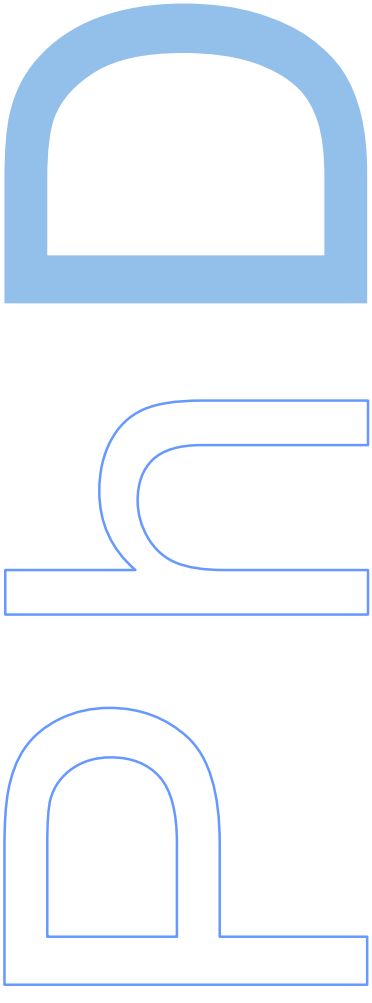
A Provable Security Treatment of Isolated Execution Environments and Applications to Secure Computation

Bernardo Portela
MAPi
DCC
2018

Orientador
Manuel Barbosa, Professor Auxiliar, FCUP



Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: Bernardo Portela DATE: 13 October 2017

Dedication and acknowledgements

First and foremost, I would like to express my dearest appreciation for the constant support and guidance of my advisor Dr. Manuel Bernardo Barbosa. Not only was he the one responsible for making me want to pursue research in Cryptography, but he was also the most significant factor towards the resolve I presently have to give the best of myself at all times. I truly hope this Ph.D. is just the beginning of a long career of collaboration.

I would also like to thank my all my co-authors, especially Bogdan Warinschi and Guillaume Scerri. They provided crucial guidance to my work in its embryonic stages, and helped make what seemed like a nice research challenge into the contribution that is now my thesis.

I would like to thank my loving parents and girlfriend for doing their absolute best to inspire me, and to ensure that I was always confident and happy. They made me the person I am today, and I am certain that without that love, this thesis would have never been written.

Furthermore, a big thanks to my research team. They have been with me at all times along the way, and have been massively supportive throughout all the work on this Ph.D. I hope to count on them from here onwards, for both research and drinks.

Last, but most certainly not least, to all of my friends with whom I have cherished great memories throughout this Ph.D. They are living proofs that there is no subject immune from mockery, and boy am I thrilled to celebrate this chapter of my life with them.

Abstract

Exciting new capabilities of modern trusted hardware technologies allow for the execution of arbitrary code within environments isolated from the rest of the system and provide cryptographic mechanisms for securely reporting on these executions to remote parties. Rigorously proving security of protocols that rely on this type of hardware raises two challenges. The first is to develop scalable new proof techniques that permit dealing with new forms of composition of cryptographic primitives that arise in this setting. In particular, dealing with implementations of cryptographic primitives provided explicitly to an attacker requires extending the set of proof techniques commonly used for security analysis.

We present the first steps in this direction by providing formal security definitions, generic constructions and security analysis for *attested computation*, *labelled attested computation* and *key-exchange for attestation*. The approach taken is incremental: each of the concepts relies on the previous ones according to an approach that is modular. These building blocks are then demonstrated to be applicable in a variety of settings for security-critical applications, namely *secure outsourced computation*, *secure function evaluation* and *multiparty computation*.

Our constructions formalise the intuition of using modern hardware to play the role of a trusted third party. The attestation guarantees of trusted hardware technologies can be used to bootstrap secure communications between participants and the trusted third party. The load of communications and computations on participants only depends on the size of each party's inputs and outputs, and is thus small and independent from the intricacies of the functionality to be computed. The remaining computational load (essentially that of the functionality being computed) is moved to an untrusted party that is running this trusted hardware machine, an attractive feature for Cloud-based scenarios. Finally, an experimental evaluation of the proposed solution on SGX-enabled hardware is presented. The implementation is open-source and functionality agnostic: it can be used to securely outsource to the Cloud arbitrary off-the-shelf collaborative software, enabling secure collaborative execution over private inputs provided by multiple parties.

Resumo

Desenvolvimentos recentes em hardware confiável permitem a execução de software em ambientes isolados do resto do sistema, e fornecem mecanismos criptográficos para validar resultados perante participantes remotos. Demonstrar rigorosamente segurança em protocolos que dependem deste tipo de tecnologias levanta dois desafios de investigação. O primeiro está no desenvolvimento de modelos de segurança que corretamente capturem as assunções e relações de confiança neste novo cenário. O segundo está no desenvolvimento de técnicas de prova escaláveis que permitam lidar com novas formas de composição de primitivas criptográficas existentes neste ambiente. Mais especificamente, a necessidade de capturar um modelo onde implementações de primitivas criptográficas são providenciadas diretamente ao adversário faz com que seja ncessário extender as técnicas clássicas de prova que são habitualmente utilizadas para realizar análises de segurança.

Este trabalho representa os primeiros passos nesta direção, apresentando definições formais, construções genéricas e respetiva análise de segurança de *attested computation*, *labelled attested computation* e *key-exchange for attestation*. A abordagem é incremental: cada um destes conceitos faz uso dos anteriores de uma forma modular. A aplicabilidade destas construções é demonstrada num conjunto de cenários que permitem execução remota de código para computação segura sobre dados privados.

As construções formalizam a intuição de usar ambientes isolados como participantes confiáveis. As garantias de atestação conseguidas por estes hardwares podem ser utilizadas para iniciar comunicações seguras entre participantes e ambientes isolados. O peso de comunicação e computação dos participantes só depende do tamanho do input/output de cada um, sendo como tal reduzido e independente da complexidade da funcionalidade a ser computada. O restante peso computacional (essencialmente, o da função a executar) encontra-se do lado do participante não-confiável equipado com hardware seguro, o que é um cenário atrativo quando se trata de aplicações baseadas em Cloud. Por fim, é apresentada uma avaliação da solução desenvolvida em hardware SGX. A implementação é open-source, e é agnóstica a nível de funcionalidade: pode ser usada para executar bibliotecas existentes de forma segura na cloud, permitindo o processamento de dados privados vindos de múltiplos participantes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Technological developments in recent years have been gradually improving the processing and storage capabilities of a wide variety of computational devices. This facilitates the deployment of efficient security-critical services (such as Internet banking, electronic voting, outsourced data management and computation) over commodity hardware, but in increasingly difficult to control environments. These applications routinely manipulate sensitive data and must ensure i. that attackers cannot tamper with their execution; and ii. that no sensitive information is leaked. Yet, satisfying these guarantees on modern execution platforms rife with vulnerabilities (e.g. in mobile devices, PCs) or inherently not trustworthy (e.g., cloud infrastructures) is a major challenge.

A promising starting point for solutions is the usage of *remote attestation* capabilities offered by modern trusted hardware: computational platforms equipped with this technology can guarantee to a remote party various degrees of integrity for the software that it runs. For instance, the Trusted Platform Module (TPM) can provide certified measurements on the state of the platform and can be used to guarantee integrity of BIOS and boot code right before it is executed. More recent technologies (e.g., ARM's TrustZone and Intel's Software Guard Extensions (SGX)) have significantly expanded the scope and guarantees of trusted hardware. They offer the ability to run applications in "clean-slate" isolated execution environments (IEE) with isolation guarantees from anything else running on the processor; the desired attestation guarantees come from reports that are authenticated by the hardware, without any possibility of manipulation by co-located software, including the operative system.

A second major challenge within this scope is to provide security guarantees that go beyond heuristic arguments. Here, the established methodology is the "provable security" approach, which advocates carrying out the analysis of systems with respect to rigorously specified models that clarify the trust relations, the powers of the adversary and what constitutes a security breach. The approach offers well-established definitional paradigms for all basic

primitives and some of the more used protocols. On the other hand, the success of applying this approach to new and more complex scenarios fundamentally hinges on one's ability to tame scalability problems, as models and proofs, even for moderate size systems, tend to be unwieldy. Some solutions to this issue exist in the form of compositional principles that, when incorporated as a native feature in the security abstractions, allow to establish the guarantees of larger systems from the guarantees on its components [Can01].

In this thesis we take a provable security approach to protocols that rely on the IEE-capabilities of modern trusted hardware. It may be tempting to assume that, for the analysis of such protocols, designing security models is a simple matter of overlaying/merging the trust model induced by the use of such hardware over well-defined security abstractions. If this were true, one could rely on established models and methodologies to perform the analysis. Unfortunately, this is not the case.

Consider the problem of securely outsourcing the computation of a program P to a remote machine. The owner of P wants to ensure that the (potentially malicious) remote platform does not tamper with the execution of the program, and that it learns no information about the input/output (I/O) behaviour of P. For remote machines with IEE capabilities, the following straightforward design has been informally proposed in the literature [HLP+13] and should, intuitively, provide the desired guarantees. First, execute a key-exchange with a remote instance of an IEE; after the key exchange finishes, use the key thus established to send encrypted inputs to the IEE, which can then decrypt and pass them to P. The output returned by P is encrypted within the IEE and sent back to the user. The construction relies on standard building blocks (key-exchange, authenticated encryption) so the security of the overall design should be reducible to that of the key exchange and authenticated encryption, for which one already has widely-accepted security models and constructions.

However, the following two important issues demonstrate that neither existent models nor existent techniques are immediately suitable for the analysis of IEE-based protocols in general, and for the protocol above in particular. The first is the concept of a *party* which is a key notion in specifying and reasoning about the security of distributed systems. Traditionally, one considers security (e.g., of a key-exchange) in a setting where there is a PKI and at least some of the parties (e.g., the servers) have associated public keys. Parties and their cryptographic material are then essentially the same thing for the purpose of the security analysis. Contrarily, in the context of secure outsourcing using trusted hardware, there is no reason why one should impose users to have long term keys; furthermore, privacy considerations require that the cryptographic operations performed by the trusted hardware on remote machines should not allow one to track different instances – which makes long term cryptographic material inadequate as a technical anchor of a party's participation in such a protocol. Indeed, the desirable functionality for many usages of such systems is that the cryptographic material associated with a computation outsourcing protocol (both for local

and remote *parties*) can be arbitrary and fixed on-the-fly, when the protocol is executed. An interesting problem is, therefore, how to define the security of applications such as outsourced computation in this setting, and how to rely on the asymmetry afforded by the trust model specific to IEE systems to realise them.

The second issue is *composability*. In outsourced computation, one might be led to think that security simply follows if the key-exchange is secure and the channel between the user and the IEE uses authenticated encryption (with appropriate replay protection): if the only information passed from the key exchange to the channel is the encryption key, then one can design and analyze the two parts separately. While in more standard scenarios this may be true, reliance on the IEE breaks the independence assumption that allows for composability results: what the specification above hides is that the code run by the IEE (i.e., the program for key-exchange, the one for the secure channels and the program that they protect) needs to be loaded at once, or else no isolation guarantees are given by the trusted hardware.[1] This means that the execution of the different parts of the program is not necessarily independent, as they unavoidably share the state of the IEE. An important question is, therefore, whether the above intuitive construction is sound, and under which conditions can one use it to perform IEE-enabled outsourcing of computation.

To summarise, protocols relying on IEE are likely to be deployed in applications with stringent security requirements, so ensuring that they fall under the scope of the provable security approach is important. Moreover, such applications are likely to be complex: inherently, they involve communication between remote parties, incorporate diverse code executed at different levels of trust, and rely on multiple cryptographic primitives and protocols as building blocks (see, for example, the One-time password protocol based on SGX [HLP+13]). However, key aspects of existing cryptographic models do not naturally translate to this new setting and, perhaps more worryingly, the type of compositional reasoning enabled by such cryptographic models is clearly unsuitable for protocols that rely on IEEs. These two questions help motivate the work in this thesis.

[1] Intuitively, mechanisms such as SGX and TrustZone are designed to protect and provide attestation guarantees over monolithic pieces of software, which must be fixed when an IEE is created and are identified using a fingerprint of the code. To go around this restriction and compose multiple programs, we have two options: i. to build a single composed program and load it in its entirety into an IEE (the approach taken throughout the thesis); or ii. to use multiple IEEs to host the various programs, and employ cryptographic protocols to protect the interactions between them using the (potentially malicious) host operating system as a communications channel. This latter option would again lead to loading composed programs into each IEE, since cryptographic code would need to be added to each of the individual programs, and this would then lead to the same problem tackled in this work.

1.1 Contributions

Consider a setting where a Client wishes to have some program be securely executed in a Server, which is equipped with the aforementioned IEE-enabling hardware. The client is trusted, and the IEE-enabling hardware must behave according to the specified isolation and attestation mechanisms, however no guarantees are ensured regarding the communication channel to the Server, or the Server itself. Indeed, despite the trusted hardware behaving according to the IEE description, the adversary should be able to freely execute arbitrary code within it.

In order to enable confidentiality and integrity guarantees to the program running within the untrusted Server, the Client must rely on attestation guarantees provided by the trusted hardware. Moreover, the Client and Server expect the overhead incurred by relying on a cryptographic protocol supported by trusted hardware to be minimal and, in particular, for the overall cost to closely resemble the computational load of the outsourced computation without security mechanisms.

Suppose the program to be outsourced is denoted P. The intuitive solution is to construct a program P^*, containing the original P and all the additional procedures required to run P^* remotely and securely. As such, program P will need to be transformed (*compiled* in the cryptographic sense of the word) into another program that, intuitively, will result from the composition of a handshake/bootstrapping procedure that will establish a secure channel with the IEE in which the program will be executed, and an instrumented version of P, that uses the aforementioned secure channel to ensure that is indeed securely executed.

This intricacy hints towards the complexity challenges that arise when one wishes to enforce correctness and security properties in composed programs running within IEEs, especially when the goal is to allow for any program P to be combined with a specific bootstrapping procedure (necessary to claim that the client can securely compute any function on the Server). Furthermore, while having an attested bootstrapping procedure for a single secure channel might suffice for the scenario provided, more complex settings will consider several local users agreeing upon program P, and will thus require for this bootstrapping procedure to consider multiple secure channels established to the same IEE, established in a potentially asynchronous way.

The approach taken to formalising and realising such composition patterns has four main stepping stones:

- The specification of a computational model for IEE-enabled machines. This includes a formalisation of programs to be executed within these environments, a specification of what is a program *activation*, what flavours of program composition are considered, and a definition of the machine API, i.e. the interface made available between the IEE

machine and its holder.

- The formalisation of raw guarantees provided by IEEs with cryptographic functionalities. Two approaches are proposed in this sense. *Attested computation* formalises the proposed application example considering a restricted model for program composition. *Labelled attested computation* generalises the same notions for attested/non-attested traces considering more flexible composition and attestation settings.

- A generic transformation from a passively secure key exchange combined with a scheme for attested computation to the bootstrapping procedure referred above. This encompasses both the proposal of how key exchange can be constructed in a setting where attested computation is performed, as well as several utility theorems describing the guarantees obtained when attestation protocols are combined with this flavour of key exchange.

- New efficient, authenticated and private remote attested computation applications relying on these composition-aware formalisations. More specifically, we propose protocols for *Secure Outsourced Computation*, following the provided example where a single user wishes to outsource the execution of program Q to an IEE, *Secure Function Evaluation*, where multiple users want to run program Q over private inputs and retrieve respective outputs, and *Secure Multiparty Computation*, where multiple users have a (potentially reactive) functionality Q responding to private inputs and producing private outputs.

Details follow.

Isolated execution environments

Our first contribution is the proposal of a computational model for IEE-enabling machines. This specification introduces an abstraction of what represents an IEE machine, and includes several core definitions to rigorously analyse the execution of programs in these environments. In this respect, we present the IEEs as idealised random access machines, whose behaviour is fully characterised by a strictly defined interface allowing for program loading and input passing (also referred to as program *activation*). We also require for the I/O behaviour of programs within IEEs to only depend on inputs and on the currently executing program, which essentially enforces isolation guarantees between concurring programs within the same IEE, and for the only information revealed about the execution of programs within IEEs to be contained in its I/O behaviour, preventing machine users to extract cryptographic material held on the secure memory of the IEE.

More specifically, we consider two flavours of programs to be run within IEEs. Non-labelled programs map to a simple execution model where programs are activated with inputs and produce outputs. Composition of non-labelled programs considers the sequential execution

of two programs, and captures a deployment where the first program is the "attested part", the bootstrapping procedure for which the execution should provide attested outputs, and the second program is the "unattested part", the function to be executed, instrumented to use the previously established secure channel. Labelled programs map to a more general execution setting where programs are activated with inputs and labels. These labels can be associated with different users and enable for more flexibility in deployment, as an arbitrary number of programs can be composed in parallel or sequentially, with potentially several stages of attestation.

It is emphasized that the proposed notion of machines is intended to be inclusive of any hardware platform that supports some form of isolated execution. For this reason, the proposed syntax is minimalistic, so that it can be restricted/extended to capture the specific guarantees awarded by different concrete hardware architectures, including TPM, TrustZone, SGX, etc. As an example, the described "vanilla" machine supports an arbitrary number of IEEs, where programs can be loaded only once, and where multiple input/output interactions are allowed with the protected code. This is a close match to the SGX/TrustZone functionalities. However, for something like TPM, one could consider a restricted machine where a limited number of IEEs exist, with constrained input/output capabilities, and running specific code (e.g., to provide key storage). Similarly, the abstraction considers IEE environments where the underlying hardware is assumed to only keep *benevolent* state, i.e., state that cannot be used to introduce destructive correlations between multiple interactions with an IEE. Again, this closely matches what happens in SGX/TrustZone, but different types of state keeping could be allowed for scenarios where such correlations are not a problem or where they must be dealt with explicitly.

Attested computation

This abstraction enables a formal treatment of IEE-based remote attested computation. A user wishes to remotely execute a program P and relies on the cryptographic infrastructure available within the IEE to attest that some incarnation P^* of this program is indeed executing within an IEE of a specific remote machine (or group of machines). A general solution to this problem relying on non-labelled programs is provided, in the form of a new cryptographic concept called *attested computation* (AC). Two core guarantees are formalised.

First, it is required that the user's local view of the execution is "as expected", i.e., that the I/O behaviour that is reconstructed locally corresponds to an honest execution of P. The second guarantee is more subtle and requires that such an *execution has actually occurred in an IEE within a specific remote platform*; in other words, the attested computation client is given the assurance that its code is being run in isolation (and displays a given I/O behaviour) within a prescribed remote physical machine (or group of machines) associated with some authenticated public parameters. This latter guarantee is crucial for bootstrapping the secure

outsourcing of code: consider for example P to be a key-exchange protocol, which will be followed by some other program that relies on the derived key. It must be the case that, at the end of the key exchange, the remote state of the key-exchange protocol is protected by an IEE.

The second guarantee required from attested computation follows naturally from the previous observation. If the attested program keeps sensitive information in its internal state (which is not revealed by its I/O behaviour as in the case of a key-exchange protocol) then execution within the remote IEE should safeguard its internal state. If this were not the case, problems would arise again when trying to compose P^* with some program that relies on the security properties of P. Attested computation schemes where the instrumented program P^* might leak more information in its I/O behaviour than P itself are excluded by introducing the notion of *minimal leakage*, which essentially states that the I/O of an attested program does not leak any information beyond what is unavoidably leaked by an honest execution.

Finally, a scheme for attested computation that relies on a remote machine offering a combination of symmetric authentication and digital signatures (a capability similar to what SGX provides) is proposed, and it is shown that such scheme is secure in the defined context.

Labeled attested computation

The previous formalisation considers a somewhat simple setting, where a program consists of an "attested part", the bootstrapping procedure for the establishment of a secure channel, and an "unnatested part", the secure computation. Attestation guarantees are established with respect to the first part of the execution, thus hindering the deployment of applications where multiple (potentially asynchronous) stages of attestation are required. Furthermore, attested computation requires for the full I/O behaviour to be maintained and validated. This scenario aims to capture the interaction between a single party and an IEE, and it is non-trivial to extend these guarantees to the interaction of multiple parties with the same IEE when the goal is to reason about concurrent asynchronous interactions.

To overcome these problems, the notion of *labelled attested computation* (LAC) is introduced, a powerful and clean generalization of its attested computation counterpart. In a nutshell, this notion makes use of labelled programs where parts of the code loaded in an IEE are marked with labels pertaining to users, such that individual users can get attestation guarantees for parts of the code corresponding to specific labels. The gain is that users can now be oblivious of other users' interactions with the IEE, which leads to significantly simpler and more efficient protocols. Nonetheless, the user can sill derive attestation guarantees about the overall execution of the system, since LAC binds each users' local view to the *same code* running within the IEE, and one can use standard cryptographic techniques to leverage this binding in order to obtain *indirect* attestation guarantees as to the honest executions of the

interactions with other users.

To understand the need for this generalization of attested computation, consider a scenario where we now have n Clients that want to jointly execute a program in the Server. This requires the IEE to run a compiled code consisting of n key exchange protocols, followed by the actual program. Employing AC would require for the execution of all key exchanges (the "attested part") to be performed sequentially on a pre-determined order, and for all participants to exchange execution traces, since attesting the trace of the IEE implies attesting its execution with all users.

Using LAC, we assign a different label to the key exchange of each user (say, the identity of the user that should be running that key exchange); LAC then guarantees, for each user, that the steps of its key exchange (and only those) have been executed properly. Attestation only requires the user's own communication with the IEE and is independent of the other exchanges. Furthermore, since the attestation guarantees provided by LAC bind an attested trace to the code running within the IEE, a local user can be sure that the other key exchanges are being executed with specific parties: the code running inside the IEE will contain public cryptographic material that enables the IEE to authenticate all the local parties and hence authenticating the code implicitly authenticates all the parties participating in the protocol.

Syntax and a formal security model for LAC will be provided, and then shown how this primitive can be used to deploy arbitrary (labelled) programs to remote IEEs with flexible attestation guarantees. The proposed provably secure LAC protocol relies on hardware equipped with SGX-like IEEs. The core guarantees are intuitively similar to a translation of attested computation to a labelled setting. Correctness ensures that a local user can accurately reconstruct a partial view of I/O sequence that took place in the remote environment, for an arbitrary set of labels. Security imposes that an adversary with control of the remote machine cannot convince the local user that some arbitrary remote (partial) execution of a program P has occurred, when it has not. Observe that this provides no guarantees with respect to parts of the execution hidden from the client, or outside the scope of some attested label set. LAC schemes must also satisfy the notion of minimal leakage, necessary for the same reasons as previously explained.

It is also expected for LAC to find applications beyond the specific one considered in this work. For example, an application which should be a direct application of LACs is the implementation of a secure bulletin board. Consider an IEE which receives inputs (which can optionally be authenticated) and appends them to an internally maintained bulletin board. Upon request of an arbitrary party, the IEE returns the current content of the IEE. The desired security is obtained by attesting (only) this communication step.

Key-exchange for attested computation

On its own, both flavours of attested computation provide only integrity guarantees: the I/O behavior of the outsourced code is exposed to untrusted code in the remote machine on which it is run. The natural solution to the problem is to establish a secure communication channel with the IEE via a key-exchange protocol. It is unclear, however, how the standard security models for key-exchange protocols map into the attested computation scenarios, and how existing constructions for secure key-exchange fare in the novel scenario studied. Indeed, for efficiency reasons one should use a key exchange protocol that is *just strong enough* to achieve this goal.

To clarify this issue, we also formalize the notion of key-exchange *for* attested computation. The proposed name is intentional: key-exchange protocols as used in this context differ significantly in syntax and security models from their more traditional counterparts. For example, the suggested syntax reflects that the code of the key-exchange is not fixed a-priori: a user can set parameters both for the component to be run locally and for the one to be executed within the IEE. This allows a user to hard-wire a new nonce in the code to be run remotely (or, as in the proposed examples, some cryptographic public key for which it knows the secret key).

As explained above, the notion of *party* in the context of attested computation needs to be different from the one adopted by traditional notions of secure key-exchange. The proposed solution is to rely on the trust model specific to IEE settings: one can assign some arbitrary strings as identifiers for the users of the local machine, and these users are allowed to specify arbitrary strings as identifiers for the remote code (a secure instantiation would require that this identifier corresponds to some cryptographic material possibly generated on the fly as explained above). The execution model and definitions are adapted for key-exchange for the modified syntax and the new notion of communicating parties is established to reflect the expected guarantees: different local and remote sessions agree on each other's identifiers, derive the same key and the key is unknown to the adversary. One crucial aspect of this security model for key-exchange is that it explicitly accounts for the fact that the remote process will be run under attestation guarantees, which maps to a *semi-active* adversarial environment.

This leads to a generic construction that combines a key-exchange protocol that is passively secure and a standard signature scheme to derive a very efficient key-exchange protocol for attestation. This construction is then used for establishing various *Utility theorems*, which specifically state what composition guarantees one gets for an arbitrary program P that is run within a remote IEE and that relies on shared keys established via attestation of a key-exchange protocol satisfying our tailored definition. More concretely, this work proposes *One-to-one AttKE Utility*, capturing the notion of combining non-labelled attested computation with a single key exchange to the IEE, *One-to-many AttKE Utility*, capturing the notion

of combining non-labelled attested computation with several key-exchange protocols with the same IEE, and *Labelled AttKE Utility*, where labelled attested computation is used to asynchronously run several key-exchange protocols with the same IEE, where each participant only has to validate its own trace.

Applications

The final stage of this work tackles several application scenarios and proposes and analyses hardware-based solutions relying on the presented building blocks.

First, we formalise Secure Outsourced Computation, the original motivating use-case for the presented approach. Syntax is provided, as well as two security notions for a secure outsourced computation protocol, one for authenticity and one for the privacy of the I/O of the outsourced program. We then prove that the construction that combines a key-exchange for attested computation with an authenticated symmetric encryption scheme and replay protection gives rise to a scheme for secure outsourced computation. This result is presented as a general formalisation (i.e., not application specific) of the intuition that by relying on more powerful hardware assumption such as those offered by SGX, one can indeed efficiently achieve a well-defined notion of secure outsourced computation that simultaneously offers verifiability *and* privacy. The proof of this result crucially relies on the utility theorem defined for the one-to-one combination of attested computation with key exchange.

Afterwards, one explores the possibility for taking advantage of these security anchors in the setting of Secure Function Evaluation (SFE) and in the setting of general Multiparty Computation (MPC). Indeed, the functionality outlined by IEEs suggests a simple and natural design for these application scenarios: load the functionality to be computed into an IEE, which plays the role of a Trusted Third Party (TTP), and have users provide inputs and receive outputs via secure channels to the IEE. Attestation ensures the authenticity of the computed function, inputs and outputs. The resulting protocol is extremely efficient when compared to existing solutions that do not rely on hardware assumptions. Indeed, the load of communications and computations on protocol participants is small and independent of the intricacies of the functionality that is being computed; it depends only on the size of each party's inputs and outputs. The remaining computational load — essentially that of computing the functionality expressed as a transition function in a standard programming language — is moved to an untrusted party running an IEE-enabled machine. This makes the protocol attractive for Cloud scenarios. Due to its obvious simplicity, variations of the overall idea have been proposed in several practice-oriented works [SCF+15a, GMF+16]. However, at the time of writing, there exists no thorough and rigorous analysis of the security guarantees provided by this solution in the sense of a general approach to MPC.

The intuitive appeal of this refined protocol obscures multiple obstacles in the process of

obtaining a formal security proof, including: i. the lack of private channels between the users and the remote machine; ii. the need to authenticate/agree on a computation in a setting where communication between parties is inherently asynchronous and only mediated by the IEE; iii. the need to ensure that the "right" parties are engaged in the computation; iv. dealing with the interaction between different parts of the code that coexist within the same IEE, sharing the same memory space, each potentially corresponding to different users; and v. ensuring that the code running inside an IEE does not leak sensitive information to untrusted code running outside. The resulting MPC protocol is non-interactive in the sense that each user can perform an initial set-up, and then provide its inputs and receive outputs independently of other protocol participants, which means that it provides a solution for "secure computation on the web" [HLP11] with standard MPC security.

Finally, we present an experimental evaluation of this MPC protocol via a detailed comparison of this solution to state-of-the-art multiparty computation. The experimental results confirm the theoretical performance advantages highlighted above, in comparison to non hardware-based solutions. The most resilient implementation of a generic MPC protocol —sgx-mpc-nacl— relies on the NaCl[2] cryptographic library [BLS12] and inherits its careful approach to dealing with timing side-channels. Side-channels in SGX-like and TrustZone-like systems are also discussed and it is explained how this *constant-time* code thwarts *all* leaks based on control-flow or memory access patterns that depend on secret data. This coding policy is generally accepted as the best software-based countermeasure against timing attacks that one can adopt, eliminating attack vectors that may arise, not only from direct measurements of execution time, but also from indirect ones relying for example on cache and page fault correlations. sgx-mpc-nacl introduces a very small overhead over the functionality that needs to be executed: it uses elliptic-curve based technology for both key exchange (Diffie-Hellman) and digital signatures, and a combination of the Salsa20 and Poly1305 encryption and authentication schemes [BLS12] for authenticated encryption. This implementation is functionality agnostic and can be used to outsource to the Cloud arbitrary off-the-shelf collaborative software, enabling multiple parties to jointly execute complex interactive computations without revealing their own inputs. One should of course mention that, in order to meet the level of side-channel attack resilience of sgx-mpc-nacl, the code that is outsourced to the Cloud should itself be implemented according to the constant-time coding policy. This, however, is a software engineering issue that is outside of the scope of this work.

1.2 Sources and structure

This section provides a list of the author's publications produced as a consequence of the work presented in this thesis, as well as projects in which the author actively contributed

[2] `https://nacl.cr.yp.to`

with the presented work. The listed published papers map to the two major stages of work development, first formalising attested computation and rigorously specifying the security guarantees achievable by the deployment of IEE-based systems, and second refining this notion to capture more general models with multiple local participants.

[BPSW16] Foundations of Hardware-Based Attested Computation and Application to SGX.

Barbosa, Portela, Scerri, Warinschi (EuroS&P 2016)

The author played a central role in the development of the whole paper, with the supervision of the remaining authors. More specifically, this includes the design of the IEE abstraction for non-labelled programs and machines, the formalisation of *attested computation* and its security analysis, the development of attested key exchange for attested computation and its security proof, the utility theorem proof, and the secure outsourced computation protocol.

[BBB+17] Secure Multiparty Computation from SGX.

Bahmani, Barbosa, Brasser, Portela, Sadeghi, Scerri, Warinschi (FC 2017)

The author played a central role in the development of the whole paper, with the supervision of the remaining authors. Specifically, this includes the generalisation to labelled programs, the formalisation of *labelled attested computation*, the extension of the utility theorem to multiple users using labels, and the design of the IEE-enabled multiparty computation protocol, its security model and associated security analysis. The implementation effort of the MPC protocol on Intel's SGX was a joint effort of the author and Raad Bahmani, also with the supervision of the remaining authors.

