal{L}_{\text{ids}}[U]$
$m_C \leftarrow$ Bind-C(π_C^j, U, m)
return m_C

Figure F.16: $\mathcal{B}_4$'s simulation of Game 4.

Game 5

In this game, unless a session's pin was corrupted *and* $\mathcal{A}$ engaged with that session using Send-Bind-T or Send-Bind-C, we replace every session key K derived when a token or client session finishes Bind successfully with an independent random key $\tilde{K}$, and bound the advantage of $\mathcal{A}$ by reducing to the *perfect forward secrecy* (pfs) security of PAKE.

We introduce two new structures $\mathcal{L}_{\text{passive}}$ and $\mathcal{L}_{\text{K}}$ to identify passive Bind executions via Execute and to save the independent random keys $\tilde{K}$ sampled in token sessions to be used in their partner client sessions.

setPIN-C(π^j_C, pin, uName):
1: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
2: $c_p \xleftarrow{\$} \pi^j_C$.puvProtocol.encrypt($\pi^j_C$.K, 0000)
3: $m \leftarrow c_p$ || uName
4: $t_p \xleftarrow{\$} \pi^j_C$.puvProtocol.authenticate(π^j_C.K, m)
5: **return** (c_p, t_p, uName)

obtainSessionKey-C-start(π^j_C, info, pin, uName):
1: Parse(version, puvProtocolList) $\leftarrow$ info
2: **if** version $= 2.0$ **then return** $\perp$
3: select puvProtocol $\leftarrow$ puvProtocolList
4: π^j_C.puvProtocol $\leftarrow$ puvProtocol
5: $(\text{id}_U, \text{id}_T) \leftarrow \pi^j_C$.ids
6: ▷ *If this is true, then this function was called from an (m)PACA Execute oracle.*
7: **if** π^j_C.msg$_1 \neq \perp$ **then**
8: initMsg $\leftarrow \pi^j_C$.msg$_1$
9: **else**
10: initMsg $\xleftarrow{\$} \mathcal{O}^{\text{Send}^0}(\text{id}_U, j, \perp)$
11: π^j_C.st$_{\text{exe}} \leftarrow$ waiting
12: π^j_C.sid $\leftarrow \pi^j_C$.sid || info || puvProtocol || initMsg
13: $\mathcal{L}_{\text{cTrace}}[\pi^j_C] \leftarrow$ initMsg
14: **return** (puvProtocol, initMsg)

obtainSessionKey-C-accept(π^j_C, pk_T):
1: $(\text{id}_U, \text{id}_T) \leftarrow \pi^j_C$.ids
2: **if** π^i_T.msg$_3 \neq \perp$ **then**
3: $c \leftarrow \pi^j_C$.msg$_3$
4: **else**
5: $c \xleftarrow{\$} \mathcal{O}^{\text{Send}^2}(\text{id}_U, j, pk_T)$
6: π^j_C.st$_{\text{exe}} \leftarrow$ bindStart
7: π^j_C.sid $\leftarrow \pi^j_C$.sid || pk_T || puvProtocol || c
8: $\mathcal{L}_{\text{cTrace}}[\pi^j_C] \leftarrow \mathcal{L}_{\text{cTrace}}[\pi^j_C]$ || pk_T || c
9: **return** c

obtainSessionKey-C-end(π^j_C, t):
1: **if** π^j_C.st$_{\text{exe}} =$ bindError **then**
2: **return** $\perp$
3: $(\text{id}_U, \text{id}_T) \leftarrow \pi^j_C$.ids
4: **if** π^j_C.msg$_5 \neq \perp$ **then**
5: valid $\leftarrow \pi^j_C$.msg$_5$
6: **else**
7: valid $\xleftarrow{\$} \mathcal{O}^{\text{Send}^4}(\text{id}_U, j, t)$
8: **if** valid $=$ false **then**
9: π^j_C.st$_{\text{exe}} \leftarrow$ bindError
10: **return** $\perp$
11: $K \leftarrow \mathcal{O}^{\text{Reveal}}(\text{id}_U, j)$
12: **if** π^j_C.pinCorr $=$ false $\wedge$ $|$ trace $\in \mathcal{L}_{\text{tTrace}}$: trace $= \mathcal{L}_{\text{cTrace}}[\pi^j_C]| = 0$ **then**
13: bad $\leftarrow$ true
14: abort
15: **if** $|$ trace $\in \mathcal{L}_{\text{cTrace}}$: trace $= \mathcal{L}_{\text{cTrace}}[\pi^j_C]| > 1$ **then**
16: bad $\leftarrow$ true
17: abort
18: π^j_C.bs $\leftarrow K$
19: π^j_C.st$_{\text{exe}} \leftarrow$ bindDone
20: π^j_C.sid $\leftarrow \pi^j_C$.sid || t || false
21: **return** accepted

setPIN-T(π^i_T, puvProtocol, c, c_p, t_p, uName):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ $\mathcal{L}_{\text{TokenSetup}}[T] \neq \perp$ **then**
2: **return** $\perp$
3: $K \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
4: $m \leftarrow c_p$ || uName
5: **if** $K = \perp \vee$ st$_T$.puvProtocol.verify(K, m, t_p) $=$ false **then**
6: **return** $\perp$
7: pin $\leftarrow \mathcal{L}_{\text{pin}}[\pi^i_T]$
8: **if** pin $\notin \mathcal{P}$ **then return** $\perp$
9: $\mathcal{L}_{\text{TokenSetup}}[T] \leftarrow$ pin
10: $(\text{id}_U, \text{id}_T) \leftarrow \pi^i_T$.ids
11: st$_T$.id$_U =$ id$_U$
12: st$_T$.id$_T =$ id$_T$
13: st$_T$.pinRetries $\leftarrow$ pinRetriesMax
14: **return** accepted

obtainSessionKey-T-start(π^i_T, puvProtocol, initMsg):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList **then return** $\perp$
2: **if** π^i_T.msg$_2 \neq \perp$ **then**
3: $pk_T \leftarrow \pi^j_C$.msg$_2$
4: **else**
5: $pk_T \xleftarrow{\$} \mathcal{O}^{\text{Send}^1}(\text{st}_T.\text{id}_T, i, \text{initMsg})$
6: π^i_T.st$_{\text{exe}} \leftarrow$ waiting
7: π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || initMsg || pk_T
8: $\mathcal{L}_{\text{tTrace}}[\pi^i_T] \leftarrow$ initMsg || pk_T
9: **return** pk_T

obtainSessionKey-T-end(π^i_T, puvProtocol, c):
1: **if** puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then**
2: **return** ($\perp$, false)
3: st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
4: **if** π^i_T.msg$_4 \neq \perp$ **then**
5: $t \leftarrow \pi^j_C$.msg$_4$
6: **else**
7: $t \xleftarrow{\$} \mathcal{O}^{\text{Send}^3}(\text{st}_T.\text{id}_T, i, c)$
8: $K \leftarrow \mathcal{O}^{\text{Reveal}}(\text{st}_T.\text{id}_T, i)$
9: **if** $(K, t) = \perp$ **then**
10: st$_T$.puvProtocol.initialize-T()
11: st$_T.m \leftarrow$ st$_T.m - 1$
12: **if** st$_T.m = 0$ **then**
13: authPowerUp-T(st$_T$)
14: **return** ($\perp$, true)
15: **else**
16: **return** ($\perp$, false)
17: $\mathcal{L}_{\text{tTrace}}[\pi^i_T] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi^i_T]$ || c
18: **if** π^i_T.pinCorr $=$ false $\wedge$ $|$ trace $\in \mathcal{L}_{\text{cTrace}}$: trace $= \mathcal{L}_{\text{tTrace}}[\pi^i_T]| = 0$ **then**
19: bad $\leftarrow$ true
20: abort
21: **if** $|$ trace $\in \mathcal{L}_{\text{cTrace}}$: trace $= \mathcal{L}_{\text{tTrace}}[\pi^i_T]| > 1$ **then**
22: bad $\leftarrow$ true
23: abort
24: st$_T.m \leftarrow 3$
25: st$_T$.pinRetries $\leftarrow$ pinRetriesMax
26: **for all** puvProtocol' $\in$ st$_T$.puvProtocolList **do**
27: st$_T$.puvProtocol'.resetPuvToken()
28: π^i_T.bs $\leftarrow K$
29: π^i_T.st$_{\text{exe}} \leftarrow$ bindDone
30: π^i_T.canValidate $\leftarrow$ true
31: π^i_T.sid $\leftarrow \pi^i_T$.sid || puvProtocol || c || t || false
32: **return** (t, false)

Figure F.17: $\mathcal{B}_4$'s simulation of Game 4 - continuation.