Throughout the Ph.D. programme, the author was also actively involved in National and European projects, which provided both crucial insight in the fundamental security definitions associated with standard secure computation scenarios, as well as feedback for use cases validating the applicability of the approach. The projects and the author's participation is now listed.

PRACTICE [3]

The author actively contributed to the specification of formal verification requirements (WP12, D12.3 [BNB+15]), and the development of precise specification of secure computation functionalities (WP13, D13.2 [BPS+16]).

SAFECLOUD [4]

The author actively contributed to the specification of privacy-protecting mechanisms for data querying (WP3, D3.2 [PPM+16]), to the architectural definition that captures

[3] `https://practice-project.eu/`

[4] `http://www.safecloud-project.eu/`

the application of these techniques in the SafeCloud framework (WP3, D3.1 [PBT+16], D3.5 [BTS+17a]), as well as to support the implementation of a key-value store following the framework specifications (WP3, D3.4 [BTS+17b], D3.5 [BTS+17a]).

NANOSTIMA [5]

NanoSTIMA sprawls several research lines with different scopes. The author works on Research Line 3, WP1. This research line directly benefits from the work developed on secure computation using IEEs, as [BBB+17] was presented in NanoSTIMA 3.5 Meeting in the context of enabling secure collection and computation of medical data.

Works not included in this thesis

For the development of efficient real-world solutions in the context of the SAFECLOUD project, the author also explored how to employ mechanisms for privacy-preserving data processing in the context of databases. More specifically, this work produced a modular and extensible architecture for NoSQL databases where developers can associate different data attributes with cryptographic procedures, thus allowing for tailor-made instantiation of security mechanisms within a system.

The design of such architecture has further motivated the research on secure outsourced computation from trusted hardware. It should be possible to modularly extend the server-side component of NoSQL databases to employ IEEs in the processing of queries, thus presenting yet another trade-off to explore in real-world deployments. Indeed, the results presented in this thesis suggest considerable performance gains in the execution of privacy-preserving data processing within IEE-enabling machines.

[MPP+17] A Practical Framework for Privacy-Preserving NoSQL Databases
Macedo, Paulo, Pontes, Portela, Oliveira, Matos, Oliveira (SRDS 2017)
The author played a central role in the development of the proposed NoSQL framework, both on the design of the required cryptographic implementations, and on the interplay between NoSQL mechanisms and cryptographic components. The author was also involved in the use case selection and specification, as well as on the experimental result validation.

Document structure

The thesis structure is as follows.

- Chapter 2 presents preliminary cryptographic definitions relevant throughout the full thesis, whose reading is optional for readers with experience in the subject matter.

[5] `http://nanostima.cintesis.eu/`

- Chapter 3 provides context of related work with respect to the state-of-the-art software based approaches to passively secure computation and actively secure computation.
- Chapter 4 discusses modern trusted hardware in cryptography, namely the hardware platforms of Intel's SGX and ARM TrustZone, as well as related contributions with respect to protocols relying on these technologies and recent side-channel attacks.
- Chapter 5 formalises the fundamental concepts of IEEs, programs, compositions and machines. It then presents the basic cryptographic primitives for IEEs, namely Attested Computation and Labelled Attested Computation, and rigorously defines the ensured security guarantees.
- Chapter 6 proposes how one can implement an efficient key-exchange protocol in the context of IEE-enabling hardware, and establishes utility theorems useful for combining these results with full-fledged protocols.
- Chapter 7 demonstrates how the proposed example of one-to-one secure outsourced computation can be achieved given Attested Computation and this new flavour of key-exchange for IEEs.
- Chapter 8 further explores the possibilities of using IEEs to the deployment of Secure Function Evaluation and Multiparty Computation protocols.
- Chapter 9 presents experimental results for the implementation of the proposed MPC protocol, to validate the expected theoretical advantages of this approach.
- Chapter 10 concludes the work and suggests branching future research directions.

Chapter 2

Preliminaries

In this chapter, we present preliminary definitions that will be relevant throughout the dissertation. These will include: i.) standard definitions of cryptographic primitives for message authentication codes, authenticated encryption and digital signatures; ii.) security definitions of key exchange protocols and a description of how these are formally analysed; and iii.) security notions and provable security approaches to the development of secure multiparty computation.

2.1 Basic cryptographic primitives

The presented fundamental cryptographic definitions are based on [KL14].

Symmetric Encryption

The symmetric encryption setting considers participants Alice and Bob, both in possession of the same secret key key, where Alice wants to send message msg to Bob via a channel controlled by eavesdropper Eve. This goal can be achieved using symmetric encryption. A symmetric encryption scheme is a triple $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ of algorithms. $\mathsf{Gen}(1^\lambda)$ is a probabilistic algorithm that takes a security parameter 1^λ and outputs a key key generated according to a distribution determined by the scheme. $\mathsf{Enc}(\mathsf{key}, \mathsf{msg})$ is a probabilistic algorithm that takes a key key and a message msg to produce an encrypted message cph. $\mathsf{Dec}(\mathsf{key}, \mathsf{cph})$ is a deterministic algorithm that takes the key key and a message encryption cph and reverts the encryption process, producing the original message msg, or a failure symbol $\perp$.

Developing a usable encryption scheme requires for two main properties to be ensured: *correctness* and *security*. Correctness will ensure that the scheme is behaving appropriately when a legitimate execution is performed. For encryption schemes, correctness requires that,

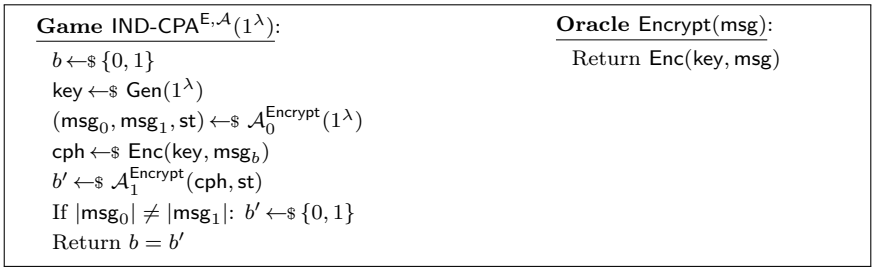


Figure 2.1: Games defining the security of a symmetric encryption scheme.

for any key within the key space, one can successfully recover the original message after a sequential encryption and decryption with the generated key, i.e.

$$\forall \mathsf{key} \in \{0,1\}^\lambda$$
$$\forall \mathsf{msg}, \mathsf{Dec}(\mathsf{key}, \mathsf{Enc}(\mathsf{key}, \mathsf{msg})) = \mathsf{msg}$$

Security aims to ensure confidentiality over an encrypted message. Here, we want to show that an adversary (Eve), when given an encrypted message, is unable to extract any useful information from it. Formally, this is translated into a security experiment that rigorously details what are the adversarial conditions, and what is the "win criteria" that constitutes a successful attack to the scheme. In this sense, encryption schemes are required to ensure Indistinguishability against Chosen Plaintext Attacks (IND-CPA), as detailed in Figure 2.1 and described as follows. After a bit is flipped and a key is generated, the adversary $\mathcal{A}_0$ gets to arbitrarily select two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$ of the same length.[1] Then, msg_0 or msg_1 will be encrypted, according to bit b, and its encrypted result cph will be returned to the adversary. $\mathcal{A}_1$ must then provide a guess as to what message was encrypted, and wins the game if it guesses correctly ($b = b'$). Throughout the process, the adversary has access to an oracle $\mathsf{Encrypt}$, allowing for it to retrieve a polynomial number of encryptions of arbitrarily chosen plaintexts.

The probability of adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ winning the security game over the random choices of all algorithms executed in the experiment and bit b, selecting b' such that $b = b'$ with probability bias from $\frac{1}{2}$, defines the advantage of an adversary breaking the encryption scheme. $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{E},\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.1 as follows.

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{E},\mathcal{A}}(\lambda) = 2 \cdot \Pr[\mathsf{IND\text{-}CPA}_{\mathsf{E},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}] - 1$$

Definition 1 (IND-CPA Security)**.** *An encryption scheme* $\mathsf{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *ensures semantic security if, for all probabilistic polynomial time (ppt) adversaries* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ *and negligible function* μ*,*

[1] This excludes attacks where an adversary selects messages with different lengths, and will trivially distinguish their encryptions by evaluating the size of the received ciphertext.

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{E},\mathcal{A}}(\lambda) < \mu(\lambda)$$

Message Authentication Codes

Message authentication codes (MACs) consider a similar scenario to the symmetric encryption, where Alice and Bob are in possession of the same secret key key, but Alice now wants to send a message/tag pair $(\mathsf{msg}, \mathsf{t})$ to Bob via a channel controlled by eavesdropper Eve. The goal here is not message confidentiality, but rather to ensure Bob that only Alice could have sent message msg. A message authentication code scheme Π is a triple of algorithms (Gen, Auth, Ver). $\mathsf{Gen}(1^\lambda)$ is a probabilistic algorithm that takes a security parameter 1^λ and outputs a key key generated according to a distribution determined by the scheme. $\mathsf{Auth}(\mathsf{key}, \mathsf{msg})$ is a deterministic algorithm that takes a key key and a message msg to produce a tag t. $\mathsf{Ver}(\mathsf{key}, \mathsf{msg}, \mathsf{t})$ is a deterministic algorithm that takes the key key, a message msg and a tag t, and verifies if the tag corresponds to the given message, outputting T if so, and F otherwise.

For message authentication schemes, correctness requires that, for any key within the key space, one can successfully validate messages after sequentially producing tags out of those messages, i.e.

$$\forall \mathsf{key} \in \{0,1\}^\lambda$$
$$\forall \mathsf{msg}, \mathsf{Ver}(\mathsf{key}, \mathsf{msg}, \mathsf{Auth}(\mathsf{key}, \mathsf{msg})) = \mathsf{T}$$

Security is focused on ensuring message authentication, proving that the received message is indeed msg, instead of some $\mathsf{msg}' \neq \mathsf{msg}$ that Eve might have created. Formally, this is known as existential unforgeability, which is also described via a security experiment that aims to rigorously define what the adversary must produce to constitute a feasible attack to the message authentication scheme. The standard notion of existential unforgeability for MACs [BKR94] is considered, as detailed in Figure 2.2 and described as follows. After a key is generated, adversary $\mathcal{A}$ must provide a pair $(\mathsf{msg}, \mathsf{t})$, which will constitute a message and its associated tag. The adversary is allowed to interact with Oracle Tag, that provides tags for arbitrary messages. The adversary wins the experiment if the produced $(\mathsf{msg}, \mathsf{t})$ successfully validates, and if that message was not queried to Oracle Tag.[2]

Security requires that the production of a valid message/tag to be either result of a legitimate oracle execution (therefore not constituting a forgery), or computationally infeasible without access to the secret key. $\mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.2 as follows.

[2]Again, this excludes a trivial attack where the adversary simply requests a tag for some message and outputs said message alongside the Oracle's result.

Figure 2.2: Games defining existential unforgeability of a message authentication scheme.

$$\mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{A}}(\lambda) = \Pr[\mathsf{Auth}_{\Pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}]$$

Definition 2 (MAC Unforgeability)**.** *A MAC scheme* $\Pi = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Ver})$ *ensures existential unforgeability if, for all ppt adversaries* $\mathcal{A}$ *and negligible function* μ*,*

$$\mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{A}}(\lambda) < \mu(\lambda)$$

Authenticated Encryption

An authenticated encryption scheme Λ is denoted as a triple of algorithms (Gen, Enc, Dec). $\mathsf{Gen}(1^\lambda)$ is a probabilistic algorithm that takes a security parameter 1^λ and outputs a key key generated according to a distribution determined by the scheme. $\mathsf{Enc}(\mathsf{key}, \mathsf{msg})$ is a probabilistic algorithm that takes a key key and a message msg to produce an encrypted message cph. $\mathsf{Dec}(\mathsf{key}, \mathsf{cph})$ is a deterministic algorithm that takes the key key and a message encryption cph and reverts the encryption process, producing the original message msg, or $\perp$ if the decryption process fails.

Correctness requires that, after a legitimate key generation, sequential encryption and decryption recovers the original message. Security of authenticated encryption provides both Indistinguishability under Chosen Ciphertext Attack (IND-CCA) and existential unforgeability. We use the standard notions of indistinguishability and existential unforgeability for authenticated encryption schemes [KY00].

IND-CCA is detailed in Figure 2.3 (top) and is described as follows. After a bit is flipped and a key is generated, the adversary $\mathcal{A}_0$ gets to arbitrarily select two messages $\mathsf{msg}_0, \mathsf{msg}_1$ of the same length. Then, msg_0 or msg_1 will be encrypted, according to bit b, and its encrypted result cph will be returned to the adversary. $\mathcal{A}_1$ must then provide a guess as to what message was encrypted, and wins the game if it guesses correctly ($b = b'$) without having requested the decryption of the ciphertext to the provided oracle. Throughout the process, the adversary has access to oracles $\mathsf{Encrypt}, \mathsf{Decrypt}$, allowing for it to retrieve a polynomial number of encryptions/decryption of arbitrarily chosen plaintexts/ciphertexts.

Existential unforgeability is similar to the security guarantees ensured by MACs, in the sense that the adversary should not be able to produce any valid ciphertext without access to the

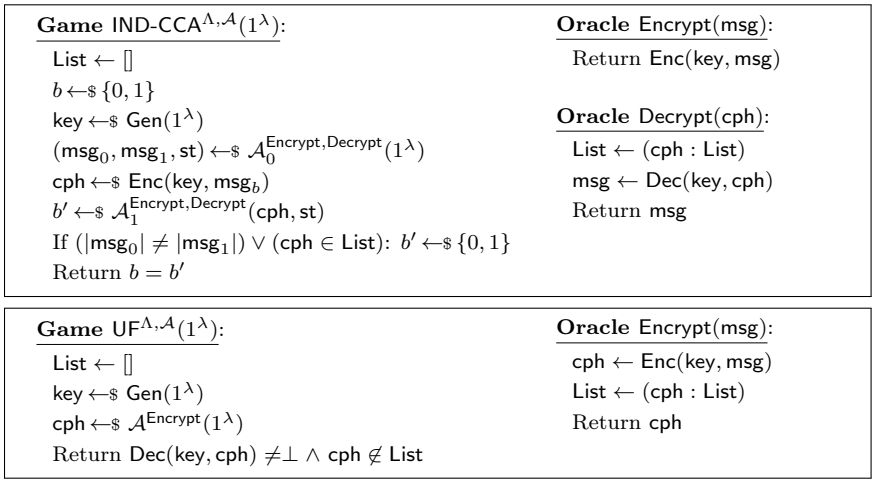


Figure 2.3: Games defining the security of an authenticated encryption scheme. Ciphertext indistinguishability (top) and existential unforgeability (bottom).

symmetric secret key. The notion of existential unforgeability for authenticated encryption schemes is detailed in Figure 2.3 (bottom) and described as follows. After a key is generated, adversary $\mathcal{A}$ must provide a ciphertext cph. The adversary is allowed to interact with Oracle Encrypt, that encrypts arbitrary messages. The adversary wins the experiment if the produced cph successfully decrypts, and if that message was not queried to Oracle Encrypt. Security of authenticated encryption requires both these notions to be fulfilled.

$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\Lambda,\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.3 (top) as follows.

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\Lambda,\mathcal{A}}(\lambda) = 2 \cdot \Pr[\mathsf{IND\text{-}CCA}_{\Lambda,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}] - 1$$

and $\mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.3 (bottom) as follows.

$$\mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{A}}(\lambda) = \Pr[\mathsf{UF}_{\Lambda,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}]$$

Definition 3 (Authenticated Encryption Security)**.** *An authenticated encryption scheme* $\Lambda = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is secure if, for all ppt adversaries* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ *and negligible function* μ*,:*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\Lambda,\mathcal{A}}(\lambda) < \mu(\lambda)$$

$$\mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{A}}(\lambda) < \mu(\lambda)$$

Figure 2.4: Games defining existential unforgeability of a digital signature scheme.

Digital Signature Schemes

In the public key setting, Alice communicates over a network controlled by Eve. Alice has a secret key sk and a publicly available key pk. Alice's goal is to produce a message/signature pair (msg, σ) to anyone with access to pk via a channel controlled by Eve. The goal is to ensure that σ can effectively be used to ensure that msg was produced by the owner of the secret key sk, Alice. There is no assumption with respect to a pre-established secret key, however it must be the case that Eve cannot replace pk by some alternative key pk'. This is achieved via a Digital Signature scheme, a triple of algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. $\mathsf{Gen}(1^\lambda)$ is a probabilistic algorithm that takes a security parameter 1^λ and outputs a key pair $(\mathsf{sk}, \mathsf{pk})$ generated according to a distribution determined by the scheme. $\mathsf{Sign}(\mathsf{sk}, \mathsf{msg})$ is a (potentially probabilistic) algorithm that takes a secret key sk and message msg to produce signature σ. $\mathsf{Vrfy}(\mathsf{pk}, \mathsf{msg}, \sigma)$ if a deterministic algorithm that takes public key pk, a message msg and a signature σ, and verifies if the signature σ corresponds to the given message, outputting T if so, and F otherwise.

After a legitimate generation of a key pair, signing a message with the secret key will always successfully validate the associated message if verified with the public key. Security will follow the standard notion of existential unforgeability for signature schemes [GMR88], an experiment that is detailed in Figure 2.4 and described as follows. After a key pair is generated, adversary $\mathcal{A}$ must provide a pair (msg, σ), which will constitute a message and its associated signature. The adversary is given the public key pk, and is allowed to interact with Oracle Sigm that provides signatures for arbitrary messages. The adversary wins the experiment if the produced (msg, σ) successfully verifies, and if that message was not queried to Oracle Sign.

Security requires the production of such message/signature pair to be computationally infeasible without access to the secret key. $\mathsf{Adv}^{\mathsf{Auth}}_{\Sigma,\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.4 as follows.

$$\mathsf{Adv}^{\mathsf{Auth}}_{\Sigma,\mathcal{A}}(\lambda) = \Pr[\mathsf{Auth}_{\Sigma,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}]$$

Definition 4 (Signature Unforgeability)**.** *A digital signature scheme* $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *ensures existential unforgeability if, for all ppt adversaries* $\mathcal{A}$ *and negligible function* μ*,*

$$\mathsf{Adv}^{\mathsf{Auth}}_{\Sigma,\mathcal{A}}(\lambda) < \mu(\lambda)$$

2.2 Key Exchange

Key exchange protocols consider a setting where Alice and Bob are communicating over an insecure network, and want to agree upon a secret key key, but are unwilling to reveal such secret to adversary Eve, which is in control of the communications channel. Two adversarial models are considered, defining the strength of the opponent. *Passive* adversaries are limited to eavesdropping. They observe the protocol being executed, and can compute over the acquired data. *Active* adversaries can act arbitrarily, tampering with messages sent over the channel, or outright preventing messages from being delivered. We employ a form of key exchange protocol that does not rely on long term secret/state or global setup and for which weak security guarantees will be required (essentially, security against a passive adversary); the classical Diffie-Hellman key exchange is a standard example for these protocols. The following definitions are based on [CK01, BSWW13].

A key exchange protocol is defined by a single ppt algorithm Π used by communicating parties. Each party has a unique identifier id, and can execute several instances of the protocol with different parties. Throughout the thesis, identifiers will be arbitrary strings, which will be given meaning by the higher-level applications relying on protocols under analysis. For $s \in \mathbb{N}$, let Π^s_{id} be the s instance of party id. Each instance is assumed to maintain the following variables.

- $\mathsf{st} \in \{0,1\}^*$ is some internal state for that instance.
- $\delta \in \{\mathsf{derived}, \mathsf{accept}, \mathsf{reject}, \perp\}$ is the state of the key.
- $\rho \in \{\mathsf{initiator}, \mathsf{responder}\}$ is the role of the participant.
- oid is the identity of the owner of the instance.
- sid and pid are the session identifier and the identity of the partner, respectively.
- key is the agreed session key.

It is required that $\mathsf{key} = \perp$ unless $\delta \in \{\mathsf{derived}, \mathsf{accept}\}$, and that oid, ρ, sid, key are only assigned once during the entire execution of the protocol (the first two when the session is initialized). Running Π with message msg, role ρ and state st to produce msg' and the updated state st' is denoted by $(\mathsf{msg}', \mathsf{st}') \leftarrow\!\!\$\ \Pi(1^\lambda, \mathsf{msg}, \mathsf{id}, \rho, \mathsf{st})$.

Message ϵ will be used to refer to the empty string. This will be passed to both parties as the initial state; it will be passed to the initiator as first input message; and it will be returned as output message by the party that executes last, to denote that no further interaction is needed. Long term secrets and/or shared initial state between several instances run by the same identity can be captured by setting their initial state accordingly.

A key exchange protocol is correct if, after a complete (honest) run between two participants with complementary roles, both reach the accept state, both derive the same key and session identifier, and both obtain correct partner identities.

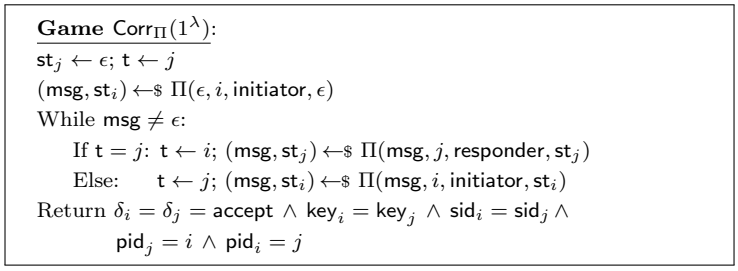


Figure 2.5: Game defining the correctness of protocol Π.

Definition 5 (Correctness)**.** *A key exchange protocol* Π *is correct if, for any distinct party identities* i *and* j*, the experiment in Figure 2.5 always returns* T*.*

Execution model

The adopted security notion is a restriction of the scenario considered in [KY03] that excludes corruptions. The execution model considers an active adversary, which is run on the security parameter, and which can interact with the following oracles:

- $\mathsf{Send}(i, s, \mathsf{msg})$ sends message msg to instance Π_i^s and receives output o. If $\delta = \mathsf{derived}$, it checks if the key derived for the executed session exists in list fake. If not, it generates a new key^* uniformly at random, and adds $(\mathsf{key}, \mathsf{key}^*)$ to the list. Finally, Send outputs the corresponding reply o.

- $\mathsf{Execute}(i, j)$ runs a new instance of the protocol between distinct parties i and j. It then generates a new key^* uniformly at random, and adds $(\mathsf{key}, \mathsf{key}^*)$ to the list. Finally, it outputs the transcript of the protocol execution and the session identifier associated with it.

- $\mathsf{Reveal}(i, s)$ outputs the session key key of Π_i^s.

- $\mathsf{Test}(i, s)$ will return $\perp$ if $\delta_i^s \neq \mathsf{accept}$. Otherwise, if $b = 0$ it outputs the key associated with Π_i^s. If $b = 1$, it searches for the key associated with Π_i^s in list fake and returns the associated key^*.

When the adversary terminates interacting with the oracles, it will eventually output a bit b' which represents his guess on what the challenge bit b is.

Entity authentication is defined following [BSWW13]. Observe that, in the case of passively secure key exchange, this is essentially a correctness property. First, a notion of partnering will be introduced, which informally states that two oracles which have derived keys are partners if they share the same session identifier. The definition makes use of the

following predicate on two instances Π_i^s and Π_j^t holding states $(\mathsf{st}_i^s, \delta_i^s, \rho_i, \mathsf{sid}_i^s, \mathsf{pid}_i^s, \mathsf{key}_i^s)$ and $(\mathsf{st}_j^t, \delta_j^t, \rho_j, \mathsf{sid}_j^t, \mathsf{pid}_j^t, \mathsf{key}_j^t)$, respectively:

$$\mathrm{P}(\Pi_i^s, \Pi_j^t) = \begin{cases} \mathsf{T} \text{ if } \mathsf{sid}_i^s = \mathsf{sid}_j^t \ \wedge \ \delta_i^s, \delta_j^t \in \{\mathsf{derived}, \mathsf{accept}\} \\ \mathsf{F} \text{ otherwise.} \end{cases}$$

Definition 6 (Partner)**.** *Two players* Π_i^s *and* Π_j^t *are partnered if* $\mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T}$.

The employed authentication notion relies on three further definitions, which demand that partnering will need to be *valid*, *confirmed* and *unique*. In short, these three requirements ensure that any instance that accepts has a partner, that this partner is unique and that partners share the same key. More specifically, valid partners must have corresponding partner identifiers (i.e. they both believe they are talking with each other), have different roles and share the same key. Confirmed partners ensures that, for each oracle that accepts, there exists at least one partner; and for unique partners there exists at most one.

Definition 7 (Valid Partners)**.** *A protocol* Π *ensures valid partners if the bad event* notval *does not occur, where* notval *is defined as follows:*

$$\begin{aligned} &\exists \Pi_i^s, \Pi_j^t \quad s.t. \ \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T} \wedge \\ &\quad (\ \mathsf{pid}_i^s \neq \mathsf{oid}_j^t \ \vee \ \mathsf{pid}_j^t \neq \mathsf{oid}_i^s \ \vee \ \rho_i = \rho_j \ \vee \ \mathsf{key}_i^s \neq \mathsf{key}_j^t \ ). \end{aligned}$$

Definition 8 (Confirmed Partners)**.** *A protocol* Π *ensures confirmed partners if the bad event* notconf *does not occur, where* notconf *is defined as follows:*

$$\exists \Pi_i^s \quad s.t. \quad \delta_i^s = \mathsf{accept} \wedge \forall \Pi_j^t, \ \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{F}.$$

Definition 9 (Unique Partners)**.** *A protocol* Π *ensures unique partners if the bad event* notuni *does not occur, where* notuni *is defined as follows:*

$$\begin{aligned} &\exists \Pi_i^s, \Pi_j^t, \Pi_k^r \quad s.t. \\ &\quad (j, \mathsf{t}) \neq (k, r) \ \wedge \ \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T} \ \wedge \ \mathrm{P}(\Pi_i^s, \Pi_k^r) = \mathsf{T} \end{aligned}$$

Intuitively, an adversary is considered to violate two-sided entity authentication if he can lead an instance of an honest party running the protocol to accept, and in doing that cause one of the bad events notval, notconf, notuni.

To exclude breaks via trivial attacks, legitimate adversaries are defined as those who ensure that the following freshness criteria is satisfied for his $\mathsf{Test}(i, s)$ queries: i.) $\mathsf{Reveal}(i, s)$ was not queried; and ii.) for all Π_j^t such that $\mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T}$, $\mathsf{Reveal}(j, t)$ was not queried. Only experiments in which the adversary is found to be legitimate will be considered. The winning event guess is defined to be $b = b'$ at the end of the experiment.

Definition 10 (Passive AKE security)**.** *A protocol* Π *is passively secure if, for any legitimate ppt adversary interacting with the execution environment described above:*

- *The adversary violates two-sided entity authentication with negligible probability, i.e.* $\Pr[\,\mathsf{notval} \vee \mathsf{notconf} \vee \mathsf{notuni}\,]$
- *Its key secrecy advantage* $2 \cdot \Pr[\,\mathsf{guess}\,] - 1$ *is negligible.*

One-Sided Authentication

A weaker form of entity authentication guarantee will also be considered, where only some parties are authenticated. To this end, it is now relevant to distinguish between Loc and Rem parties. The former is identified with responders, and the latter with initiators. In one-sided authentication, only initiators (i.e., remote parties) are authenticated, which means that responders (i.e., local parties) do not need to keep any long term secrets. Again, definitions of [BSWW13] will be followed to produce one-sided versions of the definitions for valid partners and confirmed partners.

Intuitively, on acceptance, a local party will be assured that a valid unique partnering session exists. Security will still need to ensure that each local (or remote) session has at most one remote (respectively local) partnered session. However, confirmation for the remote machine is no longer required: the remote party is allowed to accept even if there is no local matching session that has accepted.

Definition 11 (One-Sided Valid Partners)**.** *A protocol* Π *ensures one-sided valid partners if the bad event* os-notval *does not occur, where* os-notval *is defined as follows:*

$$\begin{aligned}&\exists \Pi_i^s, \Pi_j^t \quad s.t. \quad i \in \mathsf{Loc} \wedge \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{T} \wedge \\ &\quad (\,\mathsf{pid}_i^s \neq j \vee \rho_i^s \neq \mathsf{initiator} \vee \rho_j \neq \mathsf{responder} \vee \mathsf{key}_i^s \neq \mathsf{key}_j^t\,).\end{aligned}$$

Definition 12 (One-Sided Confirmed Partners)**.** *A protocol* Π *ensures one-sided confirmed partners if the bad event* os-notconf *does not occur, where* os-notconf *is defined as follows:*

$$\begin{aligned}&\exists \Pi_i^s \quad s.t. \quad i \in \mathsf{Loc} \wedge \\ &\quad \delta_i^s = \mathsf{accept} \wedge \forall \Pi_j^t,\, \mathrm{P}(\Pi_i^s, \Pi_j^t) = \mathsf{F}.\end{aligned}$$

When one-sided authentication suffices, then the definition of key exchange security is weakened by relaxing condition 1 in Definition 10, which is modified to refer to the following event.

$$\Pr[\,\mathsf{os\text{-}notval} \vee \mathsf{os\text{-}notconf} \vee \mathsf{notuni}\,]$$

2.3 Secure multiparty computation

In Secure Multiparty Computation, multiple participants want to compute some function over their input data and receive its output, without revealing their private inputs to each other. Yao's Millionaire problem is an example of this application, depicting two millionaires that wish to know who is richer, without disclosing any additional information about each other's wealth. Modern implementations allow for complex computations to be performed over private data, such as data mining algorithms [BJL12, BNTW12, Jag10], secure auctions [BCD+09] or privacy-preserving satellite collision detection [KW15].

Secure computation can be defined as a problem in which n players P_n in possession of inputs $\{x_1, \ldots, x_n\}$ agree to compute a stateless function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, such that P_i knows y_i and learns nothing additional that couldn't be deduced from their personal input and output. This specific setting is known as secure function evaluation, and can be generalised to general multiparty computation by considering stateful (reactive) functions.

General multiparty computation can trivially be achieved if one considers secure channels and a trusted third party. For any number of participants, each P_i would only need to send x_i to the trusted party and expect to receive y_i as, by definition of the trusted party, function f is executed correctly and securely. Therefore, the problem of MPC refers solely to the cases in which these conditions are unavailable.

Participants in a MPC protocol are modeled as probabilistic polynomial time interactive Turing machines (ITM). Execution of the protocol takes place in the presence of an environment $\mathcal{Z}$. $\mathcal{Z}$ is also a ppt ITM, and is responsible for providing input and receiving output from parties, as well as for modelling the protocol's adversary, corrupting and controlling corrupt players. Players of a MPC protocol exchange messages according to an established communication model. Systems may assume the existence of pair-wise secure channels, in which case the adversary gains no information regarding data exchanged between honest players. When the system makes no such assumption, players communicate through a channel that is accessible to the adversary, where security can only be achieved in a cryptographic sense. Communication may also be considered as *synchronous* or *asynchronous*. Synchronous communication implies that processors have internal clocks that are synchronized to some extent and, whenever a message is sent, it is guaranteed to arrive within a predetermined time frame. Asynchronous communication does not make assumptions regarding assurance of delivery or timing on delivered messages. This environment description models all possible applications and protocol stacks where MPC could be used.

A multiparty computation's security approach is, typically, characterizable by its adversarial model and associated definition of security. The first relates to environment assumptions in which it performs, while the latter specifies the security model considered, and how it can be measured/proved. The following definitions are based on Cramer, Damgård and Jesper

Buus' lecture notes [CDN05].

Adversarial assumptions

To capture the worst case scenario, the adversary in MPC is considered to be a single, monolithic entity, capable of controlling multiple corrupt parties while maintaining a global state. This corruption implies that the adversary gains information regarding the designated party, namely the private input x_i and every private data resulting from the process executed up to that point. This corruption may confer the adversary different levels of power:

Passive adversary is also known as "honest-but-curious", and reflects a behaviour that, without breaking the correct protocol execution, attempts to infer additional knowledge from the information it is able to obtain.

Active adversary is also known as "malicious", and reflects a behaviour that may deviate from protocol procedure. As such, it is able to perform arbitrary actions that may not produce correct (or even coherent) results, in order to achieve its goals.

An adversary is, typically, only capable of corrupting a subset of players, since no protocol can be secure if we assume the possibility for the adversary to corrupt any number of parties. As such, an adversary structure consists of all subsets with cardinality inferior than a predetermined threshold, and its corruption may be *static* or *adaptive*:

Static corruption means that the corrupted subset is chosen before starting the protocol, and remains unchanged for its full duration.

Adaptive corruption means that the adversary may choose to corrupt a new player at any time during the process, based on all the information obtained up to that point, assuming the resulting corrupted set remains within the threshold.

Security definition

The standard way to analyse a multiparty computation protocol is to take the approach of *real world versus ideal world*, a concept used in methods for formalizing composable security proofs, such as universal composability [Can01] (UC) or reactive simulatability [PW00]. Its intuition is to demonstrate that an adversary (with the power specified via the associated adversarial model) is unable to distinguish a real protocol execution from a simulated version of it. Observe that this entails both the notion of correctness, i.e. if the protocol is not executing according to its idealized version, then it is trivially distinguishable, and the notion of security, i.e. if private information was not necessary to present the adversary

with an indistinguishable simulated execution, then clearly the same private information is not accessible to the adversary in a real execution.

The real world considers a protocol Π run by participants $P_1, ..., P_n$ with some adversary $\mathcal{A}$ and environment $\mathcal{Z}$. The actual protocol execution maps to a sequence of activations. Once an activation terminates, the participant that had its input tape written on is activated next. The environment $\mathcal{Z}$ starts, and it might read the contents of uncorrupted parties' output tapes, or write on the input tapes of participants or the adversary. Upon adversary $\mathcal{A}$ activation, it may read its own tapes and communication tapes of all participants, and it can write on the communication tape of a participant (thus delivering a message)[3], or corrupt a participant. Corrupting a participant means that the adversary gains access to all tapes of that party, and controls all future actions of that party (while bound by the adversarial limitations established in the model). If the adversary has written on a participant's communication tape, that will be the next activation, otherwise the environment is activated. When a participant P is activated, it follows the protocol description, which results in either writing information to the outgoing communication tape (thus signalling a message for delivery), or writing some information on its output tape (declaring protocol output). The actual protocol execution terminates when the environment completes without writing on any input tape. As such, let $\mathsf{Real}_{\mathcal{Z},\Pi,\mathcal{A}}(1^\lambda)$ denote the output of $\mathcal{Z}$ upon interacting with $\mathcal{A}$ and participants $P_1, ..., P_n$ running protocol Π on security parameter 1^λ.

The ideal world is comprised of an ideal functionality $\mathcal{F}$, a set of dummy parties $\tilde{P}_1, ..., \tilde{P}_n$, a simulator $\mathcal{S}$ and an environment $\mathcal{Z}$. Similar to the real-world, an execution is described as a set of activations with similar sequencing rules. The environment $\mathcal{Z}$ begins by reading contents of participants (which are now dummy), and writing on input tapes of either participants or the simulator. Dummy participants $\tilde{P}$ are simple ITMs that, upon being activated with data on the input tape, simply writes it to the outgoing communication tape for the ideal functionality, and upon activation with input on communication channel (a message from the ideal functionality), simply writes it to the output tape. Messages exchanged between dummy parties and the functionality have secret and public contents, and the simulator is only capable of reading its public contents.[4] When the ideal functionality $\mathcal{F}$ is activated, it will read contents of incoming communication tapes, and potentially produce messages to dummy parties and/or to the simulator. When it completes, the environment is activated next. When the simulator $\mathcal{S}$ is activated, it can read its input tape and the public parts of messages on the communication tapes of participants and of the functionality, and then can perform one of four actions: i.) write a message from itself to the communication

[3]If the model assumes secure communication between participants, this might be limited to delivering messages that have been sent in the past. This allows for the adversary to delay messages, while limiting its power of message forgery.

[4]This can be further detailed by specifying what contents are public and what contents are private in the functionality description itself, which is something that the abstraction in this work will explicitly specify.

tape of the functionality, ii.) deliver a message from the outgoing communication of a participant to the ingoing communication of the functionality, iii.) deliver a message from the outgoing communication of the functionality to the communication tape of a participant, or iv.) corrupt a participant, gaining the same power as the adversary in the real world. Let $\mathsf{Ideal}_{\mathcal{Z},\mathcal{F},\mathcal{S}}(1^\lambda)$ denote the output of $\mathcal{Z}$ upon interacting with $\mathcal{S}$ and dummy participants $\tilde{P}_1, ..., \tilde{P}_n$ communicating with idealised functionality $\mathcal{F}$ on security parameter 1^λ.

A protocol Π is said to UC-realise an ideal functionality $\mathcal{F}$ if, for all ppt adversaries $\mathcal{A}$, there exists some simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish if it is interacting with $\mathcal{A}$ for a real-world execution, or with an $\mathcal{S}$ for an ideal-world execution.

Definition 13 (UC Security)**.** *Let $n \in \mathbb{N}$, let $\mathcal{F}$ be an ideal functionality and let Π be an n-party MPC protocol. Π UC-realises $\mathcal{F}$ if, for all $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that, for all $\mathcal{Z}$,*

$$\mathsf{Real}_{\mathcal{Z},\Pi,\mathcal{A}}(1^\lambda) \approx \mathsf{Ideal}_{\mathcal{Z},\mathcal{F},\mathcal{S}}(1^\lambda)$$

This approach is based on the intuition that, if such simulator exists, then the adversary can't possibly gain more information than what he would obtain when interacting with the same idealised version of said protocol. If the views of the real and ideal worlds follow the same distribution, the protocol is a *perfectly secure* realisation of the ideal functionality. If the views are only statistically indistinguishable, the protocol is a *statistically secure* realisation of the ideal functionality. If the views are only computationally indistinguishable, the protocol is a *computationally secure* realisation of the ideal functionality.

One of the major advantages of using this framework is that protocols UC-realising functionalities benefit from its composition theorem. Informally, this theorem states that if some protocol is demonstrated to UC-securely realise some functionality, then this is also the case whenever this functionality is used as a sub-procedure of any other protocol. In particular, this functionality can be relied upon to demonstrate the UC security of other protocols that use it, as something that can be assumed to be accessible in the real world.

This is formalised via the *hybrid model*, which extends the real world by allowing participants to send and receive messages to an unbound number of copies of $\mathcal{F}$, if it has been shown that there exists some protocol Π that UC-realises $\mathcal{F}$ according to Definition 13. The composition theorem states that if some ρ UC-realises $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, then an execution of protocol ρ^Π in the $\mathcal{F}$-hybrid model emulates an execution of ρ in the $\mathcal{G}$-hybrid model (the real world extended with the UC secure functionality).

Definition 14 (UC composition)**.** *Let $n \in \mathbb{N}$, let $\mathcal{F}$ and $\mathcal{G}$ be ideal functionalities, let ρ be an n-party MPC protocol in the $\mathcal{F}$-hybrid model, and let Π be an n-party MPC protocol that UC-realises $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Then, for any $\mathcal{A}$ in the $\mathcal{G}$-hybrid model, there exists $\mathcal{S}$ in the $\mathcal{F}$-hybrid model such that*

$$\mathsf{Hybrid}^{\mathcal{G}}_{\mathcal{Z},\rho^\Pi,\mathcal{A}}(1^\lambda) \approx \mathsf{Hybrid}^{\mathcal{F}}_{\mathcal{Z},\rho,\mathcal{S}}(1^\lambda)$$

$\textbf{Game Real}_{\Pi,\mathcal{A}}(1^\lambda)$:	$\textbf{Game Ideal}_{f,\mathcal{S}_1,\mathcal{S}_2,\mathcal{A}}(1^\lambda)$:
$(x_1, x_2, i, \mathsf{st}) \leftarrow\$\ \mathcal{A}_1(\,)$	$(x_1, x_2, i, \mathsf{st}) \leftarrow\$\ \mathcal{A}_1(\,)$
$(y, t_1, t_2) \leftarrow \Pi(x_1, x_2)$	$y \leftarrow f(x_1, x_2)$
	$t_i \leftarrow \mathcal{S}_i(x_i, y)$
Return $\mathcal{A}_2(x_1, x_2, y, t_i, \mathsf{st})$	Return $\mathcal{A}_2(x_1, x_2, y, t_i, \mathsf{st})$

Figure 2.6: Passively secure two-party SFE security experiment

In particular, when $\mathcal{G}$ is the empty functionality, then $\mathcal{G}$-hybrid model is the real world. In this case, Definition 14 ensures that Π remains secure when run concurrently with any protocol ρ.

Secure function evaluation with passive adversaries

The described UC model is designed to capture all possible instances where the problem of multiparty computation is applicable. The security analysis can be considerably simplified if restrictions are imposed to the system model. To illustrate this, the following notions consider the work of [Sch08] regarding how provable security can be achieved in the passive adversary model of two-party secure function evaluation.

Consider protocol participants P_1, P_2. Let $\Pi(x_1, x_2)$ be a two-party protocol producing (y, t_1, t_2) and $f(x_1, x_2)$ be a stateless function that produces output y, where x_i is the input of P_i, t_i is the protocol view of P_i (all randomness P_i provides as input to Π, as well as all messages exchanged during the execution of Π), and y is the output received by both participants. Proving the security of protocol Π requires the construction of two simulators $\mathcal{S}_1, \mathcal{S}_2$ that must be capable of presenting a protocol view (t_1, t_2, respectively) from their private inputs/outputs that is indistinguishable from the real-world execution.

Furthermore, these definitions can be represented and analysed in a game-based form, as demonstrated in Figure 2.6. The adversary gets to select all participants' inputs, and the participant it wishes to corrupt ($i \in \{1, 2\}$). Then, either the protocol is executed with the given inputs – the real world – or the functionality is executed with the given inputs and the simulator must produce the view of the corrupt participant, given its input and output – the ideal world. At the end of the experiment, the adversary gets to see all inputs and outputs, and the view of the corrupt participant's protocol (real or simulated).

$\mathsf{Adv}^{\mathsf{Auth}}_{\Sigma,\mathcal{A}}$ defines the advantage function associated with the experiment in Figure 2.6 as follows.

$$\mathsf{Adv}^{\mathsf{IND}}_{\Pi,f,\mathcal{A},\mathcal{S}_1,\mathcal{S}_2}(\lambda) = |\{\, \mathsf{Real}_{\Pi,\mathcal{A}}(1^\lambda) \,\} - \{\, \mathsf{Ideal}_{f,\mathcal{S}_1,\mathcal{S}_2,\mathcal{A}}(1^\lambda) \,\}|$$

Definition 15. *A protocol Π securely realises functionality f if there exists ppt simulators $\mathcal{S}_1, \mathcal{S}_2$ that, for every adversary $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2\}$ and negligible function μ, the advantage in*

distinguishing these distributions is negligible:

$$\mathsf{Adv}^{\mathsf{IND}}_{\Pi,f,\mathcal{A},\mathcal{S}_1,\mathcal{S}_2}(\lambda) < \mu(\lambda)$$

The presented security experiment is only valid for this specific setting. In particular, observe that it is unfit to capture active adversaries, since giving the protocol view directly to the adversary implies that it has no control over the protocol's execution process (other than selecting inputs a priori). One approach to extend it for allowing active adversaries could be to decompose Π into individual calls to honest participants, so as to enable the adversary to have its corrupt participant deliver arbitrary messages, and have the simulator be responsible for extracting the corrupt input from the interactions in the ideal world. This would additionally require for the simulator to have controlled access to f, so that it can provide the extracted corrupt input, and receive the associated output. Similar issues arise when considering an arbitrary number of participants, reactive (stateful) functionalities, or adaptive corruptions.

Chapter 3

Related Work

In this chapter, we present state-of-the-art research in software-based MPC solutions, to contextualize the approach taken to enable secure computation via IEE-enabled hardware. We begin by describing the basic mechanisms employed for secure computation, and then present approaches for passively secure MPC, as well as actively secure MPC.

3.1 Basic mechanisms for secure computation

The current approaches to secure computation can be grouped into families according to a small set of basic mechanisms they use. As such, we begin by describing the fundamentals of each of these mechanisms, as well as other techniques commonly used in the process of deploying real-world MPC solutions.

Garbling schemes

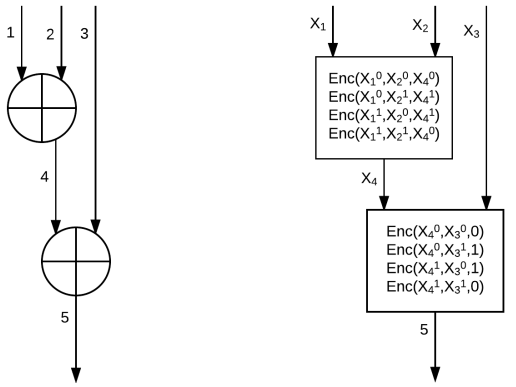
The following definitions for garbling schemes consider the work in [BHR12]. A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ is composed by a set of five algorithms, behaving as follows. String f defines the function to compute, and can be evaluated using $y \leftarrow \mathsf{ev}(f, x)$. Given input f and security parameter λ, Gb returns a triple with the garbled function and the functions for encoding inputs and decoding outputs $(F, e, d) \leftarrow\!\!\$\ \mathsf{Gb}(1^\lambda, f)$. Algorithm En performs the encoding, taking function e and plain input x to produce the garbled version of the same value $X \leftarrow \mathsf{En}(e, x)$. Algorithm Ev receives the garbled function F and maps each garbled input X to its corresponding output $Y \leftarrow \mathsf{Ev}(F, X)$. Finally, De is the algorithm used to convert garbled output Y to its corresponding plain output $y \leftarrow \mathsf{De}(d, Y)$. The presented definitions are representation-independent, in the sense that they capture instantiations other than garbled circuits (GC). However, we will only look at these notions from the scope of garbled circuits, since it is the most common approach to enable multiparty computation.

Security of garbling schemes is approached using a set of game-based experiments, matching the security properties desired from these schemes (Figure 5 of [BHR12]). Namely, garbling schemes are analysed with respect to privacy, obliviousness and authenticity. Informally, privacy ensures that knowledge of garbled input X and circuit (F, e, d) reveals no non-public information that cannot be derived from y. Obliviousness ensures that knowledge of the garbled circuit F and inputs X allows for the computation of Y but reveals nothing regarding x or y. Authenticity is a property that captures the inability for $\{F, X\}$ to be used to create a valid garbled output Y such that $F(X) \neq Y$.

An instantiation of a garbling scheme is proposed in the same work, denoted as *Garble1*, considering functions f represented as circuits. This scheme makes use of a dual-key cipher (DKC), which is essentially a symmetric encryption scheme relying on two keys for the encryption and decryption operations. More specifically, a DKC scheme is a triple $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ of algorithms. Gen is a probabilistic algorithm that takes a security parameter 1^λ and outputs keys $(\mathsf{key}_1, \mathsf{key}_2) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$ generated according to a distribution determined by the scheme. Enc is a probabilistic algorithm that takes keys $\mathsf{key}_1, \mathsf{key}_2$ and a message msg to produce an encrypted message $\mathsf{cph} \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{key}_1, \mathsf{key}_2, \mathsf{msg})$. Dec is a deterministic algorithm that takes keys $\mathsf{key}_1, \mathsf{key}_2$ and a message encryption cph and reverts the encryption process, producing the original message $\mathsf{msg} \leftarrow \mathsf{Dec}(\mathsf{key}_1, \mathsf{key}_2, \mathsf{cph})$, or a failure symbol $\perp$.

The actual process for garbling circuits begins by selecting two tokens for each wire in the circuit, representing the possible values of 0 and 1, respectively. We denote X_i^b as the token for wire i of original value b, and thus the encoding function e maps inputs $x_i \in \{0, 1\}$ to according values $\{X_i^0, X_i^1\}$. The tokens associated with circuit outputs are simply the plain values. Afterwards, all circuit gates are replaced by tables, mapping pairs of input wire tokens to their respective output wire tokens in an encrypted fashion. More specifically, each circuit gate g with input wires i, j and output wire k is replaced by a garbled table G with four entries, where each entry matches X_i^a, X_j^b with the respective output wire encrypted with the input tokens $\mathsf{Enc}(X_i^a, X_j^b, Y_k)$. Note that the security of the DKC scheme ensures that extracting an output wire token implies knowledge of both input wire tokens. As such, given circuit input wire tokens X, function F traverses the set of tables according to the original circuit description and produces output wire tokens Y. The function d has an empty description, as the output gates provide plain output values, i.e. Y are values in the clear.

As a toy example, consider the garbling of two exclusive-or operations, illustrated in Figure 3.1. Function f takes three inputs x_1, x_2, x_3 and performs $(x_1 \oplus x_2) \oplus x_3$. The corresponding circuit (Figure 3.1(a)) has input wires $1, 2, 3$, intermediate wire 4 and output wire 5, and has two XOR gates g_1, g_2: g_1 receiving inputs $1, 2$ and producing output 4, and g_2 receiving inputs $3, 4$ and producing output 5. The encoding function e maps input values to $\{X_1^0, X_1^1, X_2^0, X_2^1, X_3^0, X_3^1\}$. The garbled circuit function F will consist of a set of two tables



(a) Original Circuit (b) Garbled Circuit

Figure 3.1: Example of garbled circuit transformation.

(Figure 3.1(b)), matching the two gates:

$$G_1 = \{\mathsf{Enc}(X_1^0, X_2^0, X_4^0), \mathsf{Enc}(X_1^0, X_2^1, X_4^1), \mathsf{Enc}(X_1^1, X_2^0, X_4^1), \mathsf{Enc}(X_1^1, X_2^1, X_4^0)\}$$
$$G_2 = \{\mathsf{Enc}(X_3^0, X_4^0, 0), \mathsf{Enc}(X_3^0, X_4^1, 1), \mathsf{Enc}(X_3^1, X_4^0, 1), \mathsf{Enc}(X_3^1, X_4^1, 0)\}$$

Again, note that the output value is stored in a non-tokenized fashion, and thus the decoding function d is empty.

Garbled circuits are have received several optimizations since their inception. Free XOR gates [KS08] and garbled row reduction [NPS99, PSSW09, KMR14] have allowed for the sizes of garbled circuits and their encrypted tables to be considerably reduced, evaluation pipelining [HEKM11] enables protocols to generate garbled circuits at the same time they are being evaluated (thus removing restrictions on circuit size), and the usage of highly efficient hardware-based AES for garbling [BHKR13] enhances performance of circuit creation and evaluation.

Oblivious transfer

The two-party protocol of Oblivious Transfer (OT) was originally proposed in [EGL85], and considers a setting with participants P_1, P_2 with private inputs two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$ and bit v, respectively. The goal is for P_1 to transmit msg_v to P_2 without gaining knowledge of v, and without P_2 being able to recover both $(\mathsf{msg}_0, \mathsf{msg}_1)$. More specifically, an oblivious transfer protocol $\mathsf{OT}(\mathsf{msg}_1, \mathsf{msg}_2, v) = (\epsilon, \mathsf{msg}_v)$ must satisfy the following conditions:

- P_2 obtains msg_v and no additional information about msg_{v-1}.
- P_1 learns no information about v.

A classical instantiation of 1-out-of-2 OT is the Naor-Pinkas protocol [NP00], whose construction considered security against passive adversaries, and can be generalized to a 1-out-of-n

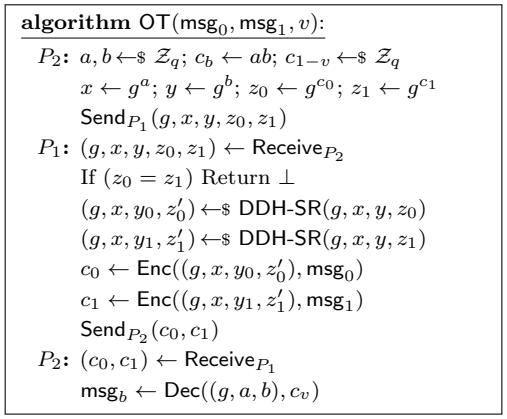


Figure 3.2: Naor-Pinkas OT protocol

OT protocol. Let G be a group of prime order q and $g \in \mathsf{G}$ be a generator. Let P_1 have input $(\mathsf{msg}_0, \mathsf{msg}_1)$ and P_2 have $v \in \{0, 1\}$, consider $(\mathsf{Enc}, \mathsf{Dec})$ public-key encryption algorithms and $(\mathsf{Send}_P, \mathsf{Receive}_P)$ be calls to a communication layer. Its construction is detailed in Figure 3.2.

Several improvements on OT have been proposed to achieve scalability for a larger number of transmissions [IKNP03, LLXX05], leveraging pre-computation of OTs without knowledge of inputs [Bea95], and general protocol optimizations [ALSZ13]. Extension to active adversaries on OT was a consideration for one of the earlier extensions in [IKNP03], where malicious P_1 are considered, and from then onwards many works have considered and improved the active OT setting [Lar14, PVW08, NNOB12, ALSZ15].

Secret sharing

Secret sharing techniques have been proposed in independent works [Sha79] and [Bla79] and are algorithms that allow for some value v to be shared between n participants $v_1, \ldots, v_n$, such that reconstructing the original value v requires at least t out of the n shares, and any other number k such that $k < t$ reveals no information about v. These schemes are known as t-out-of-n secret sharing schemes. The one originally proposed by Shamir [Sha79] was based on polynomial interpolation. First, a polynomial is of degree $t - 1$ is selected, and then any number of n points can be derived out of it, to be given to the participants. Any subset t of these points is sufficient for reconstructing the polynomial using interpolation, hence the threshold t-out-of-n.

The most commonly used secret sharing schemes for secure computations are n-out-of-n (all shares are required for value reconstruction), maintaining additively homomorphic properties.

Consider a group $\mathbb{Z}_N$. To secret share some value v among n participants, one can compute

$$\text{For } i \in (n-1) : v_i \leftarrow_{\$} \mathbb{Z}_N$$
$$v_n \leftarrow v - \sum_{i}^{(n-1)} v_i \bmod N$$

It easy to see that $v = \sum_i^n v_i$, so one can efficiently reconstruct the original value from the computed shares. This scheme also allows for operations such as additions to be performed locally

$$u + v = \sum_{i}^{n} u_i + \sum_{i}^{n} v_i \Leftrightarrow$$
$$\Leftrightarrow u + v = \sum_{i}^{n} (u_i + v_i)$$

Multiplication of shared values with constants can also be performed locally, but multiplication of two shared values requires for multiple rounds of share transmission, and thus cannot be performed locally. An alternative method for evaluating multiplications has been proposed by Beaver [Bea91], by using multiplication triples. This approach requires for the computation to be divided into two parts: a preparation stage, where some pre-processing is performed between the computing parties, and an online stage, where the actual computation is executed. Essentially, multiplication triples are random shares a_i, b_i, c_i such that $\sum_i^n c_i = \sum_i^n a_i + \sum_i^n b_i$ that can be generated in the setup phase using OT protocols [ALSZ13]. In the computation stage, the calculation of $u * v$ only requires the participants to exchange $d_i = v_i + a_i$ and $e_i = u_i + b_i$ and to compute $\sum_i^n d_i$, $\sum_i^n e_i$ and $(d*e)+(b_i*d)+(a_i*e)+c_i$ (which, after reconstruction of d, e, can now be performed locally). This pre-computation stage for multiplication triples minimizes the number of messages a participant must send to one, and reduces the overall size of the messages exchanged.

Alternatively, the GMW protocol [Gol09, GMW87] is a general approach for secure computation allowing for the evaluation of arbitrary Boolean functions securely, relying on the secret sharing of private inputs such that $v = \bigoplus_i^n v_i$. The GMW protocol benefits from XOR being an associative operation, allowing for all XOR gates to be evaluated locally, following the same logic as the one presented on secret shares over group $\mathbb{Z}_N$. The same approach for multiplication with pre-processing is also valid, where participants start with a_i, b_i, c_i such that $\bigoplus_i^n c_i = \bigoplus_i^n a_i \oplus \bigoplus_i^n b_i$ exchange d_i and e_i, so that the calculation of $\bigoplus_i^n d_i$, $\bigoplus_i^n e_i$ and $(d \wedge e) \oplus (b_i \wedge d) \oplus (a_i \wedge e) \oplus c_i$ can be performed locally.

Homomorphic encryption

Homomorphic Encryption (HE) schemes allow for executing computations over encrypted data without revealing the underlying plaintext. More concretely, a participant with private

input v encrypts its value using an HE scheme to $\mathsf{enc} \leftarrow\!\!\$\ \mathsf{HE.Enc}(\mathsf{key}, v)$ and sends it to another participant, which will take care of the computation. The properties of the scheme ensure that operations can be performed over this encrypted value without requiring the secret key, e.g. $\mathsf{enc}' \leftarrow\!\!\$\ \mathsf{HE.op}(\mathsf{enc})$. The computation result is then transmitted back to the input participant, which can retrieve the computation output by decrypting using the same HE scheme, $v' \leftarrow \mathsf{HE.Dec}(\mathsf{key}, \mathsf{enc}')$.

Partially homomorphic schemes, such as [Pai99, DJ01, ElG85, BV11, RSA78], can only be employed to execute a restricted subset of functions. These are classified by the type of operations permitted, namely additively homomorphic, multiplicatively homomorphic or somewhat homomorphic. For instance, in the Pallier cryptosystem [Pai99] the multiplication of ciphertexts ensures the addition of the plaintexts, i.e.

$$\mathsf{HE.Dec}(\mathsf{key}, \mathsf{HE.Enc}(\mathsf{key}, a) * \mathsf{HE.Enc}(\mathsf{key}, b)) = a + b$$

which means the Pallier cryptosystem is additively homomorphic. Basic RSA [RSA78] is multiplicatively homomorphic, since the addition of ciphertexts ensures the multiplication of the plaintexts. The scheme by Brakerski et al. [BV11] allows for both addition and multiplication to be performed, but is limited on the number of multiplications (more specifically, on the degree of the evaluated function, represented as a polynomial), and is thus considered somewhat homomorphic encryption. The direct application of these schemes is consequentially limited to very specialized use cases, such as secure elections [CGS97] (secure vote tallying), but these schemes can also be employed as a mechanism enabling more general cryptographic tools [DPSZ12].

The first fully homomorphic scheme was proposed by [Gen09], which removes the limitations of HE applicability, by allowing the evaluation of arbitrary circuits. Essentially, operations on somewhat homomorphic encryption schemes incrementally add noise to the ciphertext, and from some threshold onwards, it is no longer possible to decrypt the data. Gentry's scheme proposes a *bootstrapping* operation allowing for the decrypt operation to also be performed homomorphically. Consider this example for an addition gate. Private inputs $\mathsf{msg}_1, \mathsf{msg}_2$ are first encrypted

$$\mathsf{FHE.Enc}(\mathsf{pk}_A, \mathsf{msg}_1)$$
$$\mathsf{FHE.Enc}(\mathsf{pk}_A, \mathsf{msg}_2)$$

and then all values and decryption keys are encrypted again with another key pair

$$\mathsf{FHE.Enc}(\mathsf{pk}_B, \mathsf{FHE.Enc}(\mathsf{sk}_A, \mathsf{msg}_1))$$
$$\mathsf{FHE.Enc}(\mathsf{pk}_B, \mathsf{FHE.Enc}(\mathsf{sk}_A, \mathsf{msg}_2))$$
$$\mathsf{FHE.Enc}(\mathsf{pk}_B, \mathsf{sk}_A)$$

by allowing for inner decryptions, fully homomorphic encryption (FHE) can be used to decrypt the inner values (given encrypted sk_A and $\mathsf{msg}_1, \mathsf{msg}_2$) and compute the gate. This produces $\mathsf{FHE.Enc}(\mathsf{pk}_B, \mathsf{msg}_1 + \mathsf{msg}_2)$.

Gentry's implementation is discussed in [GH11] to have several downsides to direct practical application, such as the size of public keys (going up to 2.2 GBs), the key generation (up to 2.2 hours) and the re-encryption (up to 31 minutes). Research progress on FHE has improved bootstrapping performance [DM15] and leveraged weaker security assumptions to remove the bootstrapping stage [BGV14], but significant performance improvements will be necessary before FHE schemes can be applied as real-world solutions.

Commitment schemes

Bit commitment schemes [Nao91] consider a two-party P_1, P_2 setting where P_1 wants to commit a bit b to P_2, to be revealed later. A bit commitment protocol consists of two stages: the commit stage, where P_1 sends some information d that represents b, and the revealing stage, where P_1 sends additional information that allows P_2 to know b. As such, this protocol can be seen as a scheme $\Phi = \{\mathsf{Commit}, \mathsf{Reveal}\}$, where the commit stage

$$d \leftarrow\!\!\$\ \mathsf{Commit}(1^\lambda, b)$$

takes security parameter λ and the private bit b from P_1 and produces commitment d for P_2, and the reveal stage

$$(b\ /\ \perp) \leftarrow \mathsf{Reveal}(b, d)$$

takes private bit b from P_1 and commitment d from P_2 and produces either the revealed bit b or a failure symbol $\perp$. In particular, this protocol must ensure that the commitments are both hiding and committing.

- *Hiding*, which requires for P_2 to be unable to infer b from d. More specifically, it requires for P_2 to be unable to guess b with probability greater than $\frac{1}{2} + \mu(\lambda)$, for security parameter λ and negligible function μ.
- *Committing*, which requires for P_1 to be unable to reveal a different value $b' \neq b$. More specifically, it requires that, after producing d from b, verification upon revealing $b' \neq b$ succeeds with probability $\mu(\lambda)$, for security parameter λ and negligible function μ.

Bit commitments can be naturally extended to schemes for committing a set of bits, and can thus be used to commit sensitive data with variable size, such as inputs or randomness. Commitment schemes are a central component to achieve zero knowledge proofs [GMW86, BM84, BCK+14, BKLP15, BSCG+15] and as a building block to achieve several multiparty protocols [CDVdG87, GMW87, LP07, Can01, CLOS02].