Game5 ($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF

Execute (T, i, C, j):
pin $\leftarrow \mathcal{L}_{\text{valid}}[\text{st}_T.\text{user}]$
if $\text{st}_T = \perp$ **or** $\pi_T^i \neq \perp$ **or** $\pi_C^j \neq \perp$ **or** pin $= \perp$ **then return** $\perp\!\!\!\perp$
$\pi_T^i \leftarrow \text{st}_T$
$\pi_C^j.\text{pinCorr} \leftarrow \mathcal{L}_{\text{corrupt}}[\text{st}_T.\text{user}]$
trans, $m_C, m_T \leftarrow \perp$
▷ *Flag π_C^j and π_T^i as passive sessions, the value itself does not matter here*
$\mathcal{L}_{\text{passive}}[\pi_C^j, \pi_T^i] \leftarrow \text{true}$
while $\pi_C^j.\text{st}_{\text{exe}} \neq \text{bindDone}$ **do**
  $m_T \xleftarrow{\$}$ Bind-T(π_T^i, m_C)
  $m_C \xleftarrow{\$}$ Bind-C(π_C^j, U, m_T)
  trans $\leftarrow$ trans $||\ m_T\ ||\ m_C$
for all $i' \neq i$ **and** $\pi_T^{i'} \neq \perp$ **do**
  $\pi_T^{i'}.\text{isValid} \leftarrow$ false
return trans

obtainSessionKey-C-end(π_C^j, t):
if $\pi_C^j.\text{st}_{\text{exe}} = \text{bindError}$ **then**
  return $\perp$
$(K, \text{valid}) \leftarrow \pi_C^j.\text{puvProtocol.finalizeBind}(t)$
if valid $=$ false **then**
  $\pi_C^j.\text{st}_{\text{exe}} \leftarrow \text{bindError}$
  return $\perp$
if $\mathcal{L}_{\text{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\text{K}}[t]$
if $\pi_C^j.\text{pinCorr} = \text{false} \wedge\ |\ \text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]\ | = 0$ **then**
  abort
if $|\ \text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]\ | > 1$ **then**
  abort
$\pi_C^j.\text{bs} \leftarrow K$
$\pi_C^j.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
$\pi_C^j.\text{sid} \leftarrow \pi_C^j.\text{sid}\ ||\ t\ ||$ false
return accepted

obtainSessionKey-T-end$(\pi_T^i, \text{puvProtocol}, c)$:
if $\text{puvProtocol} \notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  return $(\perp, \text{false})$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
$(K, t) \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
if $(K, t) = \perp$ **then**
  $\text{st}_T.\text{puvProtocol.initialize-T}()$
  $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
  if $\text{st}_T.m = 0$ **then**
    authPowerUp-T(st_T)
    return $(\perp, \text{true})$
  else
    return $(\perp, \text{false})$
$\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ ||\ c$
if $\pi_T^i.\text{pinCorr} = \text{false} \wedge\ |\ \text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ | = 0$ **then**
  abort
if $|\ \text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ | > 1$ **then**
  abort
$\text{st}_T.m \leftarrow 3$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
for all puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
▷ *Capture only cases where execution of this function in passive via Execute. In this case, we always know what client session π_C^j is associated with the current token session π_T^i.*
if $\mathcal{L}_{\text{passive}}[_, \pi_T^i] = \text{true} \vee\ \pi_T^i.\text{pinCorr} = \text{false}$ **then**
  $\tilde{K} \xleftarrow{\$} \{0,1\}^{2\lambda}$
  $\mathcal{L}_{\text{K}}[t] \leftarrow \tilde{K}$
  $K \leftarrow \tilde{K}$
  $\pi_T^i.\text{bs} \leftarrow K$
else
  $\pi_T^i.\text{bs} \leftarrow \text{st}_T.\text{puvProtocol}.pt$
$\pi_T^i.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
$\pi_T^i.\text{canValidate} \leftarrow \text{true}$
$\pi_T^i.\text{sid} \leftarrow \pi_T^i.\text{sid}\ ||\ \text{puvProtocol}\ ||\ c\ ||\ t\ ||$ false
return (t, false)

Figure F.18: Game 5.

Reduction to the pfs security of the PAKE

We construct an adversary $\mathcal{B}_5$ against the pfs security of the underlying PAKE that perfectly simulates Game 4 to $\mathcal{A}$ when the challenger's challenge bit b is 0 and Game 5 when the challenge bit b is 1.

This simulation is very similar to the simulation of Game 4 by $\mathcal{B}_4$, except when running obtainSessionKey-T-end and obtainSessionKey-C-end. The former no longer queries $\mathcal{O}^{\text{Reveal}}$ to obtain the session key K to set as the binding state, except when responding to a Send-Bind-T when the pin has been corrupted. Instead, $\mathcal{B}_5$ queries its $\mathcal{O}^{\text{Test}}$ oracle, to obtain

either the correct session key K or an independent random key $\tilde{K}$. This key is saved in $\mathcal{L}_{\mathsf{K}}$. The latter simply uses the key in $\mathcal{L}_{\mathsf{K}}$ if it exists, and calls $\mathcal{O}^{\mathsf{Reveal}}$ if it doesn't.

Finally, $\mathcal{B}_5$ binds its guess b' to the success of $\mathcal{A}$ in winning the simulation. If $\mathcal{A}$ wins (win-SUF =1), then $\mathcal{B}_5$ outputs $b' = 0$, otherwise it outputs $b' = 1$.

$\mathcal{B}_5(\mathcal{A}, \mathcal{O}^{\mathsf{Send}^i}, \mathcal{O}^{\mathsf{Execute}}, \mathcal{O}^{\mathsf{Test}}, \mathcal{O}^{\mathsf{Corrupt}}, \mathcal{O}^{\mathsf{Reveal}})$:
$\mathcal{L}_{\mathsf{authC}}, \mathcal{L}_{\mathsf{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{valid}}, \mathcal{L}_{\mathsf{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\mathsf{passive}} \leftarrow \emptyset, \mathcal{L}_{\mathsf{K}} \leftarrow \emptyset$
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
▷ *Test oracle from [27] returns random K with $b = 1$*
if win-SUF $= 0$ **then**
  return 1
else
  return 0

obtainSessionKey-C-end(π_C^j, t):
if $\pi_C^j.\mathsf{st}_{\mathsf{exe}} =$ bindError **then**
  return $\perp$
$(\mathsf{id}_U, \mathsf{id}_T) \leftarrow \pi_C^j.\mathsf{ids}$
if $\pi_C^j.\mathsf{msg}_5 \neq \perp$ **then**
  valid $\leftarrow \pi_C^j.\mathsf{msg}_5$
else
  valid $\xleftarrow{\$} \mathcal{O}^{\mathsf{Send}^4}(\mathsf{id}_U, j, t)$
if valid $=$ false **then**
  $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindError
  return $\perp$
if $\mathcal{L}_{\mathsf{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\mathsf{K}}[t]$
else
  $K \leftarrow \mathcal{O}^{\mathsf{Reveal}}(\mathsf{id}_U, j)$
if $\pi_C^j.\mathsf{pinCorr} =$ false $\wedge$ $|\,\mathsf{trace} \in \mathcal{L}_{\mathsf{tTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{cTrace}}[\pi_C^j]\,| = 0$ **then**
  abort
if $|\,\mathsf{trace} \in \mathcal{L}_{\mathsf{tTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{cTrace}}[\pi_C^j]\,| > 1$ **then**
  abort
$\pi_C^j.\mathsf{bs} \leftarrow K$
$\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindDone
$\pi_C^j.\mathsf{sid} \leftarrow \pi_C^j.\mathsf{sid} \,||\, t \,||$ false
return accepted

obtainSessionKey-T-end(π_T^i, puvProtocol, c):
if puvProtocol $\notin \mathsf{st}_T.\mathsf{puvProtocolList} \vee \mathsf{st}_T.\mathsf{pinRetries} = 0$ **then**
  return $(\perp,$ false$)$
$\mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetries} - 1$
if $\pi_T^i.\mathsf{msg}_4 \neq \perp$ **then**
  $t \leftarrow \pi_C^j.\mathsf{msg}_4$
else
  $t \xleftarrow{\$} \mathcal{O}^{\mathsf{Send}^3}(\mathsf{st}_T.\mathsf{id}_T, i, c)$
if $t = \perp$ **then**
  $\mathsf{st}_T.\mathsf{puvProtocol.initialize\text{-}T}()$
  $\mathsf{st}_T.m \leftarrow \mathsf{st}_T.m - 1$
  if $\mathsf{st}_T.m = 0$ **then**
    $\mathsf{authPowerUp\text{-}T}(\mathsf{st}_T)$
    return $(\perp,$ true$)$
  else
    return $(\perp,$ false$)$
$\mathcal{L}_{\mathsf{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\mathsf{tTrace}}[\pi_T^i] \,||\, c$
if $\pi_T^i.\mathsf{pinCorr} =$ false $\wedge$ $|\,\mathsf{trace} \in \mathcal{L}_{\mathsf{cTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{tTrace}}[\pi_T^i]\,| = 0$ **then**
  abort
if $|\,\mathsf{trace} \in \mathcal{L}_{\mathsf{cTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{tTrace}}[\pi_T^i]\,| > 1$ **then**
  abort
$\mathsf{st}_T.m \leftarrow 3$
$\mathsf{st}_T.\mathsf{pinRetries} \leftarrow \mathsf{pinRetriesMax}$
for all puvProtocol' $\in \mathsf{st}_T.\mathsf{puvProtocolList}$ **do**
  $\mathsf{st}_T.\mathsf{puvProtocol'.resetPuvToken}()$
if $\mathcal{L}_{\mathsf{passive}}[_, \pi_T^i] =$ true $\vee$ $\pi_T^i.\mathsf{pinCorr} =$ false **then**
  $\tilde{K} \leftarrow \mathcal{O}^{\mathsf{Test}}(\mathsf{id}_T, i)$
  $\mathcal{L}_{\mathsf{K}}[t] \leftarrow \tilde{K}$
  $K \leftarrow \tilde{K}$
  $\pi_T^i.\mathsf{bs} \leftarrow K$
else
  $K \leftarrow \mathcal{O}^{\mathsf{Reveal}}(\mathsf{st}_T.\mathsf{id}_T, i)$
  $\pi_T^i.\mathsf{bs} \leftarrow K$
$\pi_T^i.\mathsf{st}_{\mathsf{exe}} \leftarrow$ bindDone
$\pi_T^i.\mathsf{canValidate} \leftarrow$ true
$\pi_T^i.\mathsf{sid} \leftarrow \pi_T^i.\mathsf{sid} \,||\, \mathsf{puvProtocol} \,||\, c \,||\, t \,||$ false
return $(t,$ false$)$

Figure F.19: $\mathcal{B}_5$'s simulation of Game 5. Parts that are identical to the reduction in Game 4 are omitted.

Advantage measure

The advantage of $\mathcal{B}_5$ against the pfs security of PAKE is:

$$\mathsf{Adv}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{B}_5) = |\, \Pr_{\mathsf{pfs}}[b' = 0 | b = 0] - \Pr_{\mathsf{pfs}}[b' = 0 | b = 1] \,| \tag{F.23}$$

In the pfs game, when $b = 0$, oracle $\mathcal{O}^{\mathsf{Test}}$ returns the derived key K to $\mathcal{A}$, which is set as the binding state for that token session π_T^i during the Execute query. In this case, $\mathcal{B}_5$

is simulating Game 4 to $\mathcal{A}$. If $b = 1$, then $\mathcal{O}^{\mathsf{Test}}$ returns a $\tilde{K}$ sampled uniformly at random, which is then set as the binding state of π_T^i, and $\mathcal{B}_5$ is simulating Game 5 to $\mathcal{A}$.

From the definition of $\mathcal{B}_5$, it follows that:

$$\mathsf{Adv}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{B}_5) = \mid \Pr_4[\text{win-SUF} = 1] - \Pr_5[\text{win-SUF} = 1] \mid \tag{F.24}$$

$$\mathsf{Adv}^{\mathsf{pfs}}_{\mathsf{PAKE}}(\mathcal{B}_5) = \mid \Pr_4 - \Pr_5 \mid \tag{F.25}$$

Game 6

In this game, the game aborts and $\mathcal{A}$ loses when causing a bad event by breaking the sid collision resistance guarantee of the underlying PAKE construction, i.e., if any two token or client sessions successfully terminate Bind with the same PAKE sid. We reuse the same $\mathcal{L}_{\mathsf{tTrace}}$ and $\mathcal{L}_{\mathsf{cTrace}}$ structures from Game 5 to keep track of the PAKE sid values for token and client sessions, respectively.

Reduction to the sid-coll security of the PAKE

We now construct an adversary $\mathcal{B}_6$ against the sid-coll security of the underlying PAKE that perfectly simulates Game 6 to $\mathcal{A}$, such that when bad happens, $\mathcal{B}_6$ wins the sid-coll game.

Advantage measure

Because Games 5 and 6 are identical until bad, through the difference lemma, we have:

$$\mid \Pr_5 - \Pr_6 \mid \leq \Pr_6[\mathsf{bad}] \tag{F.26}$$

Since bad only occurs when the sid collision resistance (sid-coll) guarantee of PAKE is broken, we have:

$$\Pr_6[\mathsf{bad}] \leq \mathsf{Adv}^{\mathsf{sid\text{-}coll}}_{\mathsf{PAKE}}(\mathcal{B}_6) \tag{F.27}$$

Therefore:

$$\mid \Pr_5 - \Pr_6 \mid \leq \mathsf{Adv}^{\mathsf{sid\text{-}coll}}_{\mathsf{PAKE}}(\mathcal{B}_6) \tag{F.28}$$

Game6 ($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF

obtainSessionKey-C-end(π_C^j, t):
if $\pi_C^j.\text{st}_{\text{exe}} =$ bindError **then**
  return $\perp$
$(K, \text{valid}) \leftarrow \pi_C^j.\text{puvProtocol.finalizeBind}(t)$
if valid $=$ false **then**
  $\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindError
  return $\perp$
if $\mathcal{L}_{\text{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\text{K}}[t]$
▷ *If the trace (sid) in $\mathcal{L}_{cTrace}[\pi_C^j]$ is not unique, a collision of client sids occured*
if $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| > 1$ **then**
  bad $\leftarrow$ true
  abort
if $\pi_C^j.\text{pinCorr} =$ false $\wedge$ $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| = 0$ **then**
  abort
if $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| > 1$ **then**
  abort
$\pi_C^j.\text{bs} \leftarrow K$
$\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindDone
$\pi_C^j.\text{sid} \leftarrow \pi_C^j.\text{sid} \,||\, t \,||$ false
return accepted

obtainSessionKey-T-end(π_T^i, puvProtocol, c):
if puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  return $(\perp,$ false$)$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
$(K, t) \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
if $(K, t) = \perp$ **then**
  $\text{st}_T.\text{puvProtocol.initialize-T}()$
  $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
  if $\text{st}_T.m = 0$ **then**
    authPowerUp-T(st_T)
    return $(\perp,$ true$)$
  else
    return $(\perp,$ false$)$
$\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi_T^i] \,||\, c$
▷ *If the trace (sid) in $\mathcal{L}_{tTrace}[\pi_T^i]$ is not unique, a collision of token sids occured*
if $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| > 1$ **then**
  bad $\leftarrow$ true
  abort
if $\pi_T^i.\text{pinCorr} =$ false $\wedge$ $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| = 0$ **then**
  abort
if $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| > 1$ **then**
  abort
$\text{st}_T.m \leftarrow 3$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
for all puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
if $\mathcal{L}_{\text{passive}}[_, \pi_T^i] =$ true $\vee$ $\pi_T^i.\text{pinCorr} =$ false **then**
  $\tilde{K} \xleftarrow{\$} \{0,1\}^{2\lambda}$
  $\mathcal{L}_{\text{K}}[t] \leftarrow \tilde{K}$
  $K \leftarrow \tilde{K}$
  $\pi_T^i.\text{bs} \leftarrow K$
else
  $\pi_T^i.\text{bs} \leftarrow \text{st}_T.\text{puvProtocol}.pt$
$\pi_T^i.\text{st}_{\text{exe}} \leftarrow$ bindDone
$\pi_T^i.\text{canValidate} \leftarrow$ true
$\pi_T^i.\text{sid} \leftarrow \pi_T^i.\text{sid} \,||$ puvProtocol $\,||\, c \,||\, t \,||$ false
return $(t,$ false$)$

Figure F.20: Game 6.