Zero-knowledge proofs

The presented definitions follow the ones discussed in [BR05, BSCTV14b]. A zero knowledge protocol considers two parties, a *prover*, which we denote as P_1, and a *verifier*, which we denote as P_2. P_1 has input x, w, P_2 has input x, z, and the goal is for P_1 to convince P_2 that $z = F(x, w)$. As such, a protocol for zero-knowledge proofs can be denoted as $\mathsf{T}/\mathsf{F} \leftarrow\!\!\$\ ZK(F, x, z, w)$, receiving the shared input x, shared output z, function F and the witness w for $z = F(x, w)$, and producing T for successful validations and F otherwise.

Protocols for zero-knowledge proofs require for the fulfilment of three properties: completeness, soundness and zero-knowledge.

- *Completeness* requires for P_2 to be convinced whenever $z = F(x, w)$. More specifically, it requires that, if $z = F(x, w)$, then P_2 returns T with probability 1.
- *Soundness* requires P_2 to be unconvinced whenever $z \neq F(x, w)$ with overwhelming probability. More specifically, it requires that, if $z \neq F(x, w)$, then P_2 returns T with probability $\mu(\lambda)$, for security parameter λ and negligible function μ.
- *Zero-knowledge* requires for the execution of the protocol to leak no information regarding w, other than what is leaked by z. More specifically, it requires the existence of some simulator $\mathcal{S}$ such that the view (the set of messages exchanged between the two participants) produced by the interaction between $P_1(x, w)$ and $P_2(x, z)$ is only distinguishable from the one produced by $\mathcal{S}(x, z)$ with probability $\mu(\lambda)$, for security parameter λ and negligible function μ.

Applications for Zero-knowledge proofs include program verification [BSCG$^+$13, BCI$^+$13, BSCTV14a, WB15], functional encryption [GGH$^+$16], e-cash [MGGR13], deniable encryption [SW14] and multiparty computation [CDVdG87, GMW87].

3.2 Passively secure multiparty computation

A major factor defining the applicability of MPC solutions is the adversary considered. We begin by presenting the MPC protocols that consider weaker (passive) adversaries, and give a high-level perspective on how they fare in real-world applications.

Yao's protocol

Yao's two-party computation protocol [Yao86] considers participants P_1 with private input x_1 and P_2 with private input x_2 that wish to compute a function f securely, which is represented as a binary circuit. In its simpler version, only P_2 is expected to receive the computation

output. To achieve this goal, the protocol relies on two central components: a garbling circuit scheme $\mathcal{G}$ and a 1-out-of-2 OT protocol OT.

P_1 will play the role of the *circuit generator* and P_2 will play the role of the *circuit evaluator*. At a high level, the protocol executes as follows.

- P_1 runs the circuit generation procedure $(F, e, d) \leftarrow\!\!\$\ \mathsf{Gb}(1^\lambda, f)$ and computes the tokens associated with its own private input $X_1^{x_1} \leftarrow \mathsf{En}(e, x_1)$.
- The garbled circuit, the decoding function, and P_1's private input tokens $\{F, d, X_1\}$ are securely sent to P_2.
- Both participants execute OT protocols $\mathsf{OT}(X_2^0, X_2^1, x_2) = (\epsilon, X_2^{X_2})$ for P_2 to receive all input tokens associated with its private input.
- P_2 can now evaluate the garbled circuit to produce the output token wires $Y \leftarrow \mathsf{Ev}(F, (X_1^{x_1}, X_2^{x_2}))$ and recover the computation output $y \leftarrow \mathsf{De}(d, Y)$.

The presented protocol is demonstrated to be secure against passive adversaries. Intuitively, for adversaries complying with the protocol specification, the guarantees achieved by garbling schemes ensure that P_2 is unable to retrieve any information regarding x_1 that could not be inferred by x_2 and y, and the guarantees given by the OT scheme ensure that P_1 recovers no information with respect to x_2.

In this regard, [PSSW09] was one of the pioneer contributions presenting reasonable results for Two-party Yao, refining the implementation on [LPS08] and presenting a set of ground breaking results for AES computation speeds, such as $7s$ for *passive* adversaries, $95s$ for *covert* adversaries and $1148s$ for *active* adversaries.

Yao's protocol is particularly effective in settings where applications can be deployed over two computing parties, such as PRACTICE's use cases [BMS+15] Privacy Preserving Genome-Wide Association Studies between Biobanks and Location Sharing with Nearby Contacts. While Yao's approach is often used in the two-party computation setting, [BMR90, BDNP08] have also proposed how to extend this protocol to the general MPC setting, which is particularly relevant since the majority of settings fundamentally require computation to be performed over an arbitrary number of participants.

Secret sharing based protocols

There are various approaches that rely on partially homomorphic secret sharing schemes. Sharemind [Bog13, BLW08, BNTW12] is a secure service platform for data collection and analysis that has had some practical applicability cases, and serves as an example for a passively secure MPC protocol based on secret sharing techniques.

Sharemind's deployment model considers an interplay between three kinds of protocol participants. *Input parties* can be any number of participants, and hold the private inputs $x^1, \ldots, x^n$. *Result parties* can also be any number of participants, and are expected to receive the computation result y. *Computing parties* are always fixed to three participants, and will be in charge of receiving shared values from input parties, executing the secure computation, and forwarding the resulting shares to the result parties.

The protocol makes use of a 3-party additive secret sharing scheme in the ring of 32-bit integers, i.e. a secret $s \in \mathcal{Z}_{2^{32}}$ is split into three random shares $s_1, s_2, s_3 \in \mathcal{Z}_{2^{32}}$ such that $s_1 + s_2 + s_3 = s \pmod{2^{32}}$, and its execution is as follows. The input parties take their private inputs $\{x^1, \ldots, x^n\}$ and secret share them to

$$\{x_1^1, x_2^1, x_3^1, \ldots, x_1^n, x_2^n, x_3^n\}$$

These shares are then divided accordingly and securely sent to the three computing parties, denoted here as C_1, C_2 and C_3

$$C_1 : \{x_1^1, \ldots, x_1^n\}$$
$$C_2 : \{x_2^1, \ldots, x_2^n\}$$
$$C_3 : \{x_3^1, \ldots, x_3^n\}$$

Sharemind then describes several protocols for performing basic arithmetic and comparison of integers, which in turn can be used to perform more complex data processing applications. After all operations are performed in the computing parties, the output shares $\{y_1, y_2, y_3\}$ are sent to the output parties, to be reconstructed into the actual computation output

$$y \leftarrow y_1 + y_2 + y_3 \pmod{2^{32}}$$

The computation protocols are presented as being provably secure in the passive adversarial model with no more than one passively corrupted party. Originally, Sharemind's definition of security followed the universal composability, but did not present a full universal composability proof for its algorithms. More recently, the security of Sharemind was formalized using reactive simulatability and under a weaker notion of privacy [BLLP14].

Sharemind has been widely used as a solution for use cases such as Genome Data Analysis [BKLS16a], Tax Fraud Detection [BoSV16] or General Statistical Analysis [BKLS16b]. Sharemind-based implementations have been demonstrated in 2013 [LTW13] to implement AES with execution times of $323ms$ and $0.29ms$ (amortised), further improved in [T$^+$16] with execution times of $223ms$ and $0.04ms$ (amortised).

Combined approaches

In general, Yao's protocol benefits from having a *constant* number of communication rounds, which makes the approach typically preferential when deployment over networks with high

latency is considered. This technique allows for offloading a considerable ratio of the heavy cryptographic work to a preparation stage, however its communication is not as efficient as its secret sharing scheme's counterparts, where some operations can be performed locally. Secret sharing schemes benefit from natural scalability to an arbitrary number of participants. The operation for the secret sharing of values tends to be very efficient, which allows for the bulk of the computation and communication to be off-loaded to potentially untrusted participants. This is a natural fit to real-world deployments where participants with inputs are not necessarily the ones doing the actual computation, such as users off-loading secure computation to a cloud provider.

Both approaches have their strengths and weaknesses, thus their optimal application is dependent on the use case scenario and its inherent computational and communicational model. On the opposite end of highly specific applications, such as those improving AES execution time on MPC, frameworks TASTY [HSS+10] and ABY [DSZ15] focus on providing the software developer with a passively-secure MPC tool that can execute general functions using both garbled circuits and secret sharing homomorphic computations. These approaches have considerable potential for real-world MPC, since they can be instantiated with the best implementations of each primitive, thus enabling applications to employ different mechanisms according to different deployment settings.

TASTY (Tool for Automating Secure Two-partY computations) is a tool suite addressing secure two-party computation in the passive adversary model. *TASTY*'s goal is to prioritize the online phase for evaluating a function f, assuming a prior setup phase for the execution. To achieve the desired speed-up, *TASTY* allows the compiling and evaluation of functions using garbled circuits and (additively) homomorphic encryption schemes, e.g. Paillier, at the same time. *TASTY* can be extended to employ other primitives, such as fully-homomorphic encryption [Gen10]. Prior to *TASTY*, compilers for secure function evaluation were restricted to compiling protocols using either garbled circuits or homomorphic encryption schemes. *TASTY* implements a hybrid approach, combining garbled circuits and homomorphic encryption, as described in [KSS10, KSS13b]. Specifically, for each function f to be evaluated, the programmer can define which parts of f should be computed by garbled circuits, and which parts should be computed by homomorphic encryption. On one hand, arithmetic circuits cannot efficiently encode non-linear functions like comparisons, which includes Yao's millionaire problem and XORs [KS08]. On the other hand, Boolean circuits cannot efficiently express arithmetic functions like multiplication. The combination of both approaches allows for experimentally supported computation speedup [HSS+10, KSS09].

ABY is a framework for secure multiparty computation that is highly focused on maximizing performance [DSZ15]. In a intuitively similar approach to *TASTY*, the idea is to combine several schemes for secure multiparty computation, such as Arithmetic sharing, Boolean sharing and Yao's garbled circuits (hence the name), to take advantage of the best aspects

of each of these approaches. The original proposal is focused on allowing for two-party computation, however there is no strict limitation on the actual number of participants. The framework is also prepared to handle reactive computations, thus allowing the servers performing computations to maintain secure state information among a sequence of executions. Arithmetic sharing encodes l-bit values additively in a ring $\mathbb{Z}_{2^l}$, with a protocol inspired by [ABL+04, KSS14]. Boolean sharing uses a XOR-based secret sharing scheme, and its circuits are evaluated by the GMW protocol. Yao sharing is made using the original protocol approach [Yao82]. To support and combine all these different secure computation methods, *ABY* proposes methods for converting one type of sharing to another. These represent operations such as (but not limited to) *Yao to Boolean Sharing*, *Boolean to Arithmetic Sharing* or *Arithmetic to Yao Sharing*.

Under the scope of performance, the experimental validation in [DSZ15] presents an overview of the execution times for the *ABY* framework. The tool was evaluated both in a local setting and in a cloud setting, to highlight potential computation bottlenecks and communication bottlenecks, and considering applications such as Modular exponentiation, Private Set Intersection and Biometric Matching. Measurement results strongly substantiate the intuitive notion that different use cases can benefit from different approaches, e.g. sometimes boolean sharing is better, sometimes garbled circuits are better, sometimes the combination of the two is the best choice, and further solidifies the potential of combining techniques in software-based MPC.

3.3 Actively secure multiparty computation

On the other side of the spectrum, we now present actively secure solutions for MPC. These enable for deployments on a wider variety of use cases, however good performance results only tend to arise from highly specialized approaches rather than their more flexible passively secure counterparts.

GMW compiler

The usage of secure computation for real-world use cases often involves deployment over highly untrustworthy participants, so active security is a common concern. In [Gol09], Goldreich proposes a compiler – the GMW compiler – allowing for any passively secure MPC protocol to be converted into an actively secure one. The approach makes use of two main components: input commitment and zero-knowledge proofs, and its intuition is to take a passively secure protocol and have every step (i.e. every exchanged message) be accompanied by a zero-knowledge proof verifying its legitimacy both with respect to the protocol description, and to the established inputs.

More concretely, consider n participants $\{P_1, \ldots, P_n\}$ with private inputs $\{x_1, \ldots, x_n\}$ and let $\mathsf{msg} \leftarrow \pi(i, \mathsf{msg}_i^k, r_i, t_i^k)$ be an n-party passively secure protocol, taking as inputs party index i, the k'th message msg_i^k, the party's randomness r_i, and the list of all messages seen up to the k'th message by the participant t_i^k. Let $\Phi = \{\mathsf{Commit}, \mathsf{Reveal}\}$ be a commitment scheme, and $ZK(F, x, z, w)$ be a protocol for zero-knowledge proofs. The GMW compiler tweaks the execution of π to perform as follows.

- First, all P_i must force one another to execute using uniformly generated randomness, while maintaining secrecy of each individual randomness. As such, all P_i will generate randomness values $\{r_i^1, \ldots, r_i^n\}$ and send r_i^j to all j. Afterwards, P_i computes its randomness from all values received and the locally generated value.

$$r_i \leftarrow \bigoplus_{j=1}^{n} r_i^j$$

- Then, all P_i prepare commitments of their inputs $dx_i \leftarrow_\$ \mathsf{Commit}(1^\lambda, x_i)$ and of their locally generated randomness $dr_i \leftarrow_\$ \mathsf{Commit}(1^\lambda, r_i^i)$, and send $\{dx_i, dr_i\}$ to all other participants.

- Finally, all messages exchanged by the protocol $\mathsf{msg} \leftarrow \pi(i, \mathsf{msg}_i^k, r_i, t_i^k)$ are accompanied by a zero-knowledge proof $ZK(F, x, z, w)$ where w is the witness opening the commitments (x_i, r_i^i), z is the produced message msg, x is the description of the protocol π and F is the decision problem "msg is consistent with π and the transcript $\{t_i^1, \ldots, t_i^k\}$, given the input and randomness commitments (dx_i, dr_i)".

The employed zero-knowledge proofs ensure that the adversary cannot deviate from the protocol description in π, enforce input consistency throughout the execution, and prevent the adversaries to use "convenient" randomness in any step of the protocol execution. This technique is sufficiently generic to be applied to any passively secure MPC protocol, and presents an important result in the research of practical MPC solutions.

However, one downside of the GMW compiler is the requirement of using zero-knowledge proofs. These tend to be very expensive, as they revolve around reducing decision problems to NP-Complete problems such as the Boolean Satisfiability Problem, Hamiltonian Cycle or the 3-Coloring, and often introduce large polynomial factors. As such, practical MPC deployments most commonly resort to other techniques for tackling with active adversaries.

Cut-and-choose

In [LP07], Pinkas and Lindell introduced an elegant protocol allowing for GC-based secure computation under the active adversary model named Cut-and-Choose. This technique is focused on preventing the circuit generator (P_1) from garbling a circuit different from the

agreed upon function, by having P_1 generate several garbled circuits to be sent to P_2, and allowing P_2 to select half of them for validation.

More concretely, cut-and-choose takes a statistical security parameter ρ which specifies how many garbled circuits will be used. This will define the error probability incurred by the cut-and-choose technique. At a high-level, the cut-and-choose protocol assumes participants P_1, P_2 with inputs x_1, x_2 and wishing to compute $\mathsf{F}(x_1, x_2)$, and executes as follows.

- P_1 commits to ρ different garbled circuits of F, with the additional tweak that each input for P_2 is replaced by a exclusive-or gate consisting of ρ new input wires for P_2. P_1 also commits to garbled values corresponding to the circuits' input wires.
- For all P_2 input bits, P_1, P_2 run 1-out-of-2 OT protocols, so that P_2 can obtain input wires corresponding to x_2.
- P_1, P_2 run a coin-tossing protocol to establish fair randomness to define which commitments and which garbled circuits will be verified.
- P_1 must then open the selected garbled circuits and inputs, and P_2 verifies their correctness. If any inconsistency is detected, the protocol aborts.
- P_1 can now send the garbled values corresponding to x_1, to be evaluated on the remaining unopened circuits.
- P_2 can now evaluate all unopened circuits and present the majority output value as the result of $\mathsf{F}(x_1, x_2)$.

The cost of cut-and-choose techniques is highly associated with the number of generated garbled circuits. Originally ([LP07]), restricting an active adversary to a $2^{-\rho}$ chance of successful attack would require 17ρ circuits, however this overhead was gradually reduced to 3ρ [LP12, hSas11, S$^+$13] and ρ [Lin16]. Recent developments in [Bra13, CAGAA$^+$13] explore the possibility of using an approach named "forge-and-lose", where cheating requires for all circuits (checked and evaluated) to be correct. This uses a specific flavour of commitments leaking some auxiliary information if two garbled circuits differ, allowing for P_2 to recover x_1 if P_1 has attempted to cheat (thus enabling the local computation of the legitimate output $\mathsf{F}(x_1, x_2)$).

Authenticated shares

The goal of achieving active security has also been approached in [NNOB12, KRW17], which use information-theoretic message authentication codes (IT-MACs) to authenticate the joint evaluation of a garbled circuit. In this setting, active corruptions may occur as both

participants can replace values on internal wires to disrupt the functionality output or force information leakage.

Security is achieved by relying on homomorphic MACs associated with all bits of shares held by both players, thus enforcing a verifiable behaviour on both participants' computations. At a high-level, this mechanism works as follows. One party (say, P_1) plays the role of a *key holder*, and the other (say, P_2) is the *MAC holder*. For a security parameter λ, P_1 holds a global key $\mathsf{key} \in \{0,1\}^\lambda$ and P_2 holds secret bits (x_0, x_1). For each of these bits, P_1 holds a corresponding key $(K_0, K_1) \in \{0,1\}^\lambda$, which are used to validate the corresponding MACs M_0, M_1 for each of these values

$$M_0 : K^0 \oplus x^0 \cdot \mathsf{key}$$
$$M_1 : K^1 \oplus x^1 \cdot \mathsf{key}$$

These MACs benefit from the same homomorphic properties as shares in the GMW protocol

$$M_0 \oplus M_1 = (K_0 \oplus K_1) \oplus (x_0 \oplus x_1) \cdot \mathsf{key}$$

Thus allowing for local computation over shares to also be reflected in their associated MACs. Whenever the shares are to be reconstructed to produce the original values, the associated MACs are also gathered, allowing for a verification of the produced result. Note that, in [NNOB12], both P_1 and P_2 will have several secret bits over which computation is to be performed, and both can behave maliciously. As such, P_1 will be the *key holder* for validating the secret values computed by P_2 (as exemplified), and P_2 will be the *key holder* for validating the secret values computed by P_1.

The adaptation of the same mechanism to the MPC setting is suggested in [WRK], retaining the homomorphic properties of the original protocol. IT-MACs are also a central approach to works such as [DPSZ12, DKL$^+$13], where MACs are used to authenticate shared values, rather than the shares themselves. This mechanism for authentication works very similarly to the MPC extension proposed in [WRK], considering a global shared MAC key such that all participants can locally compute MAC values, and use its homomorphic properties to propagate over several operations on the authenticated shared value.

Practical applicability of the state-of-the-art

Execution time is one of (if not the) most significant factor in the transition of MPC from a purely theoretical area to a feasible approach for real-world deployments. From the pioneer proposal in [PSSW09] onwards, considerable improvements have been reported. In 2011, [hSas11] proposed results for $192s$ computation of AES, considerably reducing communication and computation overheads by improving cut-and-choose techniques using *claw-free collections* [GK96] and the approach of [CC$^+$08] to reduce complexity size for the

output validation step. Yet another meaningful push was made on 2014 [FJN14] presenting an implementation with $456ms$ for AES computation, improving the "forge-and-lose" approach and parallelizing its execution using the same instruction, multiple data (SIMD) framework using a GPU.

Recent 2016 [RR16] and 2017 [WMK17] works have achieved values for $6.4ms$ (amortised) and $65ms$ (non-amortised) for AES, respectively. [RR16] builds upon the dual-execution protocol proposed in [MF06], where both participants play both roles in the GC evaluation and converge to the secure computation output, and reduce its computational overhead by developing a novel input consistency technique for this specific setting. [WMK17] makes use of Lindell's construction in [Lin16] requiring ρ garbled circuits for statistical security parameter ρ, and builds upon the general idea of [AMPR14] for replacing public-key cryptographic operations with symmetric cryptographic operations, asymptotically reducing the number of public-key operations required in the circuit evaluation. Also in 2017, [KRW17] has produced results for AES computation in $0.93ms$ (as reported in [KOR+17]), adopting and improving upon the IT-MAC approach for the design of authenticated garbled circuits. This approach by Katz et al. has also been extended to the MPC setting in [WRK], allowing for a more flexible corruption setting where up to $n-1$ participants can be controlled by the adversary.

SPDZ is another approach to the problem of secure computation [DPSZ12, DKL+13], considering active adversaries and protocols running on n participants with up to $n-1$ corruptions. Security relies on the usage of IT-MACs for authentication of shared values, and performance takes advantage of a somewhat homomorphic encryption scheme (proposed in [DPSZ12]) used to generate beaver triples to enhance processing during the online phase. In [KSS13a], the efficiency of SPDZ to compute a single AES execution was measured in a deployment using SIMD to obtain a value of $12ms$.

Other secret-sharing based MPC solutions have presented good results, such as [FLNW17] and [KOR+17]. [FLNW17] extends the work in [AFL+16] to sustain active adversaries by having cut-and-choose be used to validate beaver triple generation, and is (theoretically) expected to outperform existent execution such as [RR16] requiring communication of roughly 100 times higher than the proposed protocol. [KOR+17] also takes the lookup table approach, improving on the TinyTable protocol [DNNR16], generalising the approach to the MPC setting and refining the mechanism required for the preparation phase, achieving $0.93ms$ per AES computation.

The high amount of recent publications in the subject substantiates the considerable effort by the research community to find feasible actively secure MPC solutions. However, note that the fastest existing solutions take on a highly specialized approach to the problem, often following implementation decisions such as lookup tables, which are not scalable to considerably more complex problems.

Chapter 4

Trusted hardware in cryptography

Work that looks at provable security for realistic protocols using trusted hardware components has been developed around approaches using the Trusted Platform Module [BCC04, SRC07, BCL08, GT07, FNRT14]. However, the functionality and efficiency of these protocols offered by TPM makes them more suitable for ensuring integrity of programs right before execution, rather than the run-time guarantees that modern trusted hardware provide. In this chapter, we present the novel trusted hardware technologies of Intel SGX and ARM TrustZone, as well as an analysis of similar technologies focused on ensuring runtime guarantees, relevant research relying on these technologies to provide secure and efficient protocols, and an overview of current side-channel attacks to these specialized hardware technologies.

4.1 Software Guard Extensions

An SGX-enabled processor is equipped to protect the integrity and confidentiality of the computation inside the enclave, by isolating its code and data from the outside environment. This includes the operating system and hypervisor, as well as any hardware devices attached to the system bus. These environments have access to a special SGX call for producing a proof of integrity regarding the code running within the enclave. The proof itself is a cryptographic signature that certifies the hash of the enclave contents (namely the code and memory). This setting does not prevent the remote computer to run any specific software within enclaves, but instead allows for the user to reject any result produced by an enclave whose contents do not match the expected value.

Enclaves

In an SGX-enabled hardware, a subset of memory is reserved as *Processor Reserved Memory* (PRM). The CPU is responsible to protect this memory structure from all external (non-

enclave) memory access. The PRM manages the enclave page cache, storing enclave-related information such as code and data. The system software is in charge of assigning these enclave pages to enclaves, and the CPU will make sure that each enclave page corresponds to exactly one enclave. SGX provides a set of special CPU instructions allowing for the management of enclaves, which will now be concisely described.

Enclave creation begins with ECREATE, establishing the initial environment within the protected range of addresses, and allocating an associated data structure for the enclave on the PRM. This transits into an initially valid enclave construction. As the enclave is built, the instruction EADD allows the allocation of additional memory pages to the enclave. At each page, EEXTEND is used to measure allocated space. If all added pages have been measured, then the enclave is ready for initialization with EINIT. According to the information within the enclave, this instruction establishes fixed valid entry points for the enclave.

When the enclave is successfully initialized, it enters a locked state, preventing changes on enclave memory pages. From this point onwards, the user may enter the enclave under program control with EENTER, specifying a valid address as entry point. Within the program mode, the enclave is restricted from performing any instruction listed on a pre-established set of illegal commands, resulting in an error. From there, it is possible to obtain cryptographic keys with EGETKEY and generate reports for other enclaves with EREPORT. These reports can be used for enclaves to authenticate messages among other enclaves within the same platform.

The enclave is terminated via EEXIT. If something makes the enclave halt (either an expected or unexpected occurrence), the *Asynchronous Enclave Exit* is triggered. This saves the enclave's state using cryptographic techniques, allowing to re-enter the enclave using ERESUME. From the moment an enclave is locked, it is possible to execute management commands, such as evictions blocks or loads.

In order to provide inter-platform enclave attestation, SGX-enabled hardware also includes an Enhanced Privacy ID (EPID) scheme [BL11] that is used by a special enclave called *quoting enclave* for signing enclave attestations. EPID is a group signature scheme allowing a platform to construct signatures without uniquely identifying the actual platform that has produced it. Only the quoting enclave has access to this EPID key, which is bound to the version of the underlying firmware. The mechanism for *remote attestation* is proposed in [AGJS13], can be depicted in Figure 4.1 (from the same paper) and is described as follows:

1. The remote machine establishes communication with an SGX-enabled platform and issues a challenge to validate the machine as running the necessary components inside an enclave, including a nonce for liveness purposes.

2. The application sends to the enclave the identity of the quoting enclave and the remote challenge.

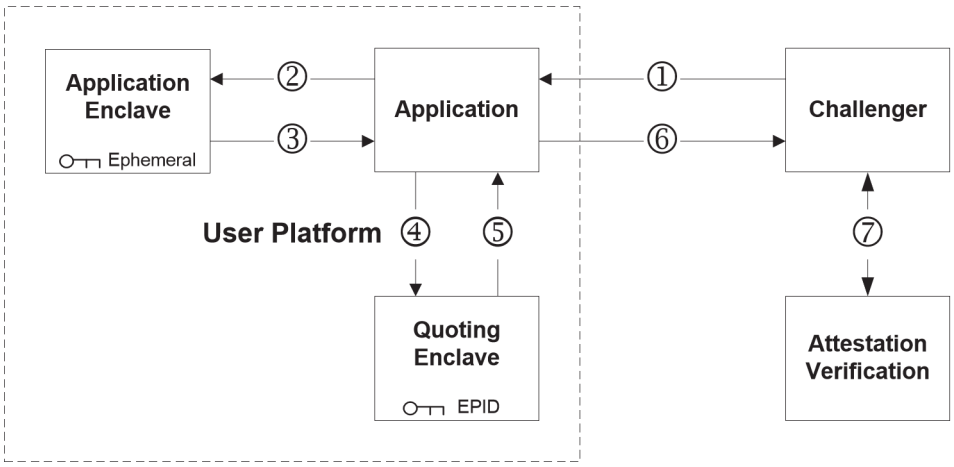


Figure 4.1: Process for remote attestation

3. The enclave generates a report (using the described instruction EREPORT) comprising the enclave contents and the challenge given to it, and returns it to the application.

4. The application simply forwards the received data structure to the special quoting enclave.

5. The quoting enclave verifies the authenticity of the received report (using EGETKEY to retrieve the associated key) and signs it with its unique EPID key. This produces what is called a *quote*, which is returned to the application.

6. The application forwards the quote to the remote machine.

7. The challenger uses a EPID public key to validate the signature included in the received quote. It can now verify the integrity of the signed data and check the response for the challenge proposed in (1).

The work developed during this thesis is mainly focused on formalizing and proving security guarantees that can be provided by schemes such as this quoting mechanism, that allows for enclaves to prove to external users that they are running according to a specific code and contents. This was the first modular security analysis of these hardware-based mechanisms, so the subsequent abstraction and formal definitions represent the initial efforts in precisely specifying the security models and definitions that allow for the implementation of provably secure protocols relying on trusted hardware providing isolated execution environments, such as SGX.

4.2 TrustZone

ARM TrustZone [AF04] is a collection of hardware modules employed to allow for a programmable environment where applications can be run with confidentiality and integrity guarantees from other software running within the same machine. This is achieved by enforcing a strict partitioning of all system resources between a Secure world, hosting secure memory and containers, and a Normal world, hosting the standard software stack.

TrustZone's CPU core is equipped with two page table base registers, providing separate address translation units for the Secure and Normal worlds. The addresses include an additional *secure bit* to establish if the contents belong to the Normal or the Secure world. Secure containers must also implement a monitor to perform context switches between the two worlds and to handle hardware exceptions, forcing the system to return to the normal world. TrustZone manages context by employing an extended bus architecture (AMBA3 AXI™) with an extra control signal for secure (low) or non-secure (high) write/read transactions, ensuring that security separation is not violated.

The development of applications to be run within the Secure world software stack can be done in a variety of ways, as suggested in [ARM09]. The simplest approach is to load a code library into the Secure world, whose scheduling is entirely controlled (activated) by the OS in the Normal world. This allows for software developers to benefit from TrustZone isolation guarantees when the target application is simpler and does not require any additional security/management mechanisms. Another approach is to have a dedicated operative system in the Secure world, which is a complex design that can independently manage several applications running within controlled environments. This enables the usage of a secure application service that initializes additional cryptographic components, and that uses the processor memory management unit to separate secure applications into resource-isolated sandboxes.

Hardware initialization and attestation

TrustZone is a system primarily designed for establishing strict isolation between Normal world applications and Secure world applications. This enables for code to be run on environments whose resources cannot be read or tampered with by external (potentially malicious) software within the same machine, but is insufficient to enable inter-platform attestation. Work in [ZZH+14] rigorously describes how TrustZone-enabled hardware is initialized to make use of a secure root-of-trust for application development. More specifically, the goal is to construct a building block in on-chip memory providing a unique device key, to generate cryptographic material for execution attestation, and a random seed, necessary to implement a secure random number generator for the secure OS.

The generation of a unique device key is achieved by combining an SRAM Physical Unclonable Function (PUFs) [Sol] and a fuzzy extractor based on BCH code [MZ]. PUFs ensure robustness, meaning they produce the same responses to the same challenges (modulo some noise), and uniqueness, meaning they produce independent responses to different challenges. These can then be combined with fuzzy extractors to generate and store keys without having to write keys on secure non-volatile memory. More concretely, the SRAM PUF will provide a start-up value r, which can be seen as a string of values of SRAM cells. Initially, the fuzzy extractor uses r on generation algorithm Gen to produce a key k and some non-sensitive helper data H, which is stored on non-volatile data. Whenever the device must reconstruct the device key, it now gets a noisy response r' instead of the original r. To reconstruct the device key, the fuzzy extractor runs Rep with r' and helper data H to reconstruct the original k. This has the added benefit of enabling easy key updates, since Gen can be run again to obtain the updated key k' and new helper data H'.

To allow for secure randomness to be generated, [ZZH+14] makes use of the entropy extractor proposed in [DPR+13] to condense the entropy of the noisy initial SRAM values and obtain a random seed, which can then be used on a random number generator whenever necessary.