Game 7

In this game, $\mathcal{A}$ loses if, at any point in the game, it causes a bad event by querying random oracle $\mathcal{H}_3$ with *any* K generated throughout the experiment and set as the binding state of some token or client session, but for which it has no information. We maintain two new structures $\mathcal{L}_{\mathcal{H}_3}$ and $\mathcal{L}_{\text{bs}}$ to keep track of all of $\mathcal{A}$'s queries to $\mathcal{H}_3$ and of all binding states about which $\mathcal{A}$ has no information, respectively.

Advantage measure

Note that, at this point in the game, $\mathcal{A}$ has no information about any binding state that results from Execute queries or Send-Bind-T queries where it behaves passively, because in these circumstances, after Game 6, K is sampled uniformly at random. Therefore, we can upper bound the amount of binding states that can cause the bad event as $(q_{\text{E}} +$

$\mathcal{B}_6(\mathcal{A}, \mathcal{O}^{\text{Send}^i}, \mathcal{O}^{\text{Execute}}, \mathcal{O}^{\text{Test}}, \mathcal{O}^{\text{Corrupt}}, \mathcal{O}^{\text{Reveal}})$:

$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
if bad $=$ true **then**
  win-pake-sid-coll
else
  lose-pake-sid-coll

obtainSessionKey-C-end(π_C^j, t):

if $\pi_C^j.\text{st}_{\text{exe}} =$ bindError **then**
  return $\perp$
$(\text{id}_U, \text{id}_T) \leftarrow \pi_C^j.\text{ids}$
if $\pi_C^j.\text{msg}_5 \neq \perp$ **then**
  valid $\leftarrow \pi_C^j.\text{msg}_5$
else
  valid $\xleftarrow{\$} \mathcal{O}^{\text{Send}^4}(\text{id}_U, j, t)$
if valid $=$ false **then**
  $\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindError
  return $\perp$
if $\mathcal{L}_{\text{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\text{K}}[t]$
else
  $K \leftarrow \mathcal{O}^{\text{Reveal}}(\text{id}_U, j)$
if $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| > 1$ **then**
  bad $\leftarrow$ true
  abort
if $\pi_C^j.\text{pinCorr} =$ false $\wedge$ $|\text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| = 0$ **then**
  abort
if $|\text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]| > 1$ **then**
  abort
$\pi_C^j.\text{bs} \leftarrow K$
$\pi_C^j.\text{st}_{\text{exe}} \leftarrow$ bindDone
$\pi_C^j.\text{sid} \leftarrow \pi_C^j.\text{sid} \,||\, t \,||$ false
return accepted

obtainSessionKey-T-end(π_T^i, puvProtocol, c):

if puvProtocol $\notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
  return $(\perp,$ false$)$
if $\pi_T^i.\text{msg}_4 \neq \perp$ **then**
  $t \leftarrow \pi_C^j.\text{msg}_4$
else
  $t \xleftarrow{\$} \mathcal{O}^{\text{Send}^3}(\text{st}_T.\text{id}_T, i, c)$
if $(K, t) = \perp$ **then**
  $\text{st}_T.\text{puvProtocol.initialize-T}()$
  $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
  if $\text{st}_T.m = 0$ **then**
    $\text{authPowerUp-T}(\text{st}_T)$
    return $(\perp,$ true$)$
  else
    return $(\perp,$ false$)$
$\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi_T^i] \,||\, c$
if $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| > 1$ **then**
  bad $\leftarrow$ true
  abort
if $\pi_T^i.\text{pinCorr} =$ false $\wedge$ $|\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| = 0$ **then**
  abort
if $|\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]| > 1$ **then**
  abort
$\text{st}_T.m \leftarrow 3$
$\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
for all puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
  $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
if $\mathcal{L}_{\text{passive}}[_, \pi_T^i] =$ true $\vee$ $\pi_T^i.\text{pinCorr} =$ false **then**
  $\tilde{K} \xleftarrow{\$} \{0,1\}^{2\lambda}$
  $\mathcal{L}_{\text{K}}[t] \leftarrow \tilde{K}$
  $K \leftarrow \tilde{K}$
  $\pi_T^i.\text{bs} \leftarrow K$
else
  $K \leftarrow \mathcal{O}^{\text{Reveal}}(\text{st}_T.\text{id}_T, i)$
  $\pi_T^i.\text{bs} \leftarrow K$
$\pi_T^i.\text{st}_{\text{exe}} \leftarrow$ bindDone
$\pi_T^i.\text{canValidate} \leftarrow$ true
$\pi_T^i.\text{sid} \leftarrow \pi_T^i.\text{sid} \,||\, \text{puvProtocol} \,||\, c \,||\, t \,||$ false
return $(t,$ false$)$

Figure F.21: $\mathcal{B}_6$'s simulation of Game 6.

$q_{\text{Send-T}}$) (in every scenario where bad can occur, Send-Bind-C queries simply use the same K sampled in Send-Bind-T, so the number of different binding states does not increase).

Let $q_{\mathcal{H}_3}$ be the maximum amount of queries $\mathcal{A}$ performs during this game. Given that the maximum amount of session keys K (i.e., binding states) that can possibly exist is $2^{2\lambda}$, we know that the probability that $\mathcal{A}$ queries $\mathcal{H}_3$ with one of the K that it has no information about is upper bounded by $q_{\mathcal{H}_3}\ (q_{\text{E}} + q_{\text{Send-T}})/2^{2\lambda}$. Therefore, we have have that:

$$\Pr_7[\text{bad}] \leq \frac{q_{\mathcal{H}_3}\ (q_{\text{E}} + q_{\text{Send-T}})}{2^{2\lambda}} \tag{F.29}$$

Since games 6 and 7 are identical until bad, by the difference lemma, we have:

Game7($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
$\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
bad $\leftarrow$ false
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
return win-SUF

Adv$\mathcal{H}_3$ (u_1):
$v \leftarrow \mathcal{H}_3(u_1)$
if $u_1 \in \mathcal{L}_{\text{bs}}$ **then**
  bad $\leftarrow$ true
  abort
$\mathcal{L}_{\mathcal{H}_3}[u_1] \leftarrow v$
return v

obtainSessionKey-C-end(π_C^j, t):
if π_C^j.st$_{\text{exe}}$ = bindError **then**
  return $\perp$
$(K, \text{valid}) \leftarrow \pi_C^j$.puvProtocol.finalizeBind(t)
if valid = false **then**
  π_C^j.st$_{\text{exe}}$ $\leftarrow$ bindError
  return $\perp$
if $\mathcal{L}_{\text{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\text{K}}[t]$
  ▷ *$\mathcal{A}$ has no information about K.*
  if $\mathcal{L}_{\mathcal{H}_3}[K] \neq \perp$ **then**
    bad $\leftarrow$ true
    abort
  $\mathcal{L}_{\text{bs}} \leftarrow \mathcal{L}_{\text{bs}} \cup \{K\}$
if $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]\ | > 1$ **then**
  abort
if π_C^j.pinCorr = false $\wedge$ $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]\ | = 0$ **then**
  abort
if $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{cTrace}}[\pi_C^j]\ | > 1$ **then**
  abort
π_C^j.bs $\leftarrow K$
π_C^j.st$_{\text{exe}}$ $\leftarrow$ bindDone
π_C^j.sid $\leftarrow \pi_C^j$.sid || t || false
return accepted

obtainSessionKey-T-end(π_T^i, puvProtocol, c):
if puvProtocol $\notin$ st$_T$.puvProtocolList $\vee$ st$_T$.pinRetries $= 0$ **then**
  return ($\perp$, false)
st$_T$.pinRetries $\leftarrow$ pinRetries $- 1$
$(K, t) \leftarrow$ st$_T$.puvProtocol.decapsulate(c)
if $(K, t) = \perp$ **then**
  st$_T$.puvProtocol.initialize-T()
  st$_T$.$m \leftarrow$ st$_T$.$m - 1$
  if st$_T$.$m = 0$ **then**
    authPowerUp-T(st$_T$)
    return ($\perp$, true)
  else
    return ($\perp$, false)
$\mathcal{L}_{\text{tTrace}}[\pi_T^i] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi_T^i]$ || c
if $|\text{ trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ | > 1$ **then**
  abort
if π_T^i.pinCorr = false $\wedge$ $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ | = 0$ **then**
  abort
if $|\text{ trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi_T^i]\ | > 1$ **then**
  abort
st$_T$.$m \leftarrow 3$
st$_T$.pinRetries $\leftarrow$ pinRetriesMax
for all puvProtocol' $\in$ st$_T$.puvProtocolList **do**
  st$_T$.puvProtocol'.resetPuvToken()
if $\mathcal{L}_{\text{passive}}[_, \pi_T^i] = \text{true} \vee \pi_T^i$.pinCorr = false **then**
  $\tilde{K} \xleftarrow{\$} \{0,1\}^{2\lambda}$
  $\mathcal{L}_{\text{K}}[t] \leftarrow \tilde{K}$
  $K \leftarrow \tilde{K}$
  π_T^i.bs $\leftarrow K$
  ▷ *$\mathcal{A}$ has no information about K.*
  if $\mathcal{L}_{\mathcal{H}_3}[\pi_T^i.\text{bs}] \neq \perp$ **then**
    bad $\leftarrow$ true
    abort
  $\mathcal{L}_{\text{bs}} \leftarrow \mathcal{L}_{\text{bs}} \cup \{\pi_T^i.\text{bs}\}$
else
  ▷ *Here $\mathcal{A}$ knows π_T^i.bs, because this only happens when the pin was corrupted and $\mathcal{A}$ actively engaged with π_T^i*
  π_T^i.bs $\leftarrow$ st$_T$.puvProtocol.pt
π_T^i.st$_{\text{exe}}$ $\leftarrow$ bindDone
π_T^i.canValidate $\leftarrow$ true
π_T^i.sid $\leftarrow \pi_T^i$.sid || puvProtocol || c || t || false
return (t, false)

Figure F.22: Game 7. Adv$\mathcal{H}_3$ is a wrapper to $\mathcal{H}_3$ that is called by $\mathcal{A}$, so we can keep track of each call and detect if $\mathcal{A}$ ever causes bad.

$$|\ \text{Pr}_6 - \text{Pr}_7\ | \leq \text{Pr}_7[\text{bad}] \tag{F.30}$$

And, therefore:

$$|\ \text{Pr}_6 - \text{Pr}_7\ | \leq \frac{q_{\mathcal{H}_3}\ (q_{\text{E}} + q_{\text{Send-T}})}{2^{2\lambda}} \tag{F.31}$$

Game 8

In this game, when $\mathcal{A}$ queries oracle Validate-T to deliver a message M and a tag t to a token session π_T^i that finished Bind, the challenger always rejects, provided M and t were not output by π_T^i's partner (if it has one), the partner was not compromised and the pin associated with π_T^i was not corrupted. In sum, given M and t that would result in $\mathcal{A}$ winning the experiment, the challenger rejects instead, making it impossible for $\mathcal{A}$ to win by forging a message tag and delivering it to any token session. We flag this event by creating a bad event bad_8.

Game8($\mathcal{A}$):
$\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
$\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
$\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
$\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
$\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
$\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
$\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
$\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
$\text{bad}_8 \leftarrow$ false
win-SUF $\leftarrow 0$
$() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
return win-SUF

Token-Win-SUF (T, i, M, t, d):
if $d \neq$ accepted **then return** 1

if $\exists (C_1, j_1), (C_2, j_2)$ *s.t.* $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi_{C_1}^{j_1}.\text{st}_{\text{exe}} = \pi_{C_2}^{j_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi_{C_1}^{j_1}.\text{sid} = \pi_{C_2}^{j_2}.\text{sid}$ **then return** 1

if $\exists (T_1, i_1), (T_2, i_2)$ *s.t.* $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi_{T_1}^{i_1}.\text{st}_{\text{exe}} = \pi_{T_2}^{i_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi_{T_1}^{i_1}.\text{sid} = \pi_{T_2}^{i_2}.\text{sid}$ **then return** 1

$(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
if $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
  if $(C, j) = (\perp, \perp)$ **or** $\pi_C^j.\text{compromised} = \text{false}$ **then**
    if $\pi_T^i.\text{pinCorr} = \text{false}$ **then**
      return $1 \wedge \text{bad}_8 \neq \text{true}$

return 0

Figure F.23: Game 8 and modified winning conditions function Token-Win-SUF.

Auth-C (C, j, M):
if $\pi_C^j = \perp$ **or** $\pi_C^j.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi_C^j.\text{isValid} = \text{false}$ **then**
  return $\perp$
$(M, t) \xleftarrow{\$}$ auth-C(π_C^j, M)
$\mathcal{L}_{\text{authC}} \leftarrow \mathcal{L}_{\text{authC}} \cup \{(C, j, M, t)\}$
return (M, t)

Validate-T (T, i, M, t, d):
if $\pi_T^i = \perp$ **or** $\pi_T^i.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi_T^i.\text{isValid} = \text{false}$ **then**
  return $\perp$
$(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
status $\leftarrow$ validate-T(π_T^i, M, t, d)
if $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
  if $(C, j) = (\perp, \perp)$ **or** $\pi_C^j.\text{compromised} = \text{false}$ **then**
    if $\pi_T^i.\text{pinCorr} = \text{false}$ **then**
      if status $=$ accepted **then**
        $\text{bad}_8 \leftarrow$ true
        win-SUF $\leftarrow$ Token-Win-SUF(T, i, M, t, d)
      return rejected
if status $=$ accepted **then**
  win-SUF $\leftarrow$ Token-Win-SUF(T, i, M, t, d)
return status

Figure F.24: Modified Auth-C and Validate-T oracles - Game 8.

Notice that, after Game 4, every token session that finishes Bind has *at most* one partner. Indeed, for two sessions π_C^j and π_T^i to be partnered, they must have finished Bind and have matching sid values ($\pi_C^j.\text{sid} = \pi_T^i.\text{sid}$). The sCTAP 2.1+ sid, after Bind is completed, is a concatenation of all the exchanged messages, including the PAKE sid. From Game

4 onward, it is impossible for any client or token session to finish Bind and have more than one PAKE partner, because $\mathcal{A}$ loses if the explicit mutual authentication guarantee is broken. Therefore, we have a guarantee that if a token session π_T^i has a partner π_C^j, then π_C^j is its *unique* partner.

It is also always the case that every token session where bad_8 can happen must have finished Bind passively, through an Execute query or a passive Send-Bind-T query. Indeed, after Game 4, $\mathcal{A}$ can no longer actively attack a token session π_T^i via the Send-Bind-T oracle without corrupting the pin. However, if $\mathcal{A}$ does corrupt the pin, then it can never cause bad_8, i.e., can never win the game via a valid forgery against any session of token T.

Therefore, *only* passive executions of Bind can create token sessions where bad_8 can happen.

Reduction to the SUF-CMA security of MAC

To bound $\mathcal{A}$'s advantage against Game 8, we consider an adversary $\mathcal{B}_8$ simulating Game 8 and fix a value n_{TS} for the maximum amount of token sessions that finished Bind passively. Then, $\mathcal{B}_8$ guesses at random in which token session bad_8 will happen first. We refer to that target session as $\pi_{T,\mathsf{target}}^i$.

If $\mathcal{A}$ ever queries Compromise on the client session that is partnered with $\pi_{T,\mathsf{target}}^i$, then the simulation aborts, since it is now impossible to cause bad_8 on the target session $\pi_{T,\mathsf{target}}^i$ (recall that if $\pi_C^j.\mathsf{compromised} = \mathsf{true}$, then $\mathcal{A}$ can never win via a valid forgery against its partner token session).

Whenever $\mathcal{A}$ queries the Auth-C oracle with a message M such that π_C^j is the partner of the target session $\pi_{T,\mathsf{target}}^i$, $\mathcal{B}_8$ calls the $\mathcal{O}_{\mathsf{Auth}}$ oracle to obtain a challenge tag t for M, and returns (M, t) to $\mathcal{A}$.

Whenever $\mathcal{A}$ queries the Validate-T oracle with message M and tag t such that π_T^i is the target session $\pi_{T,\mathsf{target}}^i$, $\mathcal{B}_8$ immediately returns M and t to its challenger and ends the simulation. If π_T^i is not the target session, $\mathcal{B}_8$ continues simulating Game 8 until either $\mathcal{A}$ queries Validate-T for session $\pi_{T,\mathsf{target}}^i$ or bad_8 happens.