The proposed system is thus comprised of a generate procedure, to be securely executed upon the hardware manufacturing, and of a building block, for whenever the same device is initialized from then onwards. Figures 4.2 and 4.3 depict an overview of these procedures, which are detailed as follows.

- Generate procedure - The specific key generation procedure feeds a randomly selected primary seed PS to both a key derivation function KDF to obtain public/private keys $(\mathsf{pk}, \mathsf{sk})$ and to a deterministic key generation algorithm KG to obtain symmetric key k. sk is then used to issue a certificate C_{pk}, which can then be employed as cryptographic material for remote attestation, whereas k is used for encrypting the image of the secure OS. This PS is also fed to the BCH encoder $C \leftarrow \mathsf{BCH}_{\mathsf{enc}}(PS)$, and the initial values for the SRAM cells r are used to create the helper data necessary for key recovery $H \leftarrow C \oplus r$.

- Building block - The noisy initial SRAM cell values r' are used to recover a noisy BCH code $C' \leftarrow r' \oplus H$, which can then be decoded to obtain the original value $PS \leftarrow \mathsf{BCH}_{\mathsf{dec}}(C')$. Given the original PS, retrieving the keys is analogous to the previous step. Every time a random value is required, one can use the entropy extractor on the SRAM cell values r' to obtain a (fresh) random seed.

This initialization is then used to construct primitives for sealing and unsealing, which binds secure data with respect to both the platform and the application that is being run. As a specific case, this allows for the implementation of a service for inter-platform

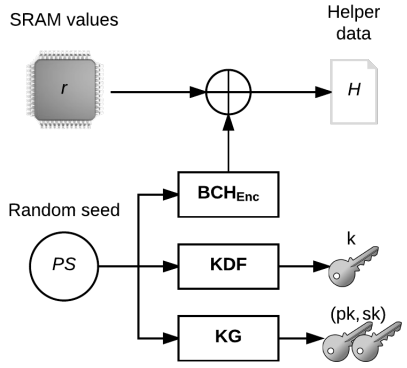


Figure 4.2: Generate procedure

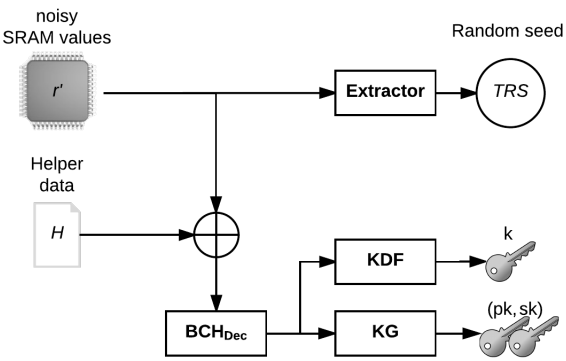


Figure 4.3: Building block procedure

attestation using the device key, similarly to the behaviour of the Signing Enclave described in Section 4.1.

4.3 Other approaches to hardware attestation

Trusted Execution Technology (TXT) [Gra09] is another approach by Intel, using the TPM's software attestation model and auxiliary tamper-resistant chip, but reducing the software inside the secure container to a virtual machine hosted by the CPU's hardware virtualisation features. An initialization authenticated code module (SINIT ACM) allows for performing system resets, and enables for software to have exclusive control over computational resources while it is active. Program execution within trusted environments requires prior measurement of the code. These values are computed using cryptographic hashes (SHA-2 in the TPM v.2.0 and onwards) held in Platform Configuration Registers, special registers used for integrity verification. However, contrary to SGX, TXT does not implement DRAM encryption, and is vulnerable to physical DRAM tampering (as is the case with similar TPM-based designs).

Alternatively, the IBM 4765 secure coprocessor [Yee94] encapsulates an entire computer system, including CPU, caches, DRAM and IO controller within a tamper-resistant environment. The focus is to provide a general-purpose computing environment withstanding logical and physical attacks, while ensuring that it is possible to remotely distinguish between a real instance of the 4765 secure coprocessor and an impersonator. In particular, the secure coprocessor destroys the secret stored as soon as it detects a tampering attempt, via an array of sensors. This system securely stores its attestation key in battery-backed memory accessible only to the secure coprocessor, responsible for measuring and loading system software, as well as provide software attestation services for applications loaded within. This hardware has shown to provide good security properties, however these tamper-resistant enclosures tend to be very expensive compared to the expected cost of computer systems [And], which deters practical deployment of solutions in these environments.

Sanctum [CLD] provides strong software isolation and protection against software attacks inferring private information from memory access patters, given a small set of hardware changes. The overarching idea is to employ a system with protected memory access and context changes to protected CPU execution environment, and to allow for code signing with a measurement root and capabilities analogous to the Signing Enclave. With respect to enclave-dedicated DRAM regions, Sanctum presents a very versatile model, where the system boots up with all DRAM regions allocated to the OS, which can freely release regions to be re-assigned to enclaves. Performance evaluation of the system suggests low overheads, however it does not provide a direct comparison with other hardware for trusted execution. Sanctum provides full access to the security monitor (central component to the management of enclaves) and encourage security analysis. This approach is motivated for comparison with SGX, accused of having security properties that are convoluted to reason over, given lack of implementation details in publicly available documentation.

TrustLite [KSSV14] is a software architecture for hardware-enforced isolation focused on applicability to embedded devices. The architecture is based on a memory protection scheme, with an execution-aware memory protection unit, secure exception engine, and a secure loader. These components are employed to allow tasks to be designed and executed with particular security mechanisms, called *trustlets*. Trustlets must ensure: i.) *data isolation*, in the sense that no software can modify their code after hardware-enforced access control is established, ii.) *attestation*, which allows for local platform state validation without external influence, as well as a mechanism for trustlets to establish mutual verification and secure channels, and iii.) *secure peripheral access*, where trustlets have access to peripherals without external interference. i. and ii. are essential to ensure trusted execution, while iii. is central to motivational application scenarios requiring for secure user I/O [KJ01, FMSW11, ZGNM12]. Remote attestation is ensured by having the secure loader establish a chain of trust that will allow for trustlet contents to be digitally signed. Security of the system considers an adversary with full control over the OS and tasks running within the platform, but requires for the bootstrapping routine (and the initialized trustlets) to behave securely.

4.4 Protocols relying on trusted execution

VC^3 [SCF+15b] was one of the earliest paper publications employing SGX. A map-reduce for generic computations, relying on establishing secure channels with SGX machines and securely transmitting keys for decrypting and computing over protected data. A more general application for an IEE-enabled Functional Encryption system is described in [FVBG], leveraging IEEs to perform Secure Outsourcing of Computations, where an isolated environment manages keys, and responds to requests according to a specific functionality f to be run with security guarantees (private genome calculations being the use case). Along the same

lines [SAM+17] also proposes a system for the secure private genome analysis using SGX systems.

Performing two-party computation using SGX has also been proposed in [GMF+16]. The motivation of this work is to present the trade-off between efficiency with trusted hardware and non-reliance on trusted hardware, and proposes a method to convert two-party computation garbled circuits to programs running within SGX, and vice-versa. Experimental validation compares implementations of two-party computation garbled circuits against passive adversaries, active adversaries, and SGX-enabled two-party computation.

SGX has been employed in the design of novel block chain systems in [MHWK16]. This work makes use of Isolated Execution Environments to perform highly efficient validation of the block chain, introducing a IEE-specific primitive named "proof of luck", allowing for low-latency transaction validation, deterministic confirmation time and equitably distributed mining. Another system [ZEE+] was independently developed to also tackle with the inefficient software solutions to deal with block chains in untrusted environments by leveraging isolated execution environments such as Intel's SGX. A third system [ZCC+16], partially developed by the authors of the second, employs SGX-enabled block chains with the goal of providing an authenticated data feed for applications such as smart contracts. [LEPS16] also looks at block chains, but instead focuses on the establishment of payment channels, achieving higher throughput than software-reliant systems.

Managing and (partially) storing sensitive database indexes [FBB+17]. HardIDX is a hardware-based approach for efficient database indexing, leveraging Isolated Execution Environments to either load index structure ($B+$ trees, entirely or partially), and adequately respond to query requests. This system does not ensure protection against an attack based on access pattern recognition, which can itself be prevented by [TLP+16], which obfuscates access patterns (similarly to ORAM) using the same trusted hardware mechanisms. This system employs a "Carousel" method for an enclave to loop over database entries, hiding the memory accesses relevant to client requested queries.

The usage of SGX-enabled systems to improve performance maintaining security was explored in the context of anonymity networks (Tor) [KSH+15]. SGX allows for Tor nodes and clients to benefit from attacks compromising program integrity. Experimentation results measure CPU cycles with and without SGX, using the OpenSGX simulator. Another application for networking (P2P) was proposed in [JTM+], where the trusted environment of IEEs is employed to eliminate the advantage of a byzantine adversary, as well as to improve performance of reliable broadcasting and common random coin generation via SGX features. SGX technologies were also explored in the context of content-based routing while preserving privacy without significant performance overheads [PPFF17]. Experimental results include the comparison between computation over: i.) plain data, ii.) asymmetric scalar-product preserving encryption (ASPE) and iii) using SGX, where performance of SGX was con-

cluded to be comparable to the plain data implementation, and considerably superior to ASPE. Additionally, SGX was also explored in the context of secure network booting for Software-Defined Networking [PG16], which improves resource utilization, scalability and administration of cloud network infrastructures.

SGXIO [WW17] is a trusted path architecture for SGX, allowing for applications running in an untrusted OS to make use of trusted paths to generic I/O devices. This is achieved via the establishment of a secure channel between some user secure I/O device, and an enclave running the desired code.

TrustShadow [GLX+17] aims to ensure trusted execution environments for security-critical applications within the context of IoT devices using ARM TrustZone technology. This approach makes use of the resource partitioning in TrustZone to ensure protection of legacy operations from potentially untrusted OSes, by initializing "shadow" counterparts of applications running in the secure world of TrustZone. The performance of TrustShadow has been evaluated with micro-benchmarks (various Linux system services) and real-world applications (embedded web server). In the same context of IoT devices, C-FLAT [AAD+16] enables remote control-flow path attestation. This approach has been implemented in Raspberry Pi devices relying on ARM TrustZone hardware extensions to be a trust anchor for a pre-processing off-line stage where control-flow paths are statically analysed, measured and securely stored in a measurement database. C-FLAT is thus devised to protect a system from adversaries attempting to hijack control-flow on an application on runtime.

Attestation mechanisms for mobile devices with TrustZone can also be ensured while maintaining anonymity [YYQ+15]. The proposed system requires a one-time TrustZone activation, which loads a trust anchor component into the SRAM that must only be accessible via the secure world, following the methodology described in [ZZH+14]. This protects against adversaries attempting to attack the scheme either by pretending to be a honest participant with another device, or by manipulating the device (either via the OS of the TrustZone-enabled device, or by actively forcing device reboot). This approach was further explored in [YYZ+16], an anonymous e-payment protocol for mobile devices that makes use of secret keys derived from the trust anchor that DAA-TZ includes on SRAM.

Protection of mobile OS's against rootkit attacks is the focus of [GVJ14], which makes use of TrustZone as an introspection mechanism for tampering detection. The motivation here is that TrustZone, despite enabling a secure environment for application execution, does not ensure legitimate execution of sensitive instructions. This is achieved by having a secure pre-boot setup, where the secure world is initialized to have specific knowledge about the expected normal world kernel memory configuration.

A provable security approach is often absent in the aforementioned contributions, where security guarantees are usually limited to high-level descriptions. In this regard, [SCF+15b, FVBG, FBB+17, JTM+] are exceptions, which present application-specific security goals in

their respective settings, and a respective security proof for the proposed protocols.

4.5 Side-channel attacks

Work in [CD] presents a thorough analysis of the SGX platform. In this extensive report, several attack vectors against this technology are suggested, such as cache timing attacks, port attacks, bus tapping attacks, chip attacks, power analysis attacks, PCI express attacks, performance monitoring side-channels or address translation attacks.

Cache attacks are also presented in [MIE17, BMD+17]. Both these works employ the prime+probe cache monitoring attack [OST06], and demonstrate how this approach allows for extracting private keys for encryptions run within SGX with significantly better efficiently than on similarly exploited scenarios. Interestingly, the efficacy of these attacks is further enhanced by performing prime+probe in isolated co-located enclaves [SWG+17], which remain concealed from standard malware detection mechanisms. Similar cache timing attacks of prime+probe can also be efficiently performed on ARM TrustZone [ZSS+16, LGS+16] in mobile devices for the extraction of AES encryption keys used in the secure world, with existing countermeasures proposed in [AL17] by mitigating cache contention about shared memory and adjusting data path timings for non-detecting cache transition.

The protection of SGX against roll-back and forking attacks is discussed in [MKD+], where a single platform is considered insufficient to efficiently prevent these threats, so multiple processors assist each other to maintain execution consistency. This is achieved by the ROTE system, which maintains a distributed counter value that can be retrieved by restarting enclaves to maintain freshness. A similar approach is also proposed in [BCLK17], that instead considers a protocol that allows for mutually trusted clients to maintain an execution context. This execution context manages a monotonic counter used for protection against roll-back and forking attacks.

Page fault patterns in SGX can allow for the profiling of program executions, as explored in [SCNS15], within the context of commonly used crypto libraries such as OpenSSL and Libgcrypt. This work presents a high success rate in secret bit recovery, and proposes a solution to reduce overheads, with support from hardware changes. Fault attacks of IEE systems are explored in [BGN+16], where memory encryption is shown to be insufficient protection, as the private RSA key of a GnuPG user is extracted in their proof-of-concept experiment. Their attack mechanism is based on L3 probing (which is shared among all CPU cores on the chip), so as to determine the execution timing for a specific memory location, which is combined with a kernel page allocator prediction mechanism, injecting page faults and provoking predictable effects.

Work of [LSG+16] proposes branch shadowing attacks on SGX. These attacks allow for an

adversary to gather the fine-grained control of an enclave program, namely the result of branch instructions and the target address of said branch, which can be achieved through timing attacks, execution traces, or by having a malicious OS interrupting enclave processes. This work also proposes hardware and software-based countermeasures, including the usage of techniques such as ORAM [RLT15] and branch removal [OCN+16].

Chapter 5

Formalising Isolated Execution Environments

In this chapter, we generalise the previously presented hardware technologies into an encompassing notion of Isolated Execution Environments, and demonstrate how an attestation mechanism can be implemented over the described hardware assumptions. More specifically, two approaches for attestation are proposed, applicable in different settings: Attested Computation, which captures the fundamental notions required for ensuring security properties of isolated execution environments, and Labelled Attested Computation, refining the previous approach to enable more efficient and complex applications.

5.1 Isolated Execution Environments

At the high-level, an Isolated Execution Environment can be seen as an idealised random access machine running some fixed program P, whose behaviour can only be influenced via a well-specified interface that permits passing inputs to the program, and receiving its outputs. Intuitively, an IEE gives the following security guarantees. The I/O behaviour of a process running in an IEE is determined by the program it is running, the semantics of the language in which the program is written, and the inputs it receives. This means, in particular, that there is strict isolation between processes running in different IEEs (and any other program running on the machine). Furthermore, the only information that is revealed about a program running within an IEE is contained in its input/output behaviour (which in most hardware systems is simply shared memory between the protected code and the untrusted software outside). As such, this section presents two flavours of computation to be run within IEEs: *non-labelled* and *labelled*. Non-labelled computation captures a more intuitive scenario, where an IEE provides attestation guarantees for the whole I/O trace up to some part of the code (potentially all of it), and from then onwards no guarantees

are enforced. Labelled computation allows for considering more complex settings, where attestation is ensured over only a subset of the whole execution classified by input/output labels, allowing for more general applications and capturing the non-attested approach as a specific case.

Programs and composition

Implicit throughout the work will be a programming language $\mathcal{L}$ in which programs are written. It is assumed that this language is used by all computational platforms, but one can admit IEE-specific system calls giving access to different cryptographic functionalities. These are referred as the *security module* interface. An additional system call rand is also assumed to be present in all platforms, giving access to fresh random coins sampled uniformly at random. Language $\mathcal{L}$ is assumed to be deterministic modulo the operation of system calls. As mentioned above, it is important for our results that system calls cannot be used by a program to store additional implicit state that would escape IEE control. To this end, it is imposed that the results of system calls within an IEE can depend only on: i. an initially shared state that is defined when a program is loaded (e.g., the cryptographic parameters of the machine, and the code of the program); ii. the input explicitly passed on that particular call; and iii. fresh random coins. As a consequence of this, it may be assumed that system calls placed by different parts of a program are identically distributed, assuming that the same input is provided. This is particularly important when program composition is considered.

A program P must be written as a transition function, mapping bit-strings to bit-strings. Such functions take a current state st and an input i, and they will produce a new output o and an updated state. This will be referred to as an *activation* and expressed as $o \leftarrow P[\mathsf{st}](i)$. Unless otherwise stated, st will be assumed to be initially empty. It is imposed that every output produced by a program includes a Boolean flag $\mathsf{finished}$ that indicates whether the transition function will accept further input. The transition function may return arbitrary output until it produces an output where $\mathsf{finished} = \mathsf{T}$, at which point it can return no further output or change its state. This notation is extended as $o \leftarrow P[\mathsf{st}; r](i)$ to account for the randomness obtained via the rand system call as extra input r; and as $(o_1, \ldots, o_n) \leftarrow P[\mathsf{st}; r](i_1, \ldots, i_n)$ to represent a sequence of activations. It is written $\mathsf{Trace}_{P[\mathsf{st};r]}(i_1, \ldots, i_n)$ for the corresponding I/O trace $(i_1, o_1, \ldots, i_n, o_n)$.

Given two programs P and Q, and a projection function between the internal states of the two programs ϕ, the sequential composition of the two programs will be referred to as $\mathsf{Compose}_{\phi}\langle P, Q\rangle$. This is defined as a transition function R that has two execution stages, which are signalled in its output via an additional stage bit. In the first stage, every input to R will activate program P. This will proceed until P's last output indicates it has finished (inclusively). The next activation will trigger the start of the second stage, at which point R initialises the state of Q using $\phi(\mathsf{st}_P)$ before activating it for the first time. Additionally it is

required that a constant indicating the current stage (termination being counted as a third stage) is appended to any output of a composition. When dealing with such a composed program, it is denoted by $\mathsf{ATrace}_{R[\mathsf{st};r]}(i_1, \ldots, i_n)$ the prefix of the trace that corresponds to the execution of P. Intuitively, this denotes the *attested trace* where only the initial part of the program must be protected via attestation.

Labelled programs and composition

Labelled computation provides a more general approach, capturing scenarios where multiple participants run code within IEEs, in which each participant is only interested in validating a part (or subtrace) of the execution: its own. The same model for programs is used, but is now extended to the settings where inputs/outputs are labelled: programs as transition functions which take a current state st and a label-input pair (l, i), and produce a new output o and an updated state. The operations are written as $o \leftarrow P[\mathsf{st}](l, i)$ for each such action and are referred to as *activations*. Throughout the thesis the focus is restricted to programs (even if they are adversarially created) for which the transition function is guaranteed to run in polynomial-time.[1] Unless otherwise stated, st is assumed to be initially empty.

The notation is extended to account for probabilistic programs that invoke the rand system call. Let $o \leftarrow P[\mathsf{st}; r](l, i)$ denote the activation of P which, when invoked on labelled input (l, i) (with internal state st and random coins r), produced output o. Sequence of activations are written as $(o_1, \ldots, o_n) \leftarrow P[\mathsf{st}; r](l_1, i_1, \ldots, l_n, i_n)$ and denoted by $\mathsf{Trace}_{P[\mathsf{st};r]}(l_1, i_1, \ldots, l_n, i_n)$ as the corresponding input/output trace $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$. When dealing with a trace T, $\mathsf{filter}[L](T)$ will be used to denote the projection of the trace that retains only I/O pairs that correspond to labels in L. For simplicity of presentation, notation will be abused to employ $\mathsf{filter}[l]$ when L is a singleton.

The basic notion of program composition is also extended to consider the two general label-based forms of program composition shown in Fig. 5.1 that can be applied recursively and interchangeably to create arbitrarily complex programs in a modular way.

Parallel composition of programs $P_1, \ldots P_n$, denoted $\langle P_1 \mid \ldots \mid P_n \rangle_{p_1,\ldots,p_n}$, can be seen as the transition function that takes inputs with extended labels of the form (p_i, l)[2]—here p_i are bitstrings used to identify the target program, where it is assumed $p_i \neq p_j$ for i, j distinct—and dispatches incoming label-input pairs to the appropriate program. Parallel composition excludes the possibility of state sharing between programs, and defines termination to occur when all composed programs have terminated. Sequential composition of two programs P and Q via projection function ϕ, denoted $\langle P; Q \rangle_{\phi,p,q}$, can be seen as the transition function

[1] In particular it is assumed that adversarially generated programs cannot *blow up* the execution time of an experiment beyond poly-time in the security parameter.

[2] It is assumed some form of non-ambiguous encoding of composed labels and output strings, but this presentation simply presents these encoded values as tuples.

Program $\langle P_1 \mid \ldots \mid P_n \rangle_{p_1,\ldots,p_n}[\mathsf{st}](l,i)$:
- If $\mathsf{st} = \epsilon$: For $i \in [1..n]$: $\mathsf{st.finished}.p_i \leftarrow \mathsf{F}$; $\mathsf{st}.p_i \leftarrow \epsilon$
- If $(\wedge_{i=1}^{n}\, \mathsf{st.finished}.p_i)$: Return ϵ
- If $\exists k \in [1..n]$ s.t. $l = (p_k, l')$:
  - If $\neg \mathsf{st.finished}.p_k$:
    - $o \leftarrow\$\ P_k[\mathsf{st}.p_k](l', i)$
    - $\mathsf{st.finished}.p_k \leftarrow o.\mathsf{finished}$
  - Else: $o \leftarrow \epsilon$
- Else: $o \leftarrow \perp$
- Return $\left(\wedge_{i=1}^{n}\, \mathsf{st.finished}.p_i, o\right)$

Program $\langle P\,;\, Q \rangle_{\phi,p,q}[\mathsf{st}](l,i)$:
- If $\mathsf{st} = \epsilon$: $\mathsf{st.stage} \leftarrow 0$; $\mathsf{st.finished} \leftarrow \mathsf{F}$ $\mathsf{st.st}' \leftarrow \epsilon$
- If $\mathsf{st.finished}$: Return ϵ
- If $\mathsf{st.stage} = 0 \wedge l = (p, l')$:
  - $o \leftarrow\$\ P[\mathsf{st.st}'](l', i)$
  - If $o.\mathsf{finished}$: $\mathsf{st.stage} \leftarrow 1$; $\mathsf{st.st}' \leftarrow \phi(\mathsf{st.st}')$
- Else:
  - If $\mathsf{st.stage} = 1 \wedge l = (q, l')$:
    - $o \leftarrow\$\ Q[\mathsf{st.st}'](l', i)$
    - $\mathsf{st.finished} \leftarrow o.\mathsf{finished}$
  - Else: $o \leftarrow \perp$
- Return $(\mathsf{st.stage}, \mathsf{st.finished}, o)$

Figure 5.1: Parallel (left) and sequential (right) program composition.

that has two execution stages, which are signalled in its output via an additional stage flag. As above, $o.\mathsf{stage}$ denotes the value of this flag in some output o. For consistency, labels are assumed to be of the form (p, l) and (q, l) where $p \neq q$ are used to identify the target program. In the first stage, every label-input pair will be checked for consistency (i.e., that it indicates P as the target program) and dispatched to program P. This will proceed until P's last output indicates it has finished (inclusively). The next activation will trigger the start of the second stage, at which point the composed program initialises the state of Q using $\phi(\mathsf{st}_P)$ before activating it for the first time. As previously, it is required that a constant indicating the current stage is appended to any output of a composition. No other state sharing between P and Q is admitted, in addition to that fixed by ϕ.

Machines

A *machine* $\mathcal{M}$ is an abstract computational device that captures the resources offered by a real world computer or group of computers, whose hardware security functionalities are initialised by a specific manufacturer before being deployed, possibly in different end-users. For example, a machine may represent a single computer produced by a manufacturer, configured with a secret signing key for a public key signature scheme, and whose public key is authenticated via some public key infrastructure, possibly managed by the manufacturer itself. Similarly, a machine may represent a group of computers, each configured with secret signing keys associated with a group signature scheme; again, the public parameters for the group would then be authenticated by some appropriate infrastructure.[3]

Machines will be modeled via a simple external interface, which can be seen as both the functionality that higher-level cryptographic schemes can rely on when using the machine, and the adversarial interface that will be the basis of the presented attack models. Loosely speaking, this interface can be thought of as the ideal functionality that captures a system

[3]If the possibility of removing elements from the group is not needed, then even sharing the same signing key for a public key encryption scheme between multiple computers could be a possibility.

such as SGX, or the Secure OS of TrustZone. The non-labelled interface is as follows:

- $\mathsf{Init}(1^\lambda)$ is the global initialisation procedure which, on input the security parameter, outputs the global parameters prms. This algorithm represents the machine's hardware initialisation procedure, which is out of the user's and the adversary's control. Intuitively, it initialises the internal security module, the internal state of the remote machine and returns any public cryptographic parameters that the security module releases. It should be emphasized that the global parameters of machines are the only pieces of information that are assumed to be authenticated using external mechanisms (such as a PKI) in the entire work.

- $\mathsf{Load}(P)$ is the IEE initialisation procedure. On input a program/transition function P, the machine produces a fresh handle hdl, creates a new IEE with handle hdl, loads P into the new IEE and returns hdl. The machine interface does not provide direct access to either the internal state of an IEE nor to its randomness input. This means that the only information that is leaked about internal state and randomness input is that revealed (indirectly) via the outputs of the program.

- $\mathsf{Run}(\mathsf{hdl}, i)$ is the process activation procedure. On input a handle hdl and an input i, it will activate process running in isolated execution environment of handle hdl with i as the next input. When the program/transition function produces the next output o, this is returned to the caller.

The described interface can be instantiated for Intel SGX or ARM TrustZone. The Init procedure can be seen as the device key initialization performed by the manufacturer on all SGX hardware, or as the generate procedure for TrustZone proposed in [ZZH+14]. The Load/Run procedures are handled by either enclave initialization and execution on SGX, or by the Secure OS application management on TrustZone.

The I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in some machine $\mathcal{M}$ is defined as the tuple $(i_1, o_1, \ldots, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of the Run procedure on hdl; $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl})$ is the code (program) running inside the process with handle hdl; $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl})$ represents the coins given to the program by the rand system call; and $\mathsf{State}_{\mathcal{M}}(\mathsf{hdl})$ is the internal state of the program. Finally, $\mathcal{A}^{\mathcal{M}}$ denotes the interaction of some algorithm with a machine $\mathcal{M}$, i.e., having access to the Load and Run oracles defined above.

Alternatively, the interface for machines can be tweaked to consider the usage of labels for program executions. More specifically, the process activation procedure is denoted as $\mathsf{Run}(\mathsf{hdl}, l, i)$. On input a handle hdl and a label-input pair (l, i), it will activate process running in isolated execution environment of handle hdl with (l, i) as the next input. When the program/transition function produces the next output o, this is returned to the caller.

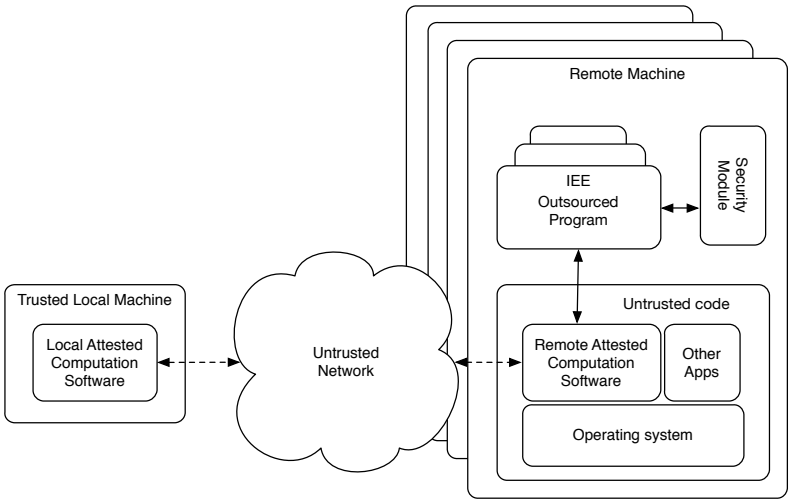


Figure 5.2: Attested Computation scenario.

Following the previous notation, the I/O trace $\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$ of a process hdl running in some machine $\mathcal{M}$ is defined as the tuple $(l_1, i_1, o_1, \ldots, l_n, i_n, o_n)$ that includes the entire sequence of n inputs/outputs resulting from all invocations of Run on hdl.

5.2 Attested Computation

One begins by formalising a cryptographic primitive that builds upon the non-labelled abstraction presented in Section 5.1, aiming to address the remote execution, i.e., outsourcing, of programs as illustrated in Figure 5.2. In this setting, a single user running software in a trusted local machine wishes to use an untrusted network to access a pool of remote machines with IEE facilities. The remote machines will be running general-purpose operating systems and other untrusted software. The goal of the user is to run a specific program P within an IEE in one of the remote machines, and to obtain assurance that, not only the program is indeed executing there, but also that it is displaying a particular I/O behaviour. This is called *attested computation*, and it is now introduced as the cryptographic primitive that formalises the simplest cryptographic application of trusted hardware systems offering IEE functionalities.

Syntax

An *Attested Computation* (AC) scheme is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ is the program compilation algorithm. On input global parameters for some machine $\mathcal{M}$, and programs P and Q, whose composition under projection

function ϕ will be outsourced, it will output program R^*, together with an initial (possibly empty) state st for the verification algorithm. This algorithm is run locally. R^* is the code to be run as an isolated process in the remote machine. Intuitively, P is the initial part of the remote code that requires attestation guarantees, whereas Q is any subsequent code that may be remotely executed (generally leveraging the security guarantees that have been bootstrapped using the initial attested execution).

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is the attestation algorithm. On input global parameters for $\mathcal{M}$, a process handle hdl and an input i, it will use the interface of $\mathcal{M}$ to obtain attested output o^*. This algorithm is run remotely, but in an unprotected environment: it is responsible for interacting with the isolated process running R^*, providing it with inputs and recovering the (possibly attested) outputs that should be returned to the local machine.

- $\mathsf{Verify}(\mathsf{prms}, i, o^*, \mathsf{st})$ is the (stateful) output verification algorithm. On input global parameters for $\mathcal{M}$, an input i, a (possibly attested) output o^* and some state st, it will produce an output value o and an updated state, or the failure symbol $\perp$. This failure symbol is encoded so as to be distinguishable from a valid output of a program, resulting from a successful verification. This algorithm is run locally on claimed outputs from the Attest algorithm.

In Figure 5.2, the local attested computation software block corresponds to $\mathsf{Compile}$ (one initial usage per program) and Verify (one usage per incoming attested output), whereas the remote attested computation software block corresponds to Attest (one usage per remote program activation, i.e. per I/O transition). The above syntax can be naturally extended to accommodate the simultaneous compilation of multiple input programs and/or the possibility that $\mathsf{Compile}$ may generate multiple output programs. This would allow this formalisation to capture, e.g., map/reduce applications such as those described in [SCF+15a].