It is not hard to see that if bad_8 happens in the target session $\pi_{T,\mathsf{target}}^i$, $\mathcal{B}_8$ always wins the SUF-CMA game by returning the (M, t) given to $\pi_{T,\mathsf{target}}^i$ by $\mathcal{A}$ in a Validate-T query.

$\mathcal{B}_8(\mathcal{A}, \mathcal{O}_{\text{Auth}}, \mathcal{O}_{\text{Ver}})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: $\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
7: $\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
8: $\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
9: $\text{bad}_8 \leftarrow \text{false}$
10: $\text{guess} \xleftarrow{\$} \{0, ..., n_{\text{TS}} - 1\}$
11: $\mathcal{L}_{\text{TS}} \leftarrow \emptyset, (M_{\text{chal}}, t_{\text{chal}}) \leftarrow (\perp, \perp)$
12: win-SUF $\leftarrow 0$
13: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^{\lambda})$
14: **return** $(M_{\text{chal}}, t_{\text{chal}})$

Figure F.25: $\mathcal{B}_8$'s simulation of Game 8.

Auth-C (C, j, M):
1: **if** $\pi^j_C = \perp$ **or** $\pi^j_C.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi^j_C.\text{isValid} = \text{false}$ **then**
2: **return** $\perp$
3: $\pi^i_{T,\text{target}} \leftarrow \mathcal{L}_{\text{TS}}[\text{guess} - 1]$
4: **if** $\pi^i_{T,\text{target}} \neq \perp \wedge (C, j) = \text{tokenBindPartner}(T, i)$ **then**
5: $(M, t) \leftarrow \mathcal{O}_{\text{Auth}}(M)$
6: **else**
7: $(M, t) \xleftarrow{\$}$ auth-C(π^j_C, M)
8: $\mathcal{L}_{\text{authC}} \leftarrow \mathcal{L}_{\text{authC}} \cup \{(C, j, M, t)\}$
9: **return** (M, t)

Compromise (C, j):
1: **if** $\pi^j_C = \perp$ **or** $\pi^j_C.\text{st}_{\text{exe}} \neq \text{bindDone}$ **then**
2: **return** $\perp$
3: $\pi^i_{T,\text{target}} \leftarrow \mathcal{L}_{\text{TS}}[\text{guess} - 1]$
4: **if** $\pi^i_{T,\text{target}} \neq \perp \wedge (C, j) = \text{tokenBindPartner}(T, i)$ **then**
5: abort
6: $\pi^j_C.\text{compromised} = \text{True}$
7: **return** $\pi^j_C.\text{bs}$

Validate-T (T, i, M, t, d):
1: **if** $\pi^i_T = \perp$ **or** $\pi^i_T.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi^i_T.\text{isValid} = \text{false}$ **then**
2: **return** $\perp$
3: $(C, j) \leftarrow \text{tokenBindPartner}(T, i)$
4: **if** $\pi^i_T = \mathcal{L}_{\text{TS}}[\text{guess} - 1]$ **then**
5: $(M_{\text{chal}}, t_{\text{chal}}) \leftarrow (M, t)$
6: return
7: **else**
8: status $\leftarrow$ validate-T(π^i_T, M, t, d)
9: **if** $(C, j, M, t) \notin \mathcal{L}_{\text{authC}}$ **then**
10: **if** $(C, j) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
11: **if** $\pi^i_T.\text{pinCorr} = \text{false}$ **then**
12: **if** status $=$ accepted **then**
13: $\text{bad}_8 \leftarrow \text{true}$
14: $(M_{\text{chal}}, t_{\text{chal}}) \leftarrow (M, t)$
15: return
16: **return** rejected
17: **if** status $=$ accepted **then**
18: win-SUF $\leftarrow$ Token-Win-SUF(T, i, M, t, d)
19: **return** status

obtainSessionKey-T-end$(\pi^i_T, \text{puvProtocol}, c)$:
1: **if** $\text{puvProtocol} \notin \text{st}_T.\text{puvProtocolList} \vee \text{st}_T.\text{pinRetries} = 0$ **then**
2: **return** $(\perp, \text{false})$
3: $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetries} - 1$
4: $(K, t) \leftarrow \text{st}_T.\text{puvProtocol.decapsulate}(c)$
5: **if** $(K, t) = \perp$ **then**
6: $\text{st}_T.\text{puvProtocol.initialize-T}()$
7: $\text{st}_T.m \leftarrow \text{st}_T.m - 1$
8: **if** $\text{st}_T.m = 0$ **then**
9: authPowerUp-T(st_T)
10: **return** $(\perp, \text{true})$
11: **else**
12: **return** $(\perp, \text{false})$
13: $\mathcal{L}_{\text{tTrace}}[\pi^i_T] \leftarrow \mathcal{L}_{\text{tTrace}}[\pi^i_T] \,||\, c$
14: **if** $|\text{trace} \in \mathcal{L}_{\text{tTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi^i_T]| > 1$ **then**
15: abort
16: **if** $\pi^i_T.\text{pinCorr} = \text{false} \wedge |\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi^i_T]| = 0$ **then**
17: abort
18: **if** $|\text{trace} \in \mathcal{L}_{\text{cTrace}} : \text{trace} = \mathcal{L}_{\text{tTrace}}[\pi^i_T]| > 1$ **then**
19: abort
20: $\text{st}_T.m \leftarrow 3$
21: $\text{st}_T.\text{pinRetries} \leftarrow \text{pinRetriesMax}$
22: **for all** puvProtocol' $\in \text{st}_T.\text{puvProtocolList}$ **do**
23: $\text{st}_T.\text{puvProtocol'.resetPuvToken}()$
24: **if** $\mathcal{L}_{\text{passive}}[_, \pi^i_T] = \text{true} \vee \pi^i_T.\text{pinCorr} = \text{false}$ **then**
25: $\tilde{K} \xleftarrow{\$} \{0,1\}^{2\lambda}$
26: $\mathcal{L}_{\text{K}}[t] \leftarrow \tilde{K}$
27: $K \leftarrow \tilde{K}$
28: $\pi^i_T.\text{bs} \leftarrow K$
29: **if** $\mathcal{L}_{\mathcal{H}_3}[\pi^i_T.\text{bs}] \neq \perp$ **then**
30: abort
31: $\mathcal{L}_{\text{bs}} \leftarrow \mathcal{L}_{\text{bs}} \cup \{\pi^i_T.\text{bs}\}$
32: $\mathcal{L}_{\text{TS}} \leftarrow \mathcal{L}_{\text{TS}} \cup \{\pi^i_T\}$
33: **else**
34: $\pi^i_T.\text{bs} \leftarrow \text{st}_T.\text{puvProtocol}.pt$
35: $\pi^i_T.\text{st}_{\text{exe}} \leftarrow \text{bindDone}$
36: $\pi^i_T.\text{canValidate} \leftarrow \text{true}$
37: $\pi^i_T.\text{sid} \leftarrow \pi^i_T.\text{sid} \,||\, \text{puvProtocol} \,||\, c \,||\, t \,||\, \text{false}$
38: **return** (t, false)

Figure F.26: Modified Auth-C, Validate-T and Compromise oracles, and function obtainSessionKey-T-end, simulated by $\mathcal{B}_8$.

Advantage measure

The advantage of $\mathcal{B}_8$ against the SUF-CMA security of MAC can be calculated as:

$$\mathsf{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_8) \geq \frac{\Pr_8[\text{bad}_8]}{n_{\text{TS}}} \tag{F.32}$$

$$n_{\text{TS}}\ \mathsf{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_8) \geq \Pr_8[\text{bad}_8] \tag{F.33}$$

Since games 7 and 8 are identical until bad_8, by the difference lemma, we have:

$$\Pr_8[\text{bad}_8] \geq |\ \Pr_7 - \Pr_8\ | \tag{F.34}$$

Thus, we have:

$$n_{\text{TS}}\ \mathsf{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_8) \geq |\ \Pr_7 - \Pr_8\ | \tag{F.35}$$

Since each token session can finish Bind passively only in Execute and Send-Bind-T queries, we know that $n_{\text{TS}} \leq q_{\text{E}} + q_{\text{Send-T}}$. Therefore:

$$(q_{\text{E}} + q_{\text{Send-T}})\ \mathsf{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_8) \geq |\ \Pr_7 - \Pr_8\ | \tag{F.36}$$

Game 9

In this game, when $\mathcal{A}$ queries oracle Validate-C to deliver a message M and a tag t to a client session π^j_C that finished Bind, the challenger always rejects, provided M and t were not output by π^j_C's partner, π^j_C was not compromised and the pin was not corrupted. In sum, given M and t that would result in $\mathcal{A}$ winning the experiment, the challenger rejects instead, making it impossible for $\mathcal{A}$ to win by forging a message tag and delivering it to any client session. We flag this event by creating a bad event bad_9. As in Game 8, it is always the case that *only* passive executions of Bind can create client sessions where bad_9 can occur.