Correctness

Intuitively, an AC scheme is correct if, for any given programs P and Q and assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a view of the I/O sequence that took place in the remote environment. Furthermore, this I/O sequence must be consistent with the semantics of $\mathsf{Compose}_\phi\langle P; Q\rangle$. In other words, suppose the compiled program is run under handle hdl^* in remote machine $\mathcal{M}$, and the local user uses Verify to reconstruct the remote I/O behaviour $(i_1, o_1, \ldots, i_n, o_n)$. Then, $R := \mathsf{Compose}_\phi\langle P; Q\rangle$, it must be that

$$\mathsf{Trace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n) = (i_1, o_n, \ldots, i_n, o_n)$$

The following definition formalizes the notion of a local user *correctly remotely executing program P* using attested computation.

Definition 16 (Correctness)**.** *An Attested Computation scheme* AC *is correct if, for all* λ*, and all adversaries* $\mathcal{A}$*, the experiment in Figure 5.3 (left) always returns* T*.*

The adversary in this correctness experiment definition is choosing inputs, hoping to find a sequence that causes the attestation protocol to behave inconsistently with respect to the semantics of P (when these are made deterministic by hard-wiring the same random coins used remotely). This approach is used for defining correctness because it makes explicit what is an honest execution of an attested computation scheme, when compared to the security experiment introduced next.

Since dealing with composed programs is core to the success of this formalisation, the correctness requirements on attested computation schemes are extended to preserve the structure of the input program (P, ϕ, Q), and to modify only the part of the code that will be attested. Formally, it is imposed that, given any program P, there exists a (unique) compiled program P^*, such that, for any mapping function ϕ and any program Q, it is defined that $\mathsf{Compose}_{\phi}\langle P^*; Q\rangle = \mathsf{Compile}(P, \phi, Q)$.

Security

Security of an attested computation scheme imposes that an adversary with absolute control of the remote machine cannot convince the local user that some arbitrary remote execution of a program P has occurred, when it has not (nothing is said about the subsequent remote execution of program Q). Formally, the adversary is allowed to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs. The adversary wins if the local user reconstructs an execution trace without aborting (i.e., all attested outputs must be accepted by the verification algorithm) and one of two conditions occur: i. the execution trace that is validated by Verify is inconsistent with the semantics of P (in which case an adversary would be able to convince the local user of an I/O sequence that could not possibly have occurred!); or ii. there does not exist a remote process hdl^* exhibiting a consistent execution trace (in which case, the adversary would be able to convince the local user that a process running P was executing in the remote machine, when it was not).

Since the adversary is free to interact with the remote machine as it pleases, it will always be capable of appending arbitrary inputs to the trace of any remote process, while refusing to deliver all of the resulting attested outputs to the local user. This justifies the winning condition in the presented security game referring to a prefix of the trace in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that the locally recovered trace and the remote trace share a common prefix ($\sqsubseteq$), which

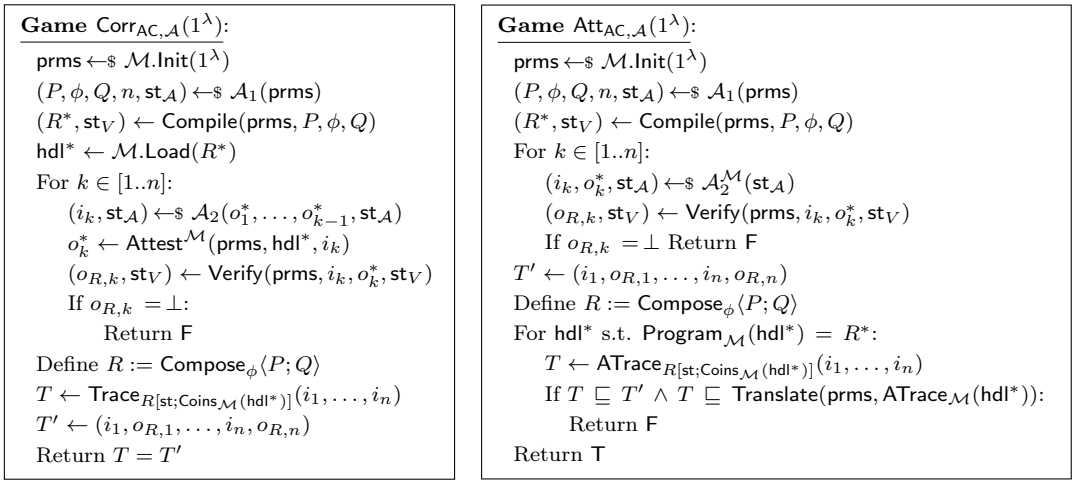


Figure 5.3: Games defining the correctness (left) and security (right) of an AC scheme.

exactly corresponds to the part of the source program's behaviour that should be protected by attestation.

Formally, it should be accounted for that the actual I/O sequence of the remote program includes more information than that of R, e.g., to allow for the cryptographic enforcement of security guarantees. The presented definition is parametrised by a $\mathsf{Translate}$ algorithm that permits formalising this notion of *semantic consistency*. Another way to see $\mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl}^*))$ is as a trace translation procedure associated with a given AC scheme, which maps remote traces into traces at the source level.

Definition 17 (Security)**.** *An attested computation scheme is secure if there exists an efficient deterministic algorithm* $\mathsf{Translate}$ *s.t., for all ppt adversaries* $\mathcal{A}$*, the probability that experiment in Figure 5.3 (bottom) returns* T *is negligible.*

Observe that the adversary loses the game as long as there exists at least one remote process that matches the locally reconstructed trace. This should be interpreted as the guarantee that IEE resources are indeed being allocated in a specific remote machine to run at least one instance of the remote program (note that if the program is deterministic, many instances could exist with exactly the same I/O behaviour, which is *not* seen as a legitimate attack). Furthermore, this definition imposes that the compiled program uses essentially the same randomness as the source program (except of course for randomness that the security module internally uses to provide its cryptographic functionality), as otherwise it will be easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of the presented modelling approach, but in no way does it limit the applicability of the primitive proposed: it just makes it explicit that the transformation that is performed on the code for attestation will typically consist of an instrumentation of the code by applying cryptographic processing to the inputs and outputs it receives.

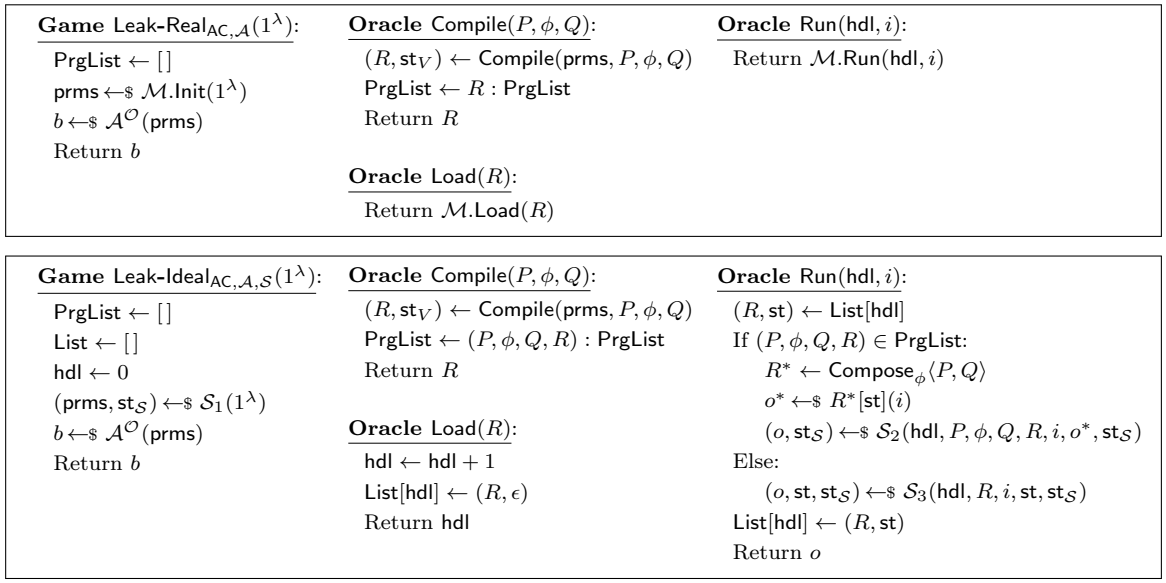


Figure 5.4: Games defining minimum leakage of an AC scheme.

Minimum leakage

From the discussion above, an AC scheme should guarantee that the I/O behaviour of the program in the remote machine includes at least the information required to reconstruct an hypothetical local execution of the source program. However, it is important to establish an additional restriction on what AC compilation actually does to a source program, to ensure that this primitive can be used to achieve ambitious goals, namely to perform attestation of the remote execution of cryptographic code.

The following definition imposes that nothing from the internal state of the source programs (in addition to what is public, i.e. the code and I/O sequence) is leaked in the trace of the compiled program when it is remotely executed.

Definition 18 (Minimal leakage)**.** *Attested Computation scheme* AC *ensures security with minimal leakage if it is secure according to Definition 20 and there exists a ppt simulator* $\mathcal{S}$ *that, for every adversary* $\mathcal{A}$*, the following distributions are identical:*

$$\{\, \mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}(1^\lambda) \,\} \approx \{\, \mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathcal{S}}(1^\lambda) \,\}$$

where games $\mathsf{Leak\text{-}Real}_{\mathsf{AC},\mathcal{A}}$ *and* $\mathsf{Leak\text{-}Ideal}_{\mathsf{AC},\mathcal{A},\mathcal{S}}$ *are shown in Figure 5.4.*

Notice that the simulator is allowed to replace the global parameters of the machine with some value prms for which it can keep some trapdoor information. Intuitively, this means that one can construct a perfect simulation of the remote trace by simply appending cryptographic material to the local trace. This property is important when claiming that the security of a cryptographic primitive is preserved when it is run within an attested computation scheme

(one can simply reduce the advantage of an adversary attacking the attested trace, to the security of the original scheme using the minimum leakage simulator).

Attested Computation in practice

The considered remote attestation protocol is inspired in the Secure Guard Extensions (SGX) architecture proposed by Intel as described in Section 4.1. A similar implementation of attested computation with a signing enclave can also be achieved in TrustZone, following the approach detailed in 4.2. The main feature of this system is that the remote machine is equipped with a security module that manages both short-term and long-term cryptographic keys, with which it is capable of producing MACs that enable authenticated communication between various IEEs and digital signatures that can be publicly verified by anyone holding the (long-term) public key for that machine (or group of machines). A simplified version of how this security module operates is now formalised.

The security module relies on a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ and a MAC scheme $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Ver})$, and it operates as follows:

- When the host machine is initialised, the security module generates a key pair $(\mathsf{pk}, \mathsf{sk})$ using $\Sigma.\mathsf{Gen}$ and a symmetric key key using $\Pi.\mathsf{Gen}$. It also creates a special process running code S^* (see below for a description of S^*) in an IEE with handle 0. The security module then securely stores the key material for future usage, and outputs the public key. In this case it is assumed that the output of $\mathcal{M}.\mathsf{Init}$ is $\mathsf{prms} = \mathsf{pk}$.

- The operation of IEE with handle 0 will be different from all other IEEs in the machine. Program S^* will permanently reside in this IEE, and it will be the only one with direct access to both sk and key.

- The code of S^* is dedicated to transforming messages authenticated with key into messages signed with sk. On each activation, it expects an input $(\mathsf{msg}, \mathsf{t})$. It obtains key from the security module and verifies the tag using $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, \mathsf{msg})$. If the previous operation was successful, it obtains sk from the security module, signs the message using $\sigma \leftarrow\!\!\$\ \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{msg})$ and writes σ to the output. Otherwise, it writes $\perp$ in the output.

- The security module exposes a single system call $\mathsf{mac}(\mathsf{msg})$ to code running in all other IEEs. Upon such a request from a process running program P, the security module returns a MAC tag t computed using key over both the code of P and the input message (msg).

Observe that the operation of the security module allows any process to produce an authenticated message that can be validated by the special process running S^* as coming from within

another IEE in the same machine.

It is assumed that the message authentication code scheme Π and the signature scheme Σ satisfy the standard notions of correctness and existential unforgeability detailed in Section 2.1, and that the machine's public key is authenticated by some external PKI.

It is now presented an AC scheme that relies on a remote machine supporting a security module with the above functionality. The operation of the various algorithms is intuitive, except for the fact that basic replay protection using a sequence number does not suffice to bind a remote process to a full trace, since the adversary could then run multiple copies of the same process and *mix and match* outputs from various traces. Instead, the remote process must commit to its entire trace whenever an attested output is produced. Details follow:

- $\mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$ will generate a new program $R^* = \mathsf{Compose}_\phi\langle P^*, Q\rangle$ and output it alongside the initial state of the verification algorithm $(R^*, [\,], 1)$, where 1 is an indicator of the stage in which remote program R^* is supposed to be executing. Program P^* is instrumented as follows: it keeps a list ios of all the I/O pairs it has previously received and computed, i.e, its own trace; on each activation with input i, P^* first computes $o \leftarrow\!\!\$\ P[\mathsf{st}_P](i)$ and updates the list by adding a new (i, o) pair; it then requests from the security module a MAC of the updated ios. Due to the operation of the security module, this will correspond to a tag t on the tuple (R^*, ios); it finally outputs $(o, \mathsf{t}, R^*, \mathsf{ios})$. Not that (R^*, ios) is included explicitly in the outputs of R^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ invokes $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i)$ using the handle and input value it has received. When the process produces an output o, Attest parses it into $(o', \mathsf{t}, R^*, \mathsf{ios})$. It may happen that parsing fails, e.g., if Q is already executing, in which case Attest simply produces o as its own output. Otherwise, it uses $\mathcal{M}.\mathsf{Run}(0, (R^*, \mathsf{ios}, \mathsf{t}))$ to convert the tag into a signature σ on the same message. If this conversion fails, then Attest produces the original output o as its own output. Otherwise, it outputs (o', σ).

- $\mathsf{Verify}(\mathsf{prms}, i, o^*, (R^*, \mathsf{ios}, \mathsf{stage}))$ returns o^* if $\mathsf{stage} = 2$. Otherwise, it first parses o^* into (o, σ), appends (i, o) to ios, and verifies the digital signature σ using prms and $(R^*, \mathsf{ios}))$. If parsing or verification fails, Verify outputs $\perp$. If not, then Verify will check if output o indicates that program P^* has finished. If so, it will update stage to value 2. In any case, it terminates outputting o.

It is easy to see that the proposed AC scheme is correct, provided that the underlying signature and message authentication code schemes are themselves correct. To see this, first note that, during the execution of P^*, unless a MAC or signature verification fails, the I/O sequence produce by Verify exactly matches that of $\mathsf{Compose}_\phi\langle P; Q\rangle$, and therefore $T = T'$ is

always T. This follows from the construction of R^*, the operation of Attest, and the fact that the associated randomness tapes are established by $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$ as identical. Furthermore, if the message authentication code scheme is correct, then the MAC verification will never fail, and if the message signature scheme is correct, then the signature verification will never fail. This is because the combined actions of R^*, Attest, the signing process running S^* and the security module lead to tags and signatures on tuples (R^*, ios) that exactly match those input to the verification algorithms $\Pi.\mathsf{Ver}$ and $\Sigma.\mathsf{Verify}$. Finally, after executing P^*, given that the associated randomness tapes are established by $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$ are identical and that traces are identical up to that point, so will be $\phi(\mathsf{st}_P)$ in both sides, and all subsequent calls to Q will display a similar behaviour.

Let $\mathsf{Translate}$ be the deterministic function that receives the machine parameters and a list of tuples of the form $(i, (o, \mathsf{t}, R^*, \mathsf{ios}))$ and returns a list of pairs of the form (i, o).

Theorem 1. *The AC scheme presented above provides secure attestation if the underlying MAC scheme Π and signature scheme Σ are existentially unforgeable. Furthermore, it unconditionally ensures minimum leakage.*

The proof is a sequence of three games presented in Figure 5.5 and Figure 5.6. The first game is simply the AC security game instantiated with the proposed protocol. In game $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a sforge event occurs. Intuitively, this event corresponds to the adversary producing a signature that was not computed by the signing process with handle 0, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{Att}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{sforge}\,]\,.$$

To upper bound the distance between these two games, an adversary $\mathcal{B}$ against the existential unforgeability of signature scheme Σ in S^* is constructed, such that

$$\Pr[\,\mathsf{sforge}\,] \leq \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{B}}(\lambda)$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the signing operations performed within the process loaded by the security module are replaced with calls to the Sign oracle provided in the existential unforgeability game. More precisely, whenever process 0 in the remote machine is expected to compute a signature on message msg, algorithm $\mathcal{B}$ calls its own oracle on (R^*, msg) to obtain σ.

When sforge is set, according to the rules of game $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}$, algorithm $\mathcal{B}$ outputs message (R^*, ios) and candidate signature σ. It remains to show that this is a valid forgery. To see this, first observe that this is indeed a valid signature, as signature verification is performed

Game $\mathbf{G0}_{\mathsf{AC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \phi, Q, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$

$(R^*, (R^*, \mathsf{ios}, \mathsf{stage}')) \leftarrow \mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$
For $k \in [1..n]$:
 $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_\mathcal{A})$
 If $\mathsf{stage}' = 1$:
  Parse $(o'_k, \sigma) \leftarrow o_k^*$; $(o_k, \mathsf{finished}, \mathsf{stage}) \leftarrow o'_k$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (R^*, (i_k, o'_k) : \mathsf{ios}))$:
   $(o_{R,k}, \mathsf{ios}) \leftarrow (o'_k, (i_k, o'_k) : \mathsf{ios})$
   If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage}' \leftarrow 2$
  Else: Return F
 Else $o_{R,k} \leftarrow o_k$
$T' \leftarrow (i_1, o_{R,1}, \ldots, i_n, o_{R,n})$
Define $R := \mathsf{Compose}_\phi\langle P; Q\rangle$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = R^*$:
 $T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$
 If $T \sqsubseteq T' \wedge T \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl}^*))$:
  Return F
Return T

Game $\mathbf{G1}_{\mathsf{AC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \phi, Q, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$
$(R^*, (R^*, \mathsf{ios}, \mathsf{stage}')) \leftarrow \mathsf{Compile}(\mathsf{prms}, P, \phi, Q)$
For $k \in [1..n]$:
 $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^\mathcal{M}(\mathsf{st}_\mathcal{A})$
 If $\mathsf{stage}' = 1$:
  Parse $(o'_k, \sigma) \leftarrow o_k^*$; $(o_k, \mathsf{finished}, \mathsf{stage}) \leftarrow o'_k$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (R^*, (i_k, o'_k) : \mathsf{ios}))$:
   $(o_{R,k}, \mathsf{ios}) \leftarrow (o'_k, (i_k, o'_k) : \mathsf{ios})$
   If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage}' \leftarrow 2$
  Else: Return F
  If $(((R^*, (i_1, o_{R,1}, \ldots, i_k, o_k)), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
   $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
 Else $o_{R,k} \leftarrow o_k$
$T' \leftarrow (i_1, o_{R,1}, \ldots, i_n, o_{R,n})$
Define $R := \mathsf{Compose}_\phi\langle P; Q\rangle$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = R^*$:
 $T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$
 If $T \sqsubseteq T' \wedge T \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl}^*))$:
  Return F
Return T

Figure 5.5: First game hop for the proof of security of the AC protocol.

on these values immediately before sforge occurs. It suffices to establish that message $(R^*, (i_1, o'_1, \ldots, i_k, o'_k))$ could not have been queried from the Sign oracle. Access to the signing key that allows signatures to be performed is only permitted to the special process with handle 0. From the construction of S^*, producing such a signature would only occur via the inclusion of $(R^*, (i_1, o'_1, \ldots, i_k, o'_k))$ in its trace. Since this is known not to be the case, (R^*, ios) could not have been queried from the signature oracle. It is therefore concluded that $\mathcal{B}$ outputs a valid forgery whenever sforge occurs.

In game $\mathsf{G_2}^{\mathsf{AC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a mforge event occurs. Intuitively, this event corresponds to the adversary producing a tag that was not computed by the security module, and hence constitutes a forgery with respect to Π. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G_2}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{mforge}\,].$$

To upper bound the distance between these two games, an adversary $\mathcal{C}$ against the existential unforgeability of MAC scheme Π in the security module is constructed, such that

$$\Pr[\,\mathsf{mforge}\,] \leq \mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{C}}(\lambda)$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G_2}^{\mathsf{AC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the MAC operations computed inside the

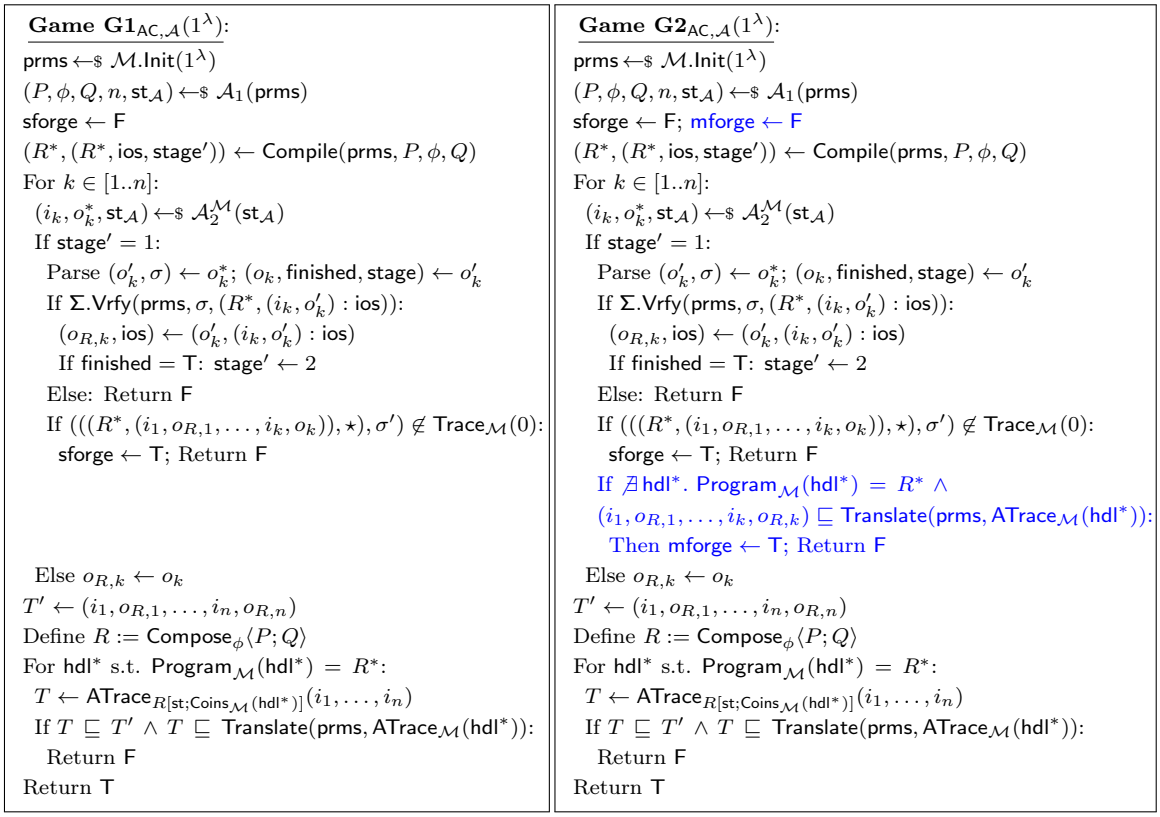


Figure 5.6: Second game hop for the proof of security of the AC protocol.

internal security module are replaced with calls to the Auth oracle provided in the existential unforgeability game. More precisely, whenever a process running code R^* within an IEE in the remote machine requests a MAC on message msg from the security module, algorithm $\mathcal{C}$ calls its own oracle on (R^*, msg) to obtain t.

Let $T \leftarrow (i_1, o_{R,1}, \ldots, i_k, o_{R,k})$. When mforge is set according to the rules of game $\mathsf{G}_2^{\mathsf{AC},\mathcal{A}}$, algorithm $\mathcal{C}$ retrieves the trace of the process with handle 0 running S^*, locates the input/output pair $(((R^*, T), \mathsf{t}), \sigma')$ and outputs message (R^*, T) and candidate tag t. To see this is a valid forgery, first observe that, having failed the sforge check, $(((R^*, T), \mathsf{t}), \sigma')$ is in the trace of the process with handle 0, so by its construction it is known that the corresponding input $((R^*, T), \mathsf{t})$ must contain a valid tag. It suffices to establish that message (R^*, T) could not have been queried from the Auth oracle. Suppose that the first part of the mforge check failed, i.e., that $\nexists\, \mathsf{hdl}^*.\ \mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = R^*$. Then, because the security module signs the code of the processes requesting the signatures, there is an assurance that such a query was never placed to the Auth oracle. Furthermore, any MAC query for a message starting with R^* must have been caused by the execution of an instance of R^*. Now suppose some instances of R^* were indeed running in the remote machine, but that none of them displayed the property $(i_1, o_{R,1}, \ldots, i_k, o_{R,k}) \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl}^*))$. Then, by the construction of R^*, it can be excluded that (R^*, T) was queried from the MAC oracle. As such, it can be concluded

that $\mathcal{C}$ outputs a valid forgery whenever mforge occurs.

To complete the proof, it is now argued that the adversary never wins in game $\mathsf{G_2}^{\mathsf{AC},\mathcal{A}}$. To see this, observe that the game reaching the final check induces the guarantee that

$$\exists\, \mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}^*) = R^* \wedge$$
$$(i_1, o_{R,1}, i_2, o_{R,2}, \ldots, i_k, o_{R,k}) \sqsubseteq \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl}^*)))$$

By the construction of R^*, it immediately follows that the final check in the game will always cause the adversary to lose:

- T is fixed by the input sequence, the value of the randomness tape and the semantics of R, which determines the sequence of outputs $(o_{1,L}, \ldots, o_{n,L})$.
- The above existential guarantee for hdl^* implies that an instance of R^* in the remote machine received the same initial k inpu sequence as that fixed by T.
- Considering $T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$, it is also inferable that the randomness tape used to produce values for T is identical to the one used in hdl^*.
- One can therefore inductively deduce, by the semantics of R^*, that the same process has produced an initial sequence of k outputs that (modulo the action of $\mathsf{Translate}$) is identical to that included in T.
- Subsequent inputs after $\mathsf{stage}' \leftarrow 2$ may produce $n-k$ additional non-attested outputs that are appended to T', and that differ from T. However, the above observation implies that $T \sqsubseteq T'$, and the proposed security claim follows.

To finish the proof, it must now be shown that this scheme also provides security with minimum leakage. This implies defining a ppt simulator $\mathcal{S}$ that provides identical distributions with respect to experiment in Figure 5.4. This is easy to ascertain given the simulator behaviour described in Figure 5.7: $\mathcal{S}_1$ and $\mathcal{S}_3$ follow the exact description of the actual machine, modulo the generation of $(\mathsf{pk}, \mathsf{sk})$ and key. $\mathcal{S}_2$ takes an external output produced by $R[\mathsf{st}](i)$ and returns an output in accordance to the behaviour of $\mathcal{M}$, which given language $\mathcal{L}$ may differ from a real output only by the random coins. As such, the distribution provided by the simulator is indistinguishable to the one provided by a real machine, and the original claim follows.

□

5.3 Labelled Attested Computation

The previous definition considers a one-to-one setting, where a local user wishes to execute its code remotely, and thus has all information necessary to validate the full I/O trace. Labelled

Simulator $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$

Simulator $\mathcal{S}$ will perform according to the $\mathcal{M}$ execution description.

- Upon input 1^λ, $\mathcal{S}_1$ generates a key pair for process S^*, a MAC key for the security module and initializes the traces as an empty list. The public key will be the public parameters, while the secret key be stored in its initial state.

$\underline{\mathcal{S}_1(1^\lambda)}$:
  $\mathsf{key} \leftarrow\$\ \Pi.\mathsf{Gen}(1^\lambda)$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \Sigma.\mathsf{Gen}(1^\lambda)$
  $\mathsf{Traces} \leftarrow [\,]$
  Return $(\mathsf{pk}, (\mathsf{key}, \mathsf{sk}, \mathsf{Traces}))$

- $\mathcal{S}_2$ maintains a list of traces Traces with the respective list ios and stage stage. Given this, it masks output o^* as if produced by an actual machine execution.

$\underline{\mathcal{S}_2(\mathsf{hdl}, P, \phi, Q, R, i, o^*, \mathsf{st}_\mathcal{S})}$:
  Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
  If $\nexists\,(\mathsf{ios}, \mathsf{stage}) \in \mathsf{Traces}[\mathsf{hdl}]$: $\mathsf{ios} \leftarrow [\,]$; $\mathsf{stage} \leftarrow 1$
  $\mathsf{ios} \leftarrow (i, o^*) : \mathsf{ios}$
  If $\mathsf{stage} = 1$: $\mathsf{msg} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, R, \mathsf{ios}))$
  Else $\mathsf{msg} \leftarrow o^*$
  Parse $(o, \mathsf{finished}, \mathsf{stage}) \leftarrow o^*$
  If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage} \leftarrow 2$
  $\mathsf{Traces}[\mathsf{hdl}] \leftarrow (\mathsf{ios}, \mathsf{stage})$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
  Return $(\mathsf{msg}, \mathsf{st}_\mathcal{S})$

- $\mathcal{S}_3$ computes the next output given input i, program R and state st. The result is afterwards treated similar to $\mathcal{S}_2$.

$\underline{\mathcal{S}_3(\mathsf{hdl}, R, i, \mathsf{st}, \mathsf{st}_\mathcal{S})}$:
  Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
  If $\mathsf{hdl} = 0$:
    Parse $(i, \mathsf{t}) \leftarrow i^*$
    If $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, i)$: Return $\Sigma.\mathsf{Sign}(\mathsf{sk}, i)$
    Else Return $\perp$
  If $\nexists\,(\mathsf{ios}, \mathsf{stage}) \in \mathsf{Traces}[\mathsf{hdl}]$: $\mathsf{ios} \leftarrow [\,]$; $\mathsf{stage} \leftarrow 1$
  $o^* \leftarrow\$\ R[\mathsf{st}](i)$
  If $\mathsf{stage} = 1$: $\mathsf{msg} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, R, \mathsf{st.ios}))$
  Else $\mathsf{msg} \leftarrow o^*$
  Parse $(o, \mathsf{finished}, \mathsf{stage}) \leftarrow o^*$
  If $\mathsf{finished} = \mathsf{T}$: $\mathsf{stage} \leftarrow 2$
  $\mathsf{Traces}[\mathsf{hdl}] \leftarrow (\mathsf{ios}, \mathsf{stage})$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
  Return $(\mathsf{msg}, \mathsf{st}_\mathcal{S})$

Figure 5.7: Description of simulator $\mathcal{S}$

Attested Computation is a more general design considering the labelled abstraction presented in Section 5.1. This approach does not rely on a fixed form of composition for attestation, but is instead agnostic of the program's internal structure; furthermore, it permits controlling data flows and attestation guarantees via the label information included in program inputs.