Reduction to the SUF-CMA security of MAC

As was the case in Game 8, we bound the probability of $\mathcal{A}$ winning against Game 9 by reducing to the SUF-CMA security of the underlying MAC.

Game9($\mathcal{A}$):
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: $\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
7: $\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
8: $\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
9: $\text{bad}_8 \leftarrow$ false
10: $\text{bad}_9 \leftarrow$ false
11: win-SUF $\leftarrow 0$
12: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
13: **return** win-SUF

Client-Win-SUF (C, j, M, t):
1:
2: **if** $\exists (C_1, j_1), (C_2, j_2)$ *s.t.* $(C_1, j_1) \neq (C_2, j_2)$ **and** $\pi^{j_1}_{C_1}.\text{st}_{\text{exe}} = \pi^{j_2}_{C_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{j_1}_{C_1}.\text{sid} = \pi^{j_2}_{C_2}.\text{sid}$ **then return** 1
3:
4: **if** $\exists (T_1, i_1), (T_2, i_2)$ *s.t.* $(T_1, i_1) \neq (T_2, i_2)$ **and** $\pi^{i_1}_{T_1}.\text{st}_{\text{exe}} = \pi^{i_2}_{T_2}.\text{st}_{\text{exe}} = \text{bindDone}$ **and** $\pi^{i_1}_{T_1}.\text{sid} = \pi^{i_2}_{T_2}.\text{sid}$ **then return** 1
5:
6: $(T, i) \leftarrow \text{clientBindPartner}(C, j)$
7: **if** $(T, i, M, t) \notin \mathcal{L}_{\text{authT}}$ **then**
8: **if** $(T, i) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
9: **if** $\pi^j_C.\text{pinCorr} = \text{false}$ **then**
10: **return** $1 \wedge \text{bad}_9 \neq \text{true}$
11:
12: **return** 0

Figure F.27: Game 9 and winning conditions function Client-Win-SUF modifications.

Validate-C (C, j, M, t):
1: **if** $\pi^j_C = \perp$ **or** $\pi^j_C.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi^j_C.\text{isValid} = \text{false}$ **then**
2: **return** $\perp$
3: $(T, i) \leftarrow \text{clientBindPartner}(C, j)$
4: status $\leftarrow$ validate-C(π^j_C, M, t)
5: **if** $(T, i, M, t) \notin \mathcal{L}_{\text{authT}}$ **then**
6: **if** $(T, i) = (\perp, \perp)$ **or** $\pi^j_C.\text{compromised} = \text{false}$ **then**
7: **if** $\pi^j_C.\text{pinCorr} = \text{false}$ **then**
8: **if** status $=$ accepted **then**
9: $\text{bad}_9 \leftarrow \text{true}$
10: win-SUF $\leftarrow$ Client-Win-SUF(C, j, M, t)
11: **return** rejected
12: **if** status $=$ accepted **then**
13: win-SUF $\leftarrow$ Client-Win-SUF(C, j, M, t)
14: **return** status

Auth-T (T, i, M):
1: **if** $\pi^i_T = \perp$ **or** $\pi^i_T.\text{st}_{\text{exe}} \neq \text{bindDone}$ **or** $\pi^i_T.\text{isValid} = \text{false}$ **then**
2: **return** $\perp$
3: $(M, t) \xleftarrow{\$}$ auth-T(π^i_T, M)
4: $\mathcal{L}_{\text{authT}} \leftarrow \mathcal{L}_{\text{authT}} \cup \{(T, i, M, t)\}$
5: **return** (M, t)

Figure F.28: Modified Auth-T and Validate-C oracles - Game 9.

We consider an adversary $\mathcal{B}_9$ against the SUF-CMA game that simulates Game 9 and fix a value n_{CS} for the maximum amount of client sessions that finish Bind with a token session. Then, $\mathcal{B}_9$ guesses at random in which client session bad_9 will happen first. We refer to that session as $\pi^j_{C,\text{target}}$.

If $\mathcal{A}$ ever queries Compromise on $\pi^j_{C,\text{target}}$, then the simulation aborts, since it is now impossible to cause bad_9 on the target session $\pi^j_{C,\text{target}}$ (recall that if $\pi^j_C.\text{compromised} = \text{true}$, then $\mathcal{A}$ can never win via a valid forgery against π^j_C).

Whenever $\mathcal{A}$ queries the Auth-T oracle with a message M such that π^i_T is the partner of the target session $\pi^j_{C,\text{target}}$, $\mathcal{B}_9$ calls the $\mathcal{O}_{\text{Auth}}$ oracle to obtain a challenge tag t for M, and returns (M, t) to $\mathcal{A}$.

Whenever $\mathcal{A}$ queries the Validate-C oracle with message M and tag t such that π^j_C is the

target session $\pi^j_{C,\text{target}}$, $\mathcal{B}_9$ immediately returns M and t to its challenger and ends the simulation. If π^j_C is not the target session, $\mathcal{B}_9$ continues simulating Game 9 until either $\mathcal{A}$ queries Validate-C for session $\pi^j_{C,\text{target}}$ or bad_9 happens.

It is not hard to see then, that if bad_9 ever happens in the target session $\pi^j_{C,\text{target}}$, $\mathcal{B}_9$ always wins the SUF-CMA game by returning the (M, t) given to $\pi^j_{C,\text{target}}$ by $\mathcal{A}$ in a Validate-C query.

$\mathcal{B}_9(\mathcal{A}, \mathcal{O}_{\text{Auth}}, \mathcal{O}_{\text{Ver}})$:
1: $\mathcal{L}_{\text{authC}}, \mathcal{L}_{\text{authT}} \leftarrow \emptyset$
2: $\mathcal{L}_{\text{valid}}, \mathcal{L}_{\text{corrupt}} \leftarrow \emptyset$
3: $\mathcal{L}_{\text{sCDH}}, \mathcal{L}_{\mathcal{H}_2} \leftarrow \emptyset$
4: $\mathcal{L}_{\text{TokenSetup}} \leftarrow \emptyset$
5: $\mathcal{L}_{\text{pin}} \leftarrow \emptyset$
6: $\mathcal{L}_{\text{cTrace}} \leftarrow \emptyset, \mathcal{L}_{\text{tTrace}} \leftarrow \emptyset$
7: $\mathcal{L}_{\text{passive}} \leftarrow \emptyset, \mathcal{L}_{\text{K}} \leftarrow \emptyset$
8: $\mathcal{L}_{\text{bs}}, \mathcal{L}_{\mathcal{H}_3} \leftarrow \emptyset$
9: $\text{bad}_8 \leftarrow \text{false}$
10: $\text{bad}_9 \leftarrow \text{false}$
11: $\text{guess} \xleftarrow{\$} \{0, \ldots, n_{\text{CS}} - 1\}$
12: $\mathcal{L}_{\text{CS}} \leftarrow \emptyset, (M_{\text{chal}}, t_{\text{chal}}) \leftarrow (\perp, \perp)$
13: $\text{win-SUF} \leftarrow 0$
14: $() \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(1^\lambda)$
15: **return** $(M_{\text{chal}}, t_{\text{chal}})$

Figure F.29: $\mathcal{B}_9$'s simulation of Game 9.