This enables the flexibility of considering I/O traces for many participants, and attestation for each participant's trace individually.

Syntax

A *Labelled Attested Computation* (LAC) scheme is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, L^*)$ is the deterministic program compilation algorithm. On input global parameters for some machine $\mathcal{M}$, program P and an attested label set L^*, it will output program P^*. This algorithm is run locally. P^* is the code to be run as an isolated process in the remote machine, whereas L^* defines which labelled inputs should be subject to attestation guarantees.
- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ is the stateless attestation algorithm. On input global parameters for $\mathcal{M}$, a process handle hdl and label-input pair (l, i), it will use the interface of $\mathcal{M}$ to obtain attested output o^*. This algorithm is run remotely, but in an unprotected environment: it is responsible for interacting with the isolated process running P^*, providing it with inputs and recovering the attested outputs that should be returned to the local machine.
- $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the public (stateful) output verification algorithm. On input global parameters for $\mathcal{M}$, a label l, an input i, an attested output o^* and some state st it will produce an output value o and an updated state, or the failure symbol $\perp$. This failure symbol is encoded so as to be distinguishable from a valid output of a program, resulting from a successful verification. This algorithm is run locally on claimed outputs from the Attest algorithm. The initial value of the verification state is set to be (P, L^*), the same inputs provided to $\mathsf{Compile}$.

Correctness

Intuitively, a LAC scheme is correct if, for any given program P and attested label set L^*, assuming an honest execution of all components in the scheme, both locally and remotely, the local user is able to accurately reconstruct a partial view of the I/O sequence that took place in the remote environment, for an arbitrary set of labels L (which may or may not be related to L^*). For non-attested labels, i.e., labels in $L \setminus L^*$, the I/O behaviour of the compiled program inside the IEE is restricted, by imposing that it is identical to that of the original program. For this reason, the set of labels L should be seen as a parameter that can be used by higher level protocols relying on LAC to specify the *partial* local view that may interest a particular party interacting with a remote machine. Different parties may be interested in different partial views, including both attested an non-attested labels, and the protocol should be correct for all of them. More technically, suppose the compiled program

is run under handle hdl^* in remote machine $\mathcal{M}$, with random coins $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$ and on labelled input sequence $(l_1, i_1, \ldots, l_n, i_n)$. Suppose also that, running the original program on the same random coins and inputs yields

$$\mathsf{Trace}_{R[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(l_1, i_1, \ldots, l_n, i_n) = (l_1, i_1, o_n, \ldots, l_n, i_n, o_n)$$

Then, for any set of labels L, if a local user recovers outputs $(o'_1, \ldots, o'_m)$ corresponding to labelled inputs (l_{k_1}, i_{k_1}) to (l_{k_m}, i_{k_m}), where $l_{k_j} \in L$, it must be the case that $(o'_1, \ldots, o'_m) = (o_{k_1}, \ldots, o_{k_m})$. Outputs for attested labels are passed through Attest and Verify, whereas inputs and outputs for non-attested labels are processed independently of these algorithms. The following definition formalizes the notion of a local user *correctly remotely executing program* P using labelled attested computation.

Definition 19 (Correctness)**.** *A labelled attested computation scheme* LAC *is correct if, for all* λ *and all adversaries* $\mathcal{A}$*, the experiment in Fig. 5.8 (left) always returns* T*.*

The adversary in this correctness experiment definition is choosing inputs, hoping to find a sequence that causes the attestation protocol to behave inconsistently with respect to the semantics of P (when these are made deterministic by hard-wiring the same random coins used remotely). This approach is used to defining correctness because it makes explicit what is an honest execution of an attested computation scheme, when compared to the security experiment introduced next.

Structural preservation

To simplify analysis of constructions and proofs, the correctness requirements on labelled attested computation schemes are extended to preserve the structure of the input program when dealing with sequential composition, and to modify only the part of the code that will be attested.

Formally, it is imposed that, for all global parameters, given any program $R = \langle P\,;\, Q\rangle_{\phi,p,q}$, and an attested label set L^* that contains only labels of the form (p, l), then there exists a (unique) compiled program P^*, such that, $\langle P^*\,;\, Q\rangle_{\phi,p,q} = \mathsf{Compile}(\mathsf{prms}, R, L)$. Note that this implies that the state of compiled program P^* somehow encodes the state of P in a way that is transparent for ϕ, this will be used to refer to the execution state of P^* and extracting values from it. Note also that, for composed programs compiled in this way, the unnatested I/O behaviour of the second program will be identical to that of the original program.

Security

Security of labelled attested computation imposes that an adversary with control of the remote machine cannot convince the local user that some arbitrary remote (partial) execution

of a program P has occurred, when it has not. It says nothing about the parts of the execution trace that are hidden from the client *or* are not in the attested label set L^*. Formally, the adversary is allowed to freely interact with the remote machine, whilst providing a sequence of (potentially forged) attested outputs for a specific label $l \in L^*$. The adversary wins if the local user reconstructs an execution trace without aborting (i.e., all attested outputs must be accepted by the verification algorithm) and one of two conditions occur: i. there does not exist a remote process hdl^* running a compiled version of P where a consistent set of inputs was provided *for label* l; or ii. the outputs recovered by the local user for those inputs are not consistent with the semantics of P if it were run locally.

Technically, these conditions are checked in the definition by retrieving the full sequence of *label-input pairs* and random coins passed to all compiled copies of P running in the remote machine and running P on the same inputs to obtain the expected outputs. One then checks that for at least one of these executions, when the traces are restricted to special label l, that the expected trace matches the locally recovered trace via Verify. Since the adversary is free to interact with the remote machine as it pleases, one cannot hope to prevent it from providing arbitrary inputs to the remote program at arbitrary points in time, while refusing to deliver the resulting (possibly attested) outputs to the local user. This justifies the winning condition referring to a prefix of the execution in the remote machine, rather than imposing trace equality. Indeed, the definition's essence is to impose that, if the adversary delivers attested outputs for a particular label in the attested label set, then the subtrace of verified outputs for that label will be an exact prefix of the projection of the remote trace for that label.

Observe that a higher-level protocol relying on LAC will be able to fully control the semantics of labels, as these depend on the semantics of the compiled program. In particular, adopting the specific forms of parallel and sequential composition presented in Section 5.1, it is possible to use labels to get the attested execution of a sub-program that is fully isolated from other programs that it is composed with. This provides a much higher degree of flexibility than what was available in the original notion of Attested Computation.

Definition 20 (Security)**.** *A labelled attested computation scheme is secure if, for all ppt adversaries $\mathcal{A}$, the probability that experiment in Fig. 5.8 (right) returns* T *is negligible.*

Observe that the adversary loses the game as long as there exists at least one remote process that matches the locally reconstructed trace. This should be interpreted as the guarantee that IEE resources are indeed being allocated in a specific remote machine to run at least one instance of the remote program (note that if the program is deterministic, many instances could exist with exactly the same I/O behaviour, which is *not* seen as a legitimate attack). Furthermore, the presented definition imposes that the compiled program uses essentially the same randomness as the source program (except of course for randomness that the security module internally uses to provide its cryptographic functionality), as otherwise it will be

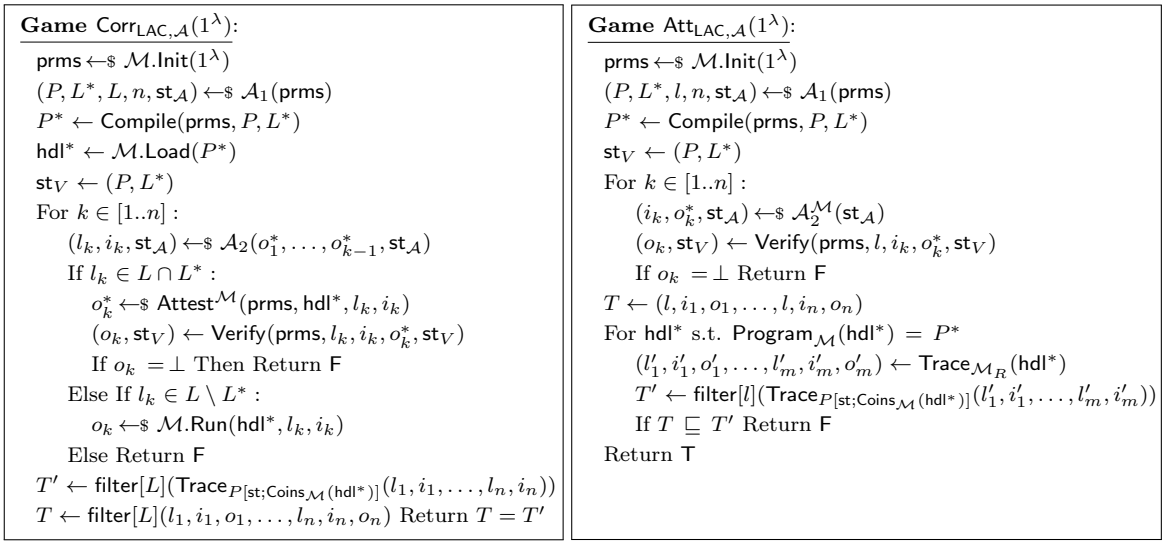


Figure 5.8: Games defining the correctness (left) and security (right) of LAC.

easy for the adversary to make the (idealized) local trace diverge from the remote. This is a consequence of this modelling approach, but in no way does it limit the applicability of the primitive proposed: it just makes it explicit that the transformation that is performed on the code for attestation will typically consist of an instrumentation of the code by applying cryptographic processing to the inputs and outputs it receives.

Minimum leakage

The above discussion shows that a LAC scheme guarantees that the I/O behaviour of the program in the remote machine includes at least the information required to reconstruct an hypothetical local execution of the source program. Next, it is required that a compiled program does not reveal any information beyond what the original program would reveal.

The following definition imposes that nothing from the internal state of the source programs (in addition to what is public, i.e., the code and I/O sequence) is leaked in the trace of the compiled program.

Definition 21 (Minimal leakage)**.** *A labelled attested computation scheme* LAC *ensures security with minimal leakage if it is secure according to Definition 20 and there exists a ppt simulator* $\mathcal{S}$ *that, for every adversary* $\mathcal{A}$*, the following distributions are identical:*

$$\{\, \mathsf{Leak\text{-}Real}_{\mathsf{LAC},\mathcal{A}}(1^\lambda) \,\} \approx \{\, \mathsf{Leak\text{-}Ideal}_{\mathsf{LAC},\mathcal{A},\mathcal{S}}(1^\lambda) \,\}$$

where games $\mathsf{Leak\text{-}Real}_{\mathsf{LAC},\mathcal{A}}$ *and* $\mathsf{Leak\text{-}Ideal}_{\mathsf{LAC},\mathcal{A},\mathcal{S}}$ *are shown in Fig. 5.9.*

Notice that the simulator is allowed to replace the global parameters of the machine with some value prms for which it can keep some trapdoor information. Intuitively this means that

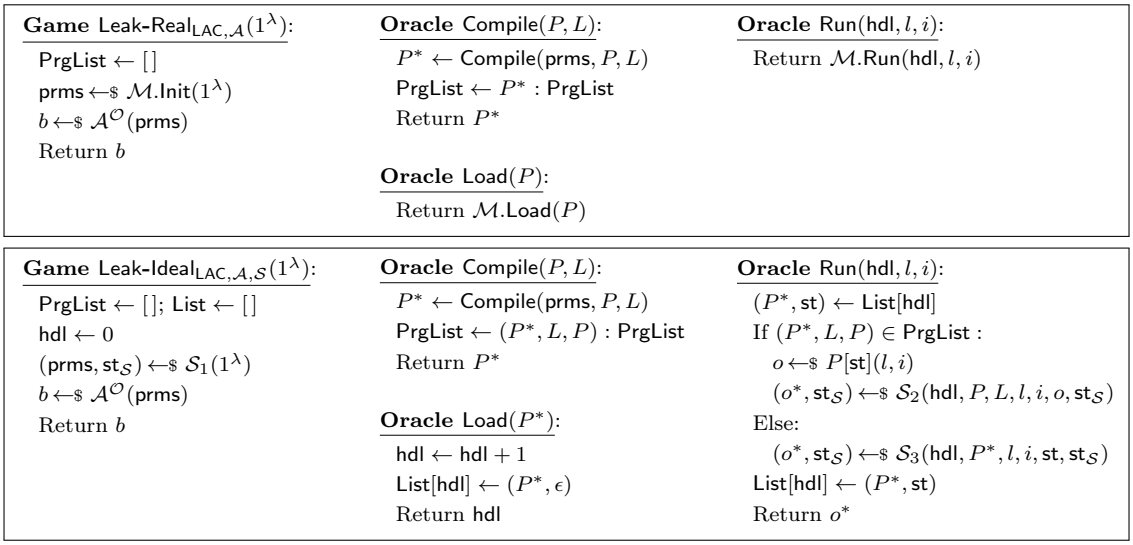


Figure 5.9: Games defining minimum leakage of LAC.

one can construct a perfect simulation of the remote trace by simply appending cryptographic material to the local trace. This property is important when claiming that the security of a cryptographic primitive is preserved when it is run within an attested computation scheme (one can simply reduce the advantage of an adversary attacking the attested trace, to the security of the original scheme using the minimum leakage simulator).

Labelled Attested Computation in practice

The protocol for labelled attested computation is very similar to the one presented in Section 5.2, with the extension of considering labels for program execution, and no longer bound by strict compositional restrictions. It is now defined a LAC scheme that relies on a remote machine supporting a security module with the same functionality as before. However, the scope is now in validating traces for specific attested labels independently from each other, rather than the full remote trace.

The proposed LAC scheme works as follows:

- Compile(prms, P, L) will generate a new program P^* and output it. Program P^* is instrumented as follows:
  - in addition to the internal state st of P, it maintains a list ios$_l$ of all the I/O pairs it has previously received and computed for each label $l \in L$.
  - On input (l, i), P^* computes $o \leftarrow\$$ P[st$_P$](l, i) and verifies if $l \in L$. If this is not the case, then P^* simply outputs non-attested output o.
  - Otherwise, it updates the list ios by appending (l, i, o), computes the subset of ios for label l : ios$_l \leftarrow$ filter$[l]$(ios) and requests from the security module a MAC of

for that list. Due to the operation of the security module, this will correspond to a tag t on the tuple (P^*, ios_l).

– It finally outputs $(o, \mathsf{t}, P^*, \mathsf{ios}_l)$. (P^*, ios_l) is included explicitly in the outputs of P^* for clarity of presentation only. This value would be kept in an insecure environment by a stateful Attest program.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, l, i)$ invokes $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, (l, i))$ using the handle and input value it has received. When the process produces an output o, Attest parses it into $(o', \mathsf{t}, P^*, \mathsf{ios}_l)$. It may happen that parsing fails, e.g., if the label is not to be attested, in which case Attest simply produces o as its own output. Otherwise, it uses $\mathcal{M}.\mathsf{Run}(0, (P^*, \mathsf{ios}_l, \mathsf{t}))$ to convert the tag into a signature σ on the same message. If this conversion fails, then Attest produces the original output o as its own output. Otherwise, it outputs (o', σ).

- $\mathsf{Verify}(\mathsf{prms}, l, i, o^*, \mathsf{st})$ is the stateful verification algorithm. The original (public) value of the state st includes uncompiled program P and the list of attested labels L (this naturally extends to including compiled program P^* since Compile is deterministic); it also includes a (initially empty) list of previously attested input-output pairs ios. Verify returns o^* if $l \notin L$. Otherwise, it first parses o^* into (o, σ), appends (l, i, o) to ios and verifies the digital signature σ using prms and $(P^*, \mathsf{filter}[l](\mathsf{ios}))$. If parsing or verification fails, Verify outputs $\perp$. If not, then Verify terminates outputting o.

Correctness

It is easy to see that the presented LAC scheme is correct, provided that the underlying signature and message authentication schemes are correct, and that it preserves the structure of compiled programs. To see that this is the case, note that during the execution of P^* for $l_k \in L$, unless a MAC or signature verification fails, the I/O sequence provided by Verify will match the one reconstructed in T' (the inputs are the same, and the associated randomness tapes are fixed by $\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)$), and therefore $T = T'$. Since these algorithms are only used for attested labels, it is only necessary to consider this possibility for labels $l \in L^* \cap L$. Now, observe that if the message authentication code scheme is correct, then the MAC verification will never fail, and if the message signature scheme is correct, then the signature verification will never fail. This is the case because the combined operations of P^*, Attest, the signing IEE running S^* and the security module lead to tags and signatures on pairs (P^*, ios_l) that exactly match the inputs provided to the verification algorithms in $\Pi.\mathsf{Ver}$ and $\Sigma.\mathsf{Verify}$. It now follows that the received trace and the reconstructed trace will be the same for all labels in L.

Security

Theorem 2. *The LAC scheme presented above provides secure attestation if the underlying MAC scheme* Π *and signature scheme* Σ *are existentially unforgeable. Furthermore, it unconditionally ensures minimum leakage.*

The proof is a sequence of two games presented in Figure 5.10 and Figure 5.11, following the same reasoning as its attested computation counterpart proof of Theorem 1. The first game is simply the LAC security game instantiated with the proposed protocol.

In game $\mathsf{G_1}^{\mathsf{AC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a sforge event occurs. Intuitively, this event corresponds to the adversary producing a signature that was not computed by the signing process with handle 0, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{Att}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{sforge}\,]\,.$$

Game $\mathbf{G0}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$

$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
  Parse $(o_k, \sigma) \leftarrow o_k^*$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
    $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
  Else: Return F

$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
  $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return F
Return T

Game $\mathbf{G1}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
  Parse $(o_k, \sigma) \leftarrow o_k^*$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
    $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
  Else: Return F
  If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
    $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
  $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return F
Return T

Figure 5.10: First game hop for the proof of security of the LAC protocol.

The distance between these two games is upper bound, by constructing an adversary $\mathcal{B}$ against the existential unforgeability of signature scheme Σ in S^* such that

$$\Pr[\,\mathsf{sforge}\,] \leq \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{B}}(\lambda)$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G_1}^{\mathsf{LAC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the signing operations performed within the process loaded by the security module are replaced with calls to the Sign oracle provided in the

existential unforgeability game. More precisely, whenever process 0 in the remote machine is expected to compute a signature on message msg, algorithm $\mathcal{B}$ calls its own oracle on (P^*, msg) to obtain σ.

When sforge is set, according to the rules of game $\mathsf{G}_1{}^{\mathsf{LAC},\mathcal{A}}$, algorithm $\mathcal{B}$ outputs message $(P^*, \mathsf{filter}[l](\mathsf{ios}))$ and candidate signature σ. It remains to show that this is a valid forgery. To see this, first observe that this is indeed a valid signature, as signature verification is performed on these values immediately before sforge occurs. It suffices to establish that message $(P^*, (l, i_1, o_1, \ldots, l, i_k, o_k))$ could not have been queried from the Sign oracle. Access to the signing key that allows signatures to be performed is only permitted to the special process with handle 0. From the construction of S^*, is is known that producing such a signature would only occur via the inclusion of $(P^*, (l, i_1, o_1, \ldots, l, i_k, o_k))$ in its trace. Since this is not the case, $(P^*, \mathsf{filter}[l](\mathsf{ios}))$ could not have been queried from the signature oracle. It can therefore be concluded that $\mathcal{B}$ outputs a valid forgery whenever sforge occurs.

In game $\mathsf{G}_2{}^{\mathsf{LAC},\mathcal{A}}(1^\lambda)$, the adversary loses whenever a mforge event occurs. Intuitively, this event corresponds to the adversary producing a tag that was not computed by the security module, and hence constitutes a forgery with respect to Π. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G}_1{}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G}_2{}^{\mathsf{LAC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{mforge}\,].$$

Game $\mathbf{G1}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
  Parse $(o_k, \sigma) \leftarrow o_k^*$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
    $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
  Else: Return F
  If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
    $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
  $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return F
Return T

Game $\mathbf{G2}_{\mathsf{LAC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, L^*, l, n, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$\mathsf{sforge} \leftarrow \mathsf{F}$; $\mathsf{mforge} \leftarrow \mathsf{F}$
$P^* \leftarrow \mathsf{Compile}(\mathsf{prms}, P, L^*)$
For $k \in [1..n]$:
  $(i_k, o_k^*, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_2^{\mathcal{M}}(\mathsf{st}_V, \mathsf{st}_\mathcal{A})$
  Parse $(o_k, \sigma) \leftarrow o_k^*$
  If $\Sigma.\mathsf{Vrfy}(\mathsf{prms}, \sigma, (P^*, \mathsf{filter}[l]((l, i_k, o_k) : \mathsf{ios})))$:
    $\mathsf{ios} \leftarrow ((l, i_k, o_k) : \mathsf{ios})$
  Else: Return F
  If $((P^*, (l, i_1, o_1, \ldots, l, i_k, o_k), \star), \sigma') \notin \mathsf{Trace}_\mathcal{M}(0)$:
    $\mathsf{sforge} \leftarrow \mathsf{T}$; Return F
  If $\nexists\, \mathsf{hdl}^*.\ \mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^* \wedge$
    $(l, i_1, o_1, \ldots, l, i_k, o_k) \sqsubseteq \mathsf{filter}[l]$
      $(\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(\mathsf{Trace}_\mathcal{M}(\mathsf{hdl}^*)))$:
    Then $\mathsf{mforge} \leftarrow \mathsf{T}$; Return F
$T \leftarrow (l, i_1, o_1, \ldots, l, i_n, o_n)$
For hdl^* s.t. $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}^*) = P^*$
  $(l'_1, i'_1, o'_1, \ldots, l'_m, i'_m, o'_m) \leftarrow \mathsf{Trace}_{\mathcal{M}_R}(\mathsf{hdl}^*)$
  $T' \leftarrow \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(l'_1, i'_1, \ldots, l'_m, i'_m))$
  If $T \sqsubseteq T'$ Return F
Return T

Figure 5.11: Second game hop for the proof of security of the LAC protocol.

The distance between these two games is upper bound, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of MAC scheme Π in the security module such that

$$\Pr[\,\mathsf{mforge}\,] \leq \mathsf{Adv}^{\mathsf{Auth}}_{\Pi,\mathcal{C}}(\lambda)$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$ as follows: the operation of machine $\mathcal{M}$ is simulated exactly with the caveat that the MAC operations computed inside the internal security module are replaced with calls to the Auth oracle provided in the existential unforgeability game. More precisely, whenever a process running code R^* within an IEE in the remote machine requests a MAC on message msg from the security module, algorithm $\mathcal{C}$ calls its own oracle on (P^*, msg) to obtain t.

Let $T \leftarrow (l, i_1, o_1, \ldots, l, i_k, o_k)$. When mforge is set according to the rules of game $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$, algorithm $\mathcal{C}$ retrieves the trace of the process with handle 0 running S^*, locates the input/output pair $(((P^*, T), \mathsf{t}), \sigma')$ and outputs message (P^*, T) and candidate tag t. To see this is a valid forgery, first observe that, having failed the sforge check, one knows that $(((P^*, T), \mathsf{t}), \sigma')$ is in the trace of the process with handle 0, so by its construction it is also known that the corresponding input $((P^*, T), \mathsf{t})$ must contain a valid tag. It suffices to establish that message (P^*, T) could not have been queried from the Auth oracle. Suppose that the first part of the mforge check failed, i.e., that $\nexists\, \mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}^*) = P^*$. Then, because the security module signs the code of the processes requesting the signatures, we are sure that such a query was never placed to the Auth oracle. Furthermore, any MAC query for a message starting with P^* must have been caused by the execution of an instance of P^*. Now suppose some instances of P^* were indeed running in the remote machine, but that none of them displayed the property $(l, i_1, o_1, \ldots, l, i_k, o_k) \sqsubseteq \mathsf{filter}[l](\mathsf{Translate}(\mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl}^*)))$. Then, by the construction of P^*, we can also exclude that (P^*, T) was queried from the MAC oracle. As such, it can be concluded that $\mathcal{C}$ outputs a valid forgery whenever mforge occurs.

To complete the proof, it must be shown that the adversary never wins in game $\mathsf{G_2}^{\mathsf{LAC},\mathcal{A}}$. To see this, observe that the game reaching the final check implies that

$$\exists\, \mathsf{hdl}^*.\ \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}^*) = P^* \wedge$$
$$(l, i_1, o_1, \ldots, l, i_n, o_n) \sqsubseteq \mathsf{filter}[l](\mathsf{Trace}_{P[\mathsf{st};\mathsf{Coins}_{\mathcal{M}}(\mathsf{hdl}^*)]}(\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl}^*))$$

Which exactly matches the final criteria of $T \sqsubseteq T'$.

To finish the proof, it must now be shown that this scheme also provides security with minimum leakage. This implies defining a ppt simulator $\mathcal{S}$ that provides identical distributions with respect to experiment in Figure 5.9. This is easy to ascertain given the simulator behaviour described in Figure 5.12: $\mathcal{S}_1$ and $\mathcal{S}_3$ follow the exact description of the actual machine, modulo the generation of $(\mathsf{pk}, \mathsf{sk})$ and key. $\mathcal{S}_2$ takes an external output produced by $P[\mathsf{st}](l, i)$ and returns an output in accordance to the behaviour of $\mathcal{M}$, which given language $\mathcal{L}$ may differ from a real output only by the random coins. As such, the distribution provided

by the simulator is indistinguishable to the one provided by a real machine, and the claim follows.

□

Simulator $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$

Simulator $\mathcal{S}$ will perform according to the $\mathcal{M}$ execution description.

- Upon input 1^λ, $\mathcal{S}_1$ generates a key pair for process S^*, a MAC key for the security module and initializes the traces and states as an empty list. The public key will be the public parameters, while the secret key be stored in its initial state.

$\underline{\mathcal{S}_1(1^\lambda)}$:
  $\mathsf{key} \leftarrow\$\ \Pi.\mathsf{Gen}(1^\lambda)$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \Sigma.\mathsf{Gen}(1^\lambda)$
  $\mathsf{Traces} \leftarrow [\,]$
  Return $(\mathsf{pk}, (\mathsf{key}, \mathsf{sk}, \mathsf{Traces}))$

- $\mathcal{S}_2$ maintains a list of traces Traces with the respective list ios and stage stage. Given this, it masks output o^* as if produced by an actual machine execution.

$\underline{\mathcal{S}_2(\mathsf{hdl}, P, L, l, i, o^*, \mathsf{st}_\mathcal{S})}$:
  Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
  If $\mathsf{Traces}[\mathsf{hdl}] \neq [\,]$: $\mathsf{ios} \leftarrow \mathsf{Traces}[\mathsf{hdl}]$
  Else: $\mathsf{ios} \leftarrow [\,]$
  If $l \in L^*$:
    $\mathsf{ios} \leftarrow (l, i, o^*) : \mathsf{ios}$
    $\mathsf{msg} \leftarrow (o^*, \Pi.\mathsf{Mac}(\mathsf{key}, P, \mathsf{filter}[l](\mathsf{ios})))$
  Else: $\mathsf{msg} \leftarrow o^*$
  $\mathsf{Traces}[\mathsf{hdl}] \leftarrow \mathsf{ios}$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
  Return $(\mathsf{msg}, \mathsf{st}_\mathcal{S})$

- $\mathcal{S}_3$ computes the next output given input i, program R and state st. $\mathsf{Sim}(P^*[\mathsf{st}], l, i, \mathsf{ios}, \mathsf{key})$ refers to the operation of executing P^* with label l and input i for trace ios and considering a security module with key key, producing output msg and updated trace ios. The result is afterwards treated similar to $\mathcal{S}_2$.

$\underline{\mathcal{S}_3(\mathsf{hdl}, P^*, l, i, \mathsf{st}, \mathsf{st}_\mathcal{S})}$:
  Parse $(\mathsf{key}, \mathsf{sk}, \mathsf{Traces}) \leftarrow \mathsf{st}_\mathcal{S}$
  If $\mathsf{hdl} = 0$:
    Parse $(\mathsf{msg}, \mathsf{t}) \leftarrow i^*$
    If $\Pi.\mathsf{Ver}(\mathsf{key}, \mathsf{t}, \mathsf{msg})$: Return $\Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{msg})$
    Else Return $\perp$
  If $\mathsf{Traces}[\mathsf{hdl}] \neq [\,]$: $\mathsf{ios} \leftarrow \mathsf{Traces}[\mathsf{hdl}]$
  Else: $\mathsf{ios} \leftarrow [\,]$
  $L \leftarrow \mathsf{labels}(P^*)$
  $(\mathsf{msg}, \mathsf{ios}) \leftarrow\$\ \mathsf{Sim}(P^*[\mathsf{st}], l, i, \mathsf{ios}, \mathsf{key})$
  $\mathsf{Traces}[\mathsf{hdl}] \leftarrow \mathsf{ios}$; $\mathsf{st}_\mathcal{S} \leftarrow (\mathsf{key}, \mathsf{sk}, \mathsf{Traces})$
  Return $(\mathsf{msg}, \mathsf{st}_\mathcal{S})$

Figure 5.12: Description of simulator $\mathcal{S}$

Chapter 6

Secure Channels to Isolated Execution Environments

In this chapter, we make use of the proposed primitives Attested Computation and Labelled Attested Computation as stepping stones to achieve specific key-exchange mechanisms that rely on said hardware assumptions and respective utility theorems. These results can then be employed as building blocks in the construction of more ambitious protocol implementations.

6.1 Attested Key Exchange for Attested Computation

An intermediate step in constructing high-level applications that rely on the proposed mechanisms for attestation is the establishment of a secure communications channel with a process running a particular program inside an IEE in the remote machine. After such a channel has been established, standard cryptographic techniques can be used to ensure (in combination with the isolation provided by IEEs) the integrity and confidentiality of subsequent computations. This section presents how one can combine attested computation with a specific flavour of a key exchange protocol for constructing a bootstrapping process to enable this scenario.

First, the precise requirements for a key exchange protocol that can be used in this setting are formalised (this is called *authenticated key exchange for attested computation*) and it is shown how a simple transformation can be used to construct such protocols from any passively secure key exchange protocol. Later on, various utility theorems will be defined, precisely describing what it means to use attested computation (or labelled attested computation) to establish a secure channel with an arbitrary remote program.

Syntax

A *Key Exchange for Attested Computation* (AttKE) protocol is defined by the following pair of algorithms.

- $\mathsf{Setup}(1^\lambda, \mathsf{id})$ is the remote program generation algorithm, which is run on the local machine to initialise a fresh instance of the AttKE protocol under party identifier id. On input the security parameter and id, it will output the code for a program $\mathsf{Rem}_{\mathsf{KE}}$ and the initial state st_L of the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm. This algorithm is run locally.
- $\mathsf{Rem}_{\mathsf{KE}}$ (which is generated dynamically by Setup) is a program that will be run as a part of an IEE process in the remote machine, and it will keep the entire remote state of the key exchange protocol in that protected environment.
- $\mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_L, \mathsf{msg})$ is the algorithm that runs the local end of the AttKE protocol, interacting with $\mathsf{Rem}_{\mathsf{KE}}$. On input its current state and an incoming message msg, it will output an updated state and an outgoing message.