Advantage measure

The advantage of $\mathcal{B}_9$ against the SUF-CMA security of MAC is upper bounded by:

$$\text{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_9) \geq \frac{\Pr_9[\text{bad}_9]}{n_{\text{CS}}} \tag{F.37}$$

$$n_{\text{CS}} \, \text{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_9) \geq \Pr_9[\text{bad}_9] \tag{F.38}$$

Since games 8 and 9 are identical until bad_9 happens, by the difference lemma, we have:

$$\Pr_9[\text{bad}_9] \geq | \Pr_8 - \Pr_9 | \tag{F.39}$$

Thus, we have:

$$n_{\text{CS}} \, \text{Adv}^{\text{SUF-CMA}}_{\text{MAC}}(\mathcal{B}_9) \geq | \Pr_8 - \Pr_9 | \tag{F.40}$$

Validate-C (C, j, M, t):

if $\pi_C^j = \perp$ **or** $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **or** $\pi_C^j.\mathsf{isValid} = \mathsf{false}$ **then**
  return $\perp$
$(T, i) \leftarrow \mathsf{clientBindPartner}(C, j)$
if $\pi_C^j = \mathcal{L}_{\mathsf{CS}}[\mathsf{guess} - 1]$ **then**
  $(M_{\mathsf{chal}}, t_{\mathsf{chal}}) \leftarrow (M, t)$
  return
else
  $\mathsf{status} \leftarrow \text{validate-C}(\pi_C^j, M, t)$
if $(T, i, M, t) \notin \mathcal{L}_{\mathsf{authT}}$ **then**
  if $(T, i) = (\perp, \perp)$ **or** $\pi_C^j.\mathsf{compromised} = \mathsf{false}$ **then**
    if $\pi_C^j.\mathsf{pinCorr} = \mathsf{false}$ **then**
      if $\mathsf{status} = \mathsf{accepted}$ **then**
        $\mathsf{bad}_9 \leftarrow \mathsf{true}$
        $(M_{\mathsf{chal}}, t_{\mathsf{chal}}) \leftarrow (M, t)$
        return
      return rejected
if $\mathsf{status} = \mathsf{accepted}$ **then**
  win-SUF $\leftarrow$ Client-Win-SUF(C, j, M, t)
return status

Auth-T (T, i, M):

if $\pi_T^i = \perp$ **or** $\pi_T^i.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **or** $\pi_T^i.\mathsf{isValid} = \mathsf{false}$ **then**
  return $\perp$
$\pi_{C,\mathsf{target}}^j \leftarrow \mathcal{L}_{\mathsf{CS}}[\mathsf{guess} - 1]$
if $\pi_{C,\mathsf{target}}^j \neq \perp \wedge (T, i) = \mathsf{clientBindPartner}(C, j)$ **then**
  $(M, t) \leftarrow \mathcal{O}_{\mathsf{Auth}}(M)$
else
  $(M, t) \xleftarrow{\$} \text{auth-T}(\pi_T^i, M)$
$\mathcal{L}_{\mathsf{authT}} \leftarrow \mathcal{L}_{\mathsf{authT}} \cup \{(T, i, M, t)\}$
return (M, t)

obtainSessionKey-C-end(π_C^j, t):

if $\pi_C^j.\mathsf{st}_{\mathsf{exe}} = \mathsf{bindError}$ **then**
  return $\perp$
$(K, \mathsf{valid}) \leftarrow \pi_C^j.\mathsf{puvProtocol.finalizeBind}(t)$
if $\mathsf{valid} = \mathsf{false}$ **then**
  $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{bindError}$
  return $\perp$
if $\mathcal{L}_{\mathsf{K}}[t] \neq \perp$ **then**
  $K \leftarrow \mathcal{L}_{\mathsf{K}}[t]$
  if $\mathcal{L}_{\mathcal{H}_3}[K] \neq \perp$ **then**
    abort
  $\mathcal{L}_{\mathsf{bs}} \leftarrow \mathcal{L}_{\mathsf{bs}} \cup \{K\}$
  $\mathcal{L}_{\mathsf{CS}} \leftarrow \mathcal{L}_{\mathsf{CS}} \cup \{\pi_C^j\}$
if $|\ \mathsf{trace} \in \mathcal{L}_{\mathsf{cTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{cTrace}}[\pi_C^j]\ | > 1$ **then**
  abort
if $\pi_C^j.\mathsf{pinCorr} = \mathsf{false} \wedge |\ \mathsf{trace} \in \mathcal{L}_{\mathsf{tTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{cTrace}}[\pi_C^j]\ | = 0$ **then**
  abort
if $|\ \mathsf{trace} \in \mathcal{L}_{\mathsf{tTrace}} : \mathsf{trace} = \mathcal{L}_{\mathsf{cTrace}}[\pi_C^j]\ | > 1$ **then**
  abort
$\pi_C^j.\mathsf{bs} \leftarrow K$
$\pi_C^j.\mathsf{st}_{\mathsf{exe}} \leftarrow \mathsf{bindDone}$
$\pi_C^j.\mathsf{sid} \leftarrow \pi_C^j.\mathsf{sid} \,||\, t \,||\, \mathsf{false}$
return accepted

Compromise (C, j):

if $\pi_C^j = \perp$ **or** $\pi_C^j.\mathsf{st}_{\mathsf{exe}} \neq \mathsf{bindDone}$ **then**
  return $\perp$
if $\pi_C^j = \mathcal{L}_{\mathsf{CS}}[\mathsf{guess} - 1]$ **then**
  abort
$\pi_C^j.\mathsf{compromised} = \mathsf{True}$
return $\pi_C^j.\mathsf{bs}$

Figure F.30: Modified Auth-T, Validate-C and Compromise oracles, and function obtainSessionKey-C-end, simulated by $\mathcal{B}_9$.

Since each client session can finish Bind passively in Execute and Send-Bind-C queries, we know that $n_{\mathsf{CS}} \leq q_{\mathsf{E}} + q_{\mathsf{Send\text{-}C}}$. Therefore:

$$(q_{\mathsf{E}} + q_{\mathsf{Send\text{-}C}})\ \mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_9) \geq |\ \Pr_8 - \Pr_9\ | \tag{F.41}$$

Final Analysis

Note that, at this point, $\mathcal{A}$ cannot win against Game 9, since it must be able to win via one of the four conditions in Token-Win-SUF or one of the three conditions in Client-Win-SUF.

Win via Token-Win-SUF

Condition 1 of Token-Win-SUF ($d = 0$) is always false, since Token-Win-SUF is only called after a validation from the Validate-T oracle is successful, which is dependant on the user decision bit d.

Conditions 2 and 3 of Token-Win-SUF will also never be true. Indeed, since collisions between both client and token sid values have been ruled out in Game 6 (because collisions between PAKE sid values have been ruled out, which are part of the client and token sid), it can never be the case that two different client or token sessions have the same sid. Thus, these conditions are always false.

Condition 4 of Token-Win-SUF is also always false. Recall that, for $\mathcal{A}$ to win, it must give a pair (M, t) to the Validate-T oracle for a token session π_T^i and (M, t) must pass validation. Additionally, π_T^i must not have had its pin corrupted, and either not have a partner or have a unique partner that was not compromised. In the latter case, (M, t) must also not have been output by that partner.

Indeed, if the target π_T^i which is validating (M, t) from $\mathcal{A}$ has a (unique) partner π_C^j, then $\mathcal{A}$ hasn't actively attacked π_T^i during the Bind session, and therefore has no information whatsoever about the binding state, which is sampled uniformly at random since Game 5.

If $\mathcal{A}$ actively attacked π_T^i by performing a Bind session itself, the experiment always rejects (through breaking the ma security of PAKE), provided the pin was not corrupted. If it was corrupted, then $\mathcal{A}$ can never win against any session of that token. This eliminates the attack where $\mathcal{A}$ simply guesses the pin. This means $\mathcal{A}$ has no information about the pin without corrupting its user U.

Finally, $\mathcal{A}$ can never win by correctly forging a tag t, since the token session always rejects any valid forgery attempt from $\mathcal{A}$.

Win via Client-Win-SUF

Conditions 1 and 2 are identical to conditions 2 and 3 from Token-Win-SUF, so they are always false.

Condition 3, similarly to condition 4 of Token-Win-SUF, is also false, since $\mathcal{A}$ can never learn anything about the binding state used to authenticate messages to any client session without corrupting a pin or compromising the session itself, and can never win by actually submitting a valid forged tag t because the client session always rejects such an attempt.

Advantage measure

Given that $\mathcal{A}$ can no longer win against Game 9, we have:

$$\mathsf{Pr}_9 = 0 \tag{F.42}$$

Note that the running times of all adversaries are close to that of $\mathcal{A}$.

This completes the proof.