When analysing the security of such a protocol it is imposed that the $\mathsf{Loc}_{\mathsf{KE}}$ algorithm and all $\mathsf{Rem}_{\mathsf{KE}}$ programs that may be produced by Setup keep in their state the same information that was required for general key exchange algorithms (described in Section 2.1). Let the instances of local key exchange executions be referred to as $\mathsf{Loc}^s_{\mathsf{KE}}$, for $s \in \mathbb{N}$. The local identity will be implicit in this notation since the following discussion is focused on the case where a single local identity id is considered. This is done for the sake of rigour and clarity of presentation: by looking at this simplified case, one can present the following security models in game-based form, whilst taming the complexity of the resulting games. The extension of these results to the more general case where several local identities are considered is straightforward. On the remote side, the identity of the remote process will actually be generated on the fly by the combined actions of the Setup algorithm and possibly the protocol execution itself, as it may depend for example on the code of the remote program. For this reason the remote instances will be enumerated as $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$ for $i, j \in \mathbb{N}$, and observe that the value of variable oid in this case will be set during the execution of the program itself, rather than passed explicit as an input to one of the algorithms.

Correctness

An AttKE is correct if, after a complete (honest) run between two participants, one local and one remote, and where the remote program is always the one to initiate the communication, both reach the accept state, both derive the same key and session identifier and have matching partner identities. More formally, a protocol $P = \{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}\}$ is correct if, for any arbitrary identity id, the experiment in Figure 6.1 always returns T. It should be noted

Game $\mathsf{Corr}_{\mathsf{AttKE}}(1^\lambda)$:
$(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{st}_R \leftarrow \epsilon$
$\mathsf{msg} \leftarrow\$\ \mathsf{Rem}_{\mathsf{KE}}[\mathsf{st}_R](\epsilon)$
$\mathsf{t} \leftarrow \mathsf{T}$
While $\mathsf{msg} \neq \epsilon$:
  If t: $(\mathsf{st}_L, \mathsf{msg}) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_L, \mathsf{msg})$
  Else: $\mathsf{msg} \leftarrow\$\ \mathsf{Rem}_{\mathsf{KE}}[\mathsf{st}_R](\mathsf{msg})$
  $\mathsf{t} \leftarrow \neg\mathsf{t}$
Return $\mathsf{st}_L.\delta = \mathsf{st}_R.\delta = \mathsf{accept} \wedge \mathsf{st}_L.\mathsf{key} = \mathsf{st}_R.\mathsf{key} \wedge \mathsf{st}_L.\mathsf{sid} = \mathsf{st}_R.\mathsf{sid} \wedge$
  $\mathsf{st}_L.\mathsf{pid} = \mathsf{st}_R.\mathsf{oid} \wedge \mathsf{st}_L.\mathsf{oid} = \mathsf{st}_R.\mathsf{pid} \wedge \mathsf{st}_L.\mathsf{oid} = \mathsf{id}$

Figure 6.1: Game defining the correctness of an AttKE scheme.

that the presented definition of correctness imposes that remote programs always operate as initiators and local machines as the responders in the key exchange.

Execution Environment

The specific flavour of key exchange considered is clarified by the execution environment in Figure 6.2. This follows the standard modelling of active attackers, e.g. [KY03], when one excludes the possibility of corruption (which it is done solely for the sake of simplicity). There are, however, two modifications that attend to the fact that AttKE remote programs are designed to be executed under attested computation guarantees. On one hand, the adversary is given the power to create as many remote AttKE programs as it may need, by using the NewLocal oracle, revealing the entire code of the remote AttKE program (and implicitly all of its initial internal state, which is assumed to be empty) to the adversary. This captures the fact that remote AttKE programs will be loaded into IEEs in an otherwise untrusted remote machine, and it implies that remote AttKE programs cannot keep *any* long term secret information. Intuitively, this limitation will be compensated by the attested computation protocol. On the other hand, the adversary is able to freely interact with remote processes, but it is constrained in its interaction with the local machine. Indeed the SendLocal oracle filters which messages the adversary can deliver to the local machine by checking that these are consistent with at least one remote process that the adversary is interacting with. This captures the fact that AttKE is designed to interact over a partially authenticated channel from the remote machine to the local machine, which will be provided by an attested computation protocol.

Partnering

The natural extension of the partnering properties introduced for passive key exchange in Section 2.1 to the AttKE setting is considered. In addition to the syntactic modifications that result from referring to $\mathsf{Loc}^s_{\mathsf{KE}}$ and $\mathsf{Rem}^{i,j}_{\mathsf{KE}}$, validity is further restricted so that partnering

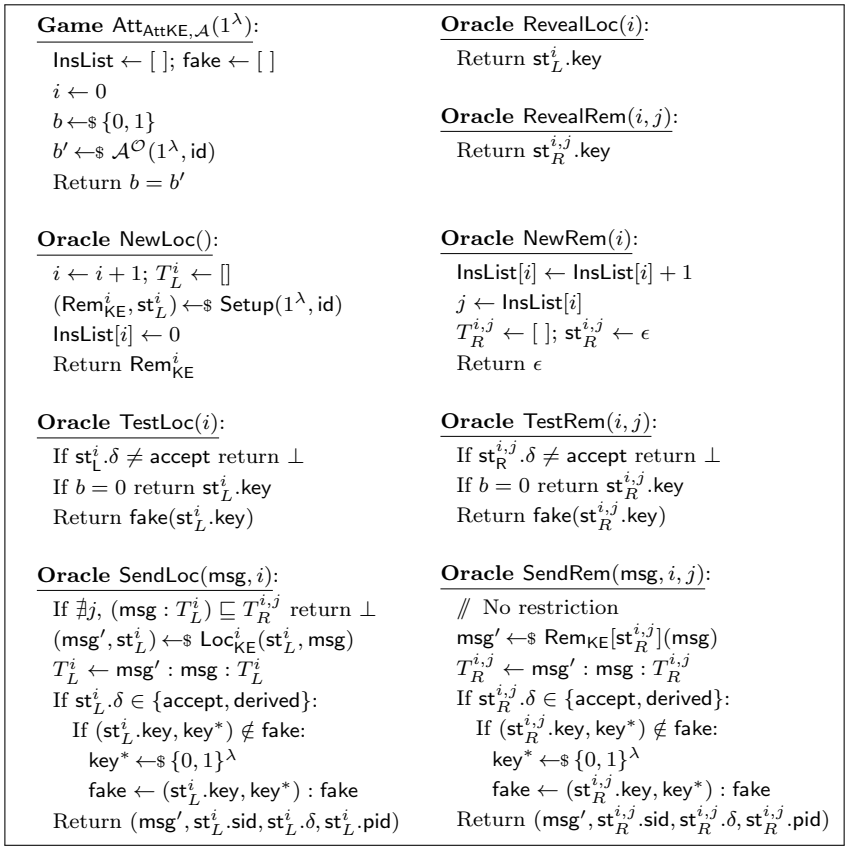


Figure 6.2: Execution environment for AttKEs.

is only valid when it occurs between local and remote instances, in which the latter is the initiator. To this end, the following predicate on two instances $\mathsf{Loc}_{\mathsf{KE}}^s$ and $\mathsf{Rem}_{\mathsf{KE}}^{i,j}$ holding $\mathsf{st}_L^s = (\mathsf{st}^s, \delta^s, \rho^s, \mathsf{sid}^s, \mathsf{pid}^s, \mathsf{oid}^s, \mathsf{key}^s)$ and $\mathsf{st}_R^{i,j} = (\mathsf{st}^{i,j}, \delta^{i,j}, \rho^{i,j}, \mathsf{sid}^{i,j}, \mathsf{pid}^{i,j}, \mathsf{oid}^{i,j}, \mathsf{key}^{i,j})$ is used, respectively:

$$\mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \begin{cases} \mathsf{T} \text{ if } \mathsf{sid}^s = \mathsf{sid}^{i,j} \ \wedge \ \delta^s, \delta^{i,j} \in \{\mathsf{derived}, \mathsf{accept}\} \\ \mathsf{F} \text{ otherwise.} \end{cases}$$

The definition of partner is the obvious one, whereas invalid partners now includes an extra possibility.

Definition 22 (Partner)**.** *Two instances* $\mathsf{Loc}_{\mathsf{KE}}^s$ *and* $\mathsf{Rem}_{\mathsf{KE}}^{i,j}$ *are partnered if*

$$\mathrm{P}(\mathsf{Loc}_{\mathsf{KE}}^s, \mathsf{Rem}_{\mathsf{KE}}^{i,j}) = \mathsf{T}\,.$$

Definition 23 (Valid Partners)**.** *A protocol* AttKE *ensures valid partners if the bad event*

notval *does not occur, where* notval *is defined as one of the following events occurring:*

$$\begin{aligned}
\exists \mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}} \ s.t.\ \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge (\ \mathsf{pid}^{s} \neq \mathsf{oid}^{i,j} \ \vee \ \mathsf{oid}^{s} \neq \mathsf{pid}^{i,j} \vee \\
\mathsf{key}^{s} \neq \mathsf{key}^{i,j} \ \vee \ \rho^{s} \neq \mathsf{responder} \ \vee \ \rho^{i,j} \neq \mathsf{initiator}\ ) \\
\exists \mathsf{Loc}^{r}_{\mathsf{KE}}, \mathsf{Loc}^{s}_{\mathsf{KE}} \ s.t.\ r \neq s \wedge \mathrm{P}(\mathsf{Loc}^{r}_{\mathsf{KE}}, \mathsf{Loc}^{s}_{\mathsf{KE}}) = \mathsf{T} \\
\exists \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{k,l}_{\mathsf{KE}} \ s.t.\ (i,j) \neq (k,l) \wedge \mathrm{P}(\mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{k,l}_{\mathsf{KE}})
\end{aligned}$$

For completeness, adapted definitions of confirmed and unique partners are also presented.

Definition 24 (Confirmed Partners)**.** *A protocol* AttKE *ensures confirmed partners if the bad event* notconf *does not occur, where* notconf *is defined as at least one of the following two events occurring:*

$$\begin{aligned}
\exists \mathsf{Loc}^{s}_{\mathsf{KE}} \ s.t.\ \delta^{s} = \mathsf{accept} \quad \wedge \quad \forall\, \mathsf{Rem}^{i,j}_{\mathsf{KE}},\ \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{F} \\
\exists \mathsf{Rem}^{i,j}_{\mathsf{KE}} \ s.t.\ \delta^{i,j} = \mathsf{accept} \quad \wedge \quad \forall\, \mathsf{Loc}^{s}_{\mathsf{KE}},\ \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{F}.
\end{aligned}$$

Definition 25 (Unique Partners)**.** *A protocol* AttKE *ensures unique partners if the bad event* notuni *does not occur, where* notuni *is defined as at least one of the following two events occurring:*

$$\begin{aligned}
\exists \mathsf{Loc}^{s}_{\mathsf{KE}}, \quad & \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \mathsf{Rem}^{i',j'}_{\mathsf{KE}} \ \ s.t. \\
& (i,j) \neq (i',j') \wedge \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i',j'}_{\mathsf{KE}}) = \mathsf{T} \\
\exists \mathsf{Rem}^{i,j}_{\mathsf{KE}}, \quad & \mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Loc}^{s'}_{\mathsf{KE}} \ \ s.t. \\
& s \neq s' \wedge \mathrm{P}(\mathsf{Loc}^{s}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T} \wedge \mathrm{P}(\mathsf{Loc}^{s'}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T}\,.
\end{aligned}$$

As before, an adversary is considered to violate entity authentication if he can get a session to accept, but there is no unique and confirmed valid session in its intended partner. More formally, one wishes to verify that none of the bad events notval, notconf, notuni occurs. In the attested computation scenario, it is common to use one-sided authentication where only the local party receives authentication guarantee. Such definitions can be easily derived from the ones presented above, analogously to what was done in Section 2.1.

Security

Again, the set of TestLoc and TestRem queries must be restricted in order to exclude trivial attacks. An adversary is legitimate if it respects the following freshness criteria:

- For all $\mathsf{TestLoc}(i)$ queries, the following holds: 1. $\mathsf{RevealLoc}(i)$ was not queried; and 2. for all $\mathsf{Rem}^{j,k}_{\mathsf{KE}}$ s.t. $\mathrm{P}(\mathsf{Rem}^{j,k}_{\mathsf{KE}}, \mathsf{Loc}^{s}_{\mathsf{KE}}) = \mathsf{T}$, $\mathsf{RevealRem}(j,k)$ was not queried.
- For all $\mathsf{TestRem}(i,j)$ queries, the following holds: 1. $\mathsf{RevealRem}(i,j)$ was not queried; and 2. for all $\mathsf{Loc}^{k}_{\mathsf{KE}}$ s.t. $\mathrm{P}(\mathsf{Loc}^{k}_{\mathsf{KE}}, \mathsf{Rem}^{i,j}_{\mathsf{KE}}) = \mathsf{T}$, $\mathsf{RevealLoc}(i)$ was not queried.

It established that the winning event guess occurs if $b = b'$ at the end of the experiment. AttKE security is defined by requiring both mutual authentication of parties and key secrecy.

Definition 26 (AttKE security)**.** *An AttKE protocol is secure if, for any ppt adversary in Figure 6.2, and for any local party identifier string* id*: 1. the adversary violates entity authentication with negligible probability* $\Pr[\,\mathsf{notval} \vee \mathsf{notconf} \vee \mathsf{notuni}\,]$*; and 2. its key secrecy advantage* $2 \cdot \Pr[\,\mathsf{guess}\,] - 1$ *is negligible.*

Generic Construction

It is now presented a construction of an AttKE scheme from any passively secure key exchange protocol, relying additionally on a existentially unforgeable signature scheme. The intuition here is that the attested computation protocol guarantees correct remote execution of a program, but does not ensure uniqueness, i.e., it does not exclude that potentially many replicas of the same key exchange protocol instance could be running in the remote machine. By binding a fresh signature verification key with the identifier for the remote party associated with the key exchange protocol and generating a fresh nonce at the start of every execution, one can remotely execute the key exchange code whilst ensuring one-to-one authentication at the process level. This transformation can be seen as a weaker version of the well-known passive-to-active compilation process by Katz et al. [KY03], since the specific target security model is not fully active

Consider a passively-secure authenticated key exchange protocol Π and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. This construction splits the execution of Π between the local machine and a remote isolated execution environment: the responder will run locally and the initiator will run remotely within a program $\mathsf{Rem}_{\mathsf{KE}}$.[1] The code of the remote program will have hard-wired into it a unique verification key for the signature scheme. The first activation of $\mathsf{Rem}_{\mathsf{KE}}$ initialises an internal state and computes a nonce, together with the first message in the key exchange protocol. The party identifier string of the remote process will then be defined to comprise the verification key and the nonce. The local part of the protocol signs the full communication trace so far. Subsequent activations of remote program $\mathsf{Rem}_{\mathsf{KE}}$ will simply respond according to the key exchange protocol description, rejecting all inputs that fail signature verification. The details of this construction are shown in Figure 6.3.

- Setup first generates a fresh key pair for the signature scheme and constructs program $\mathsf{Rem}_{\mathsf{KE}}$, parametrised by algorithm Π and verification key pk, as described in Figure 6.3 (top). In this program, state variables δ, ρ, key, sid and pid are all shared with Π (this is implicit in the figure). The initial value of st_L will store id, along with the initially

[1] Setting the remote machine as the initiator of the protocol is the most common scenario. This is considered for simplicity; the converse can be treated analogously.

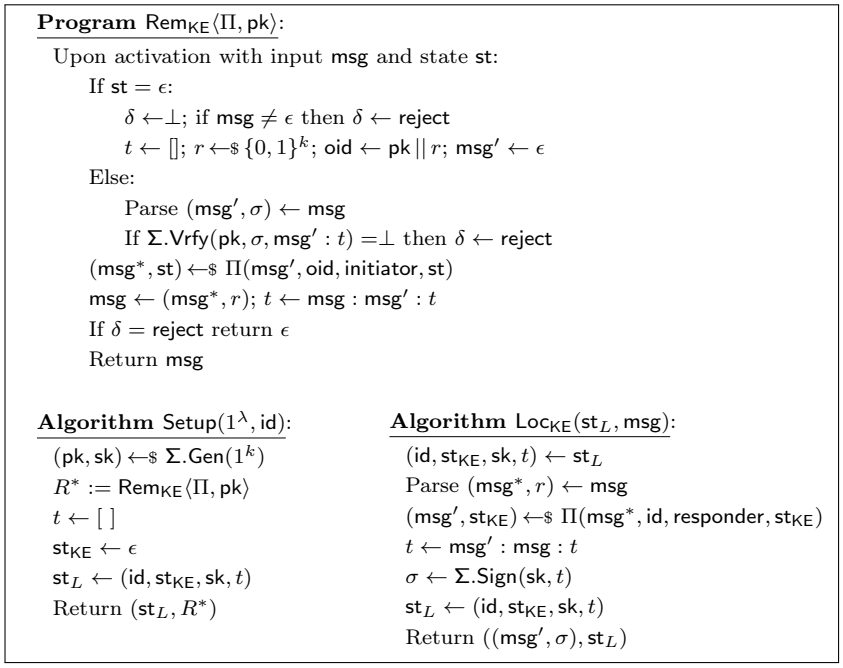


Figure 6.3: Details of the AttKE construction.

empty state for the key exchange $\mathsf{st}_K E$, the signing key for the signature scheme and an initially empty trace t log.

- $\mathsf{Loc}_{\mathsf{KE}}$ takes $(\mathsf{st}_L, \mathsf{msg})$, runs $\Pi(\mathsf{msg}, \mathsf{id}, \mathsf{responder}, \mathsf{st}_{\mathsf{KE}})$ to compute the next message o, produces signature σ of the entire updated protocol trace, and returns the updated state st_L and message (o, σ).

The following theorem establishes the correctness and security of the generic construction.

Theorem 3. *Given a correct passively secure key exchange protocol* Π *and an existentially unforgeable signature scheme* Σ*, the generic construction above yields a correct and secure AttKE protocol.*

The proof for this theorem will follow the intuition of Katz and Yung for Theorem 1 in [KY03]. It starts by bounding the probabilities for the occurrence of bad events ($\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$), and then argue that the behaviour of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ to an adversary is the same as the one of $\mathsf{G4}^{\Pi,\mathcal{A}}$ using a passive adversary for the original protocol Π. At this final stage one can show that, given correctness and security guarantees of Π, a correct and secure AttKE protocol is achieved. The proof consists in a sequence of five games presented in Figures 6.4 to 6.8.

The first game is simply the AttKE security game in Figure 6.2 instantiated with the construction detailed in Figure 6.3.

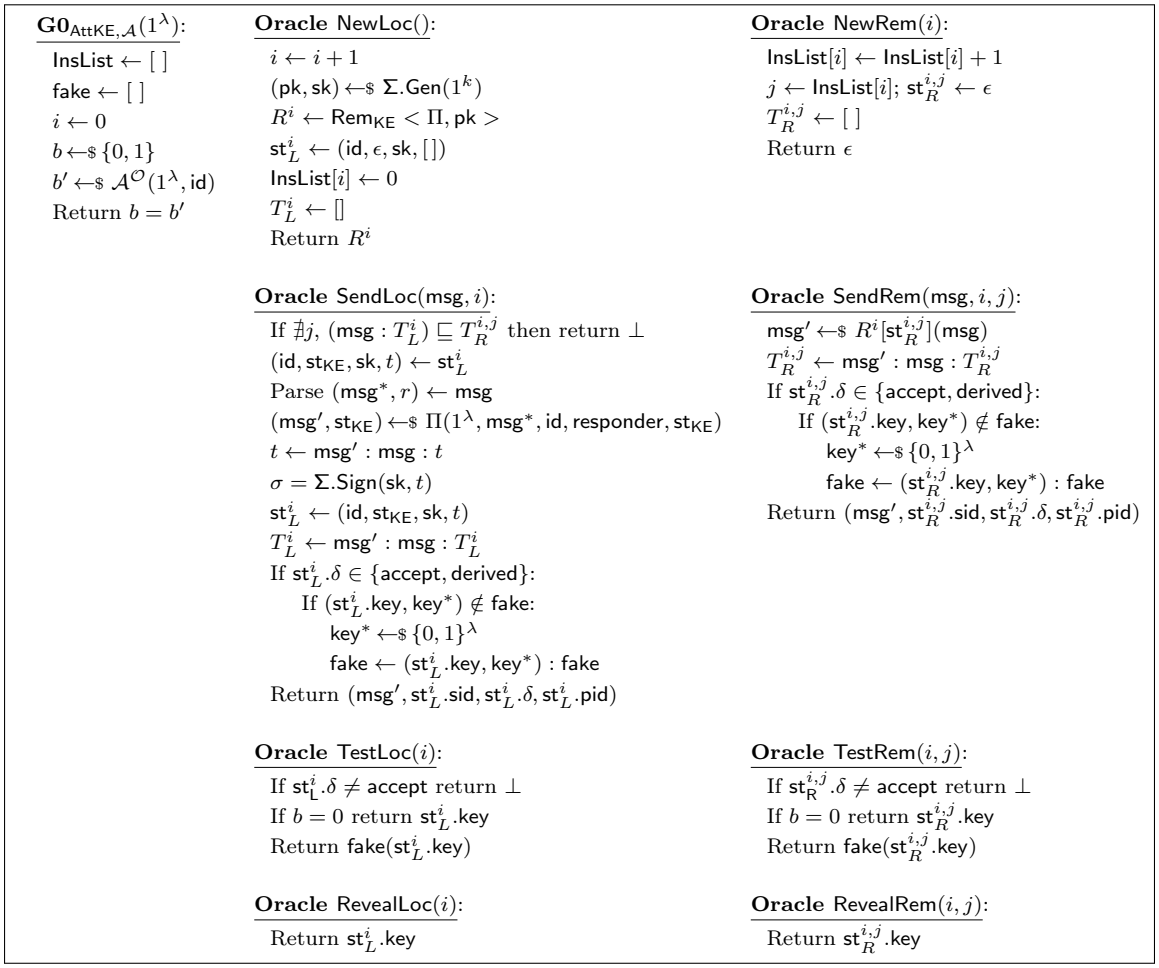


Figure 6.4: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

In the second game $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a repeat event occurs. Intuitively, this event corresponds to the adversary generating two sessions with the same key pair, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{repeat}\,]\,.$$

Let q be the maximum number of calls to $\mathsf{NewLocal}$ allowed. The distance between these two games is upper bound, by constructing an adversary $\mathcal{B}$ against the existential unforgeability of signature scheme Σ such that

$$\Pr[\,\mathsf{repeat}\,] \;\leq\; \frac{\mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{B}}(\lambda) * \mathsf{q}}{2}\,.$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{B}$ has to try and guess which session will have a duplicate key. As such, it samples

uniformly from [1..q] a session s and replaces the public key generated by $\Sigma.\mathsf{Gen}$ in NewLocal for instance $i = s$ with the public key $\mathsf{pk}_{\mathsf{UF}}$ provided by $\mathsf{UF}^{\Sigma,\mathcal{B}}$. Every time instance s has to produce a signature in $\mathsf{SendLocal}(\mathsf{msg}, s)$, instead of $\Sigma.\mathsf{Sign}(\mathsf{sk}, t)$, $\mathcal{B}$ calls Oracle $\mathsf{Sign}(t)$. Additionally, $\mathcal{B}$ will store all key pairs and session ids generated by NewLocal s.t. $i \neq s$ in a list keys. When the game terminates, the adversary looks up on list keys for a key pair with public key $\mathsf{pk}_{\mathsf{UF}}$, extracts the associated secret key sk, executes $\sigma \leftarrow\!\!\$\ \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{msg}')$ for any msg' not yet queried to Oracle Sign and presents (msg', σ) as a challenge for $\mathsf{UF}^{\Sigma,\mathcal{B}}$. It remains to show that, when repeat is set, $\mathcal{B}$ wins $\mathsf{UF}^{\Sigma,\mathcal{B}}$ with probability $2/\mathsf{q}$

When the game ends and $\mathsf{repeat} = \mathsf{T}$, there are at least two duplicate R. This implies that $\exists i, j$ s.t. $(\mathsf{pk}_i, \mathsf{sk}_i) = (\mathsf{pk}_j, \mathsf{sk}_j), i \neq j$. If it is the case that $i = s \vee j = s$, $\mathcal{B}$ has either sk_i or sk_j in keys, and can use that to generate the signature that wins the $\mathsf{UF}^{\Sigma,\mathcal{B}}$ game. Since $i, j \in [1..\mathsf{q}]$ and s is sampled uniformly from [1..q], this happens with probability $2/\mathsf{q}$, and one can therefore conclude that $\mathcal{B}$ outputs a valid forgery with the same probability whenever repeat occurs.

In game $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a rnonce event occurs. Intuitively, this event corresponds to the adversary generating two duplicate nonces. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{rnonce}\,]\,.$$

Let q_R be the number of calls to NewRemote allowed to the adversary. The distance between these two games is upper bound such that

$$\Pr[\,\mathsf{rnonce}\,] \;\leq\; \frac{\mathsf{q}_R^2}{2^k}\,.$$

From the rules of $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ and the construction of R, it is known that a new r is generated at every query of SendRemote such that $T_R^{i,j} = [\,]$. This only happens at most once for every new $T_R^{i,j}$, i.e., at most once for every call of NewRemote. Since r is sampled uniformly from a subset of $\{0,1\}^k$, one can conclude that the probability of rnonce is $\mathsf{q}_R^2/2^k$.

In game $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a forge event occurs. Intuitively, this event corresponds to the adversary producing a signature that was not computed by SendLocal, and hence constitutes a forgery with respect to Σ. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{forge}\,]\,.$$

Let q be the maximum number of calls to NewLocal allowed. The distance between these two games is upper bound, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of signature scheme Σ such that

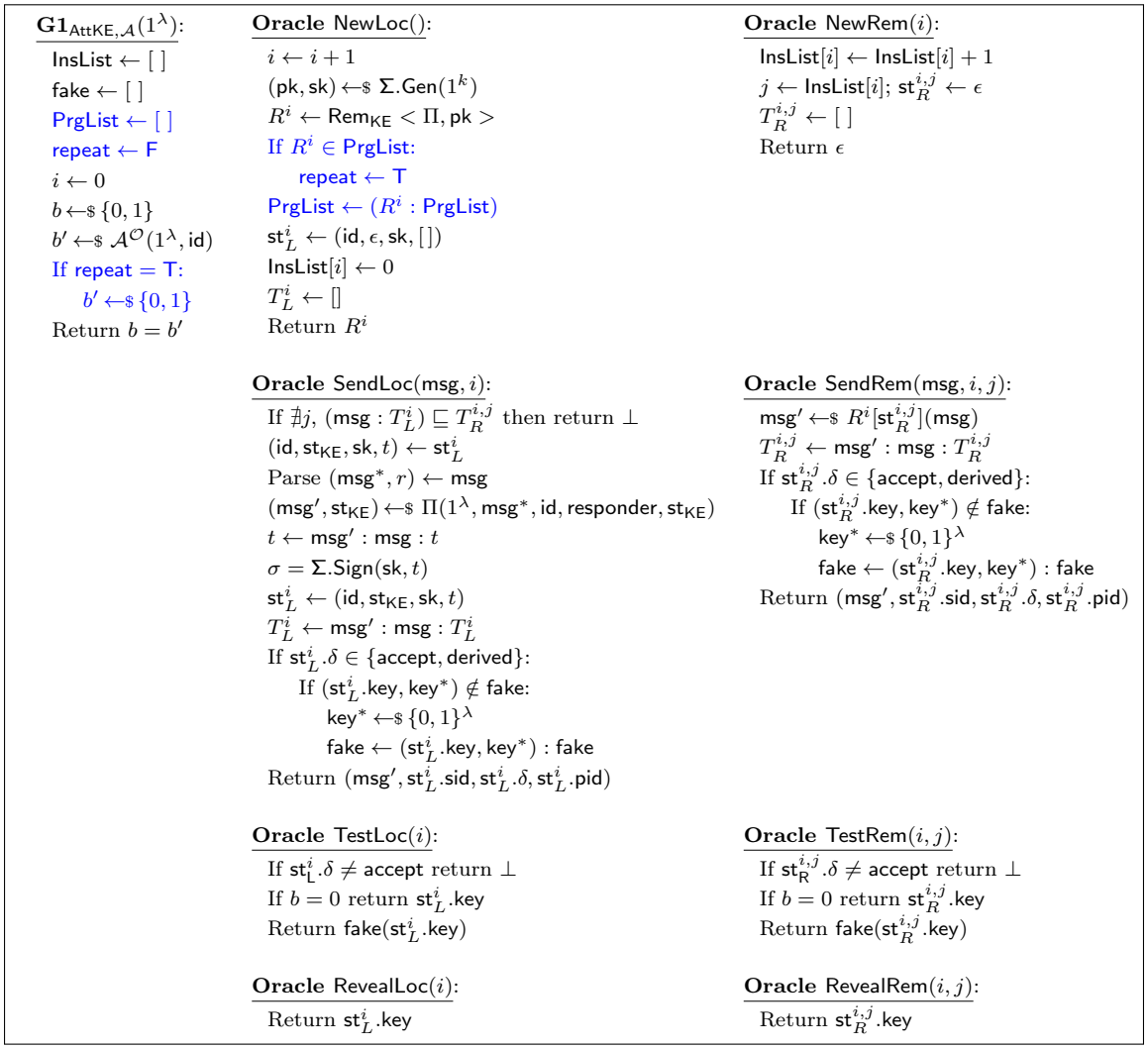


Figure 6.5: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

$$\Pr[\,\mathsf{forge}\,] \;\leq\; \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{C}}(\lambda) * \mathsf{q}\,.$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{C}$ has to try and guess which session will have a duplicate key. As such, it samples uniformly from $[1..\mathsf{q}]$ a session s and replaces the public key generated by $\Sigma.\mathsf{Gen}$ in NewLocal for instance $i = s$ with the public key $\mathsf{pk}_{\mathsf{UF}}$ provided by $\mathsf{UF}^{\Sigma,\mathcal{C}}$. Every time instance s has to produce a signature in $\mathsf{SendLocal}(\mathsf{msg}, s)$, instead of $\Sigma.\mathsf{Sign}(\mathsf{sk}, t)$, $\mathcal{C}$ calls Oracle $\mathsf{Sign}(t)$. During the execution of $\mathsf{SendRemote}(\mathsf{msg}, i, j)$ that sets $\mathsf{forge} = \mathsf{T}$, $\mathcal{C}$ presents $((o : \mathsf{st}^{i,j}_R.t), \sigma)$ as a challenge for $\mathsf{UF}^{\Sigma,\mathcal{C}}$. It remains to show that, when forge is set, $\mathcal{C}$ wins $\mathsf{UF}^{\Sigma,\mathcal{C}}$ with probability $1/\mathsf{q}$.

When $\mathsf{forge} = \mathsf{T}$ it is inferred that, for some execution of $\mathsf{SendRemote}(\mathsf{msg}, i, j)$, $\mathsf{st}^{i,j}_R.\delta \neq$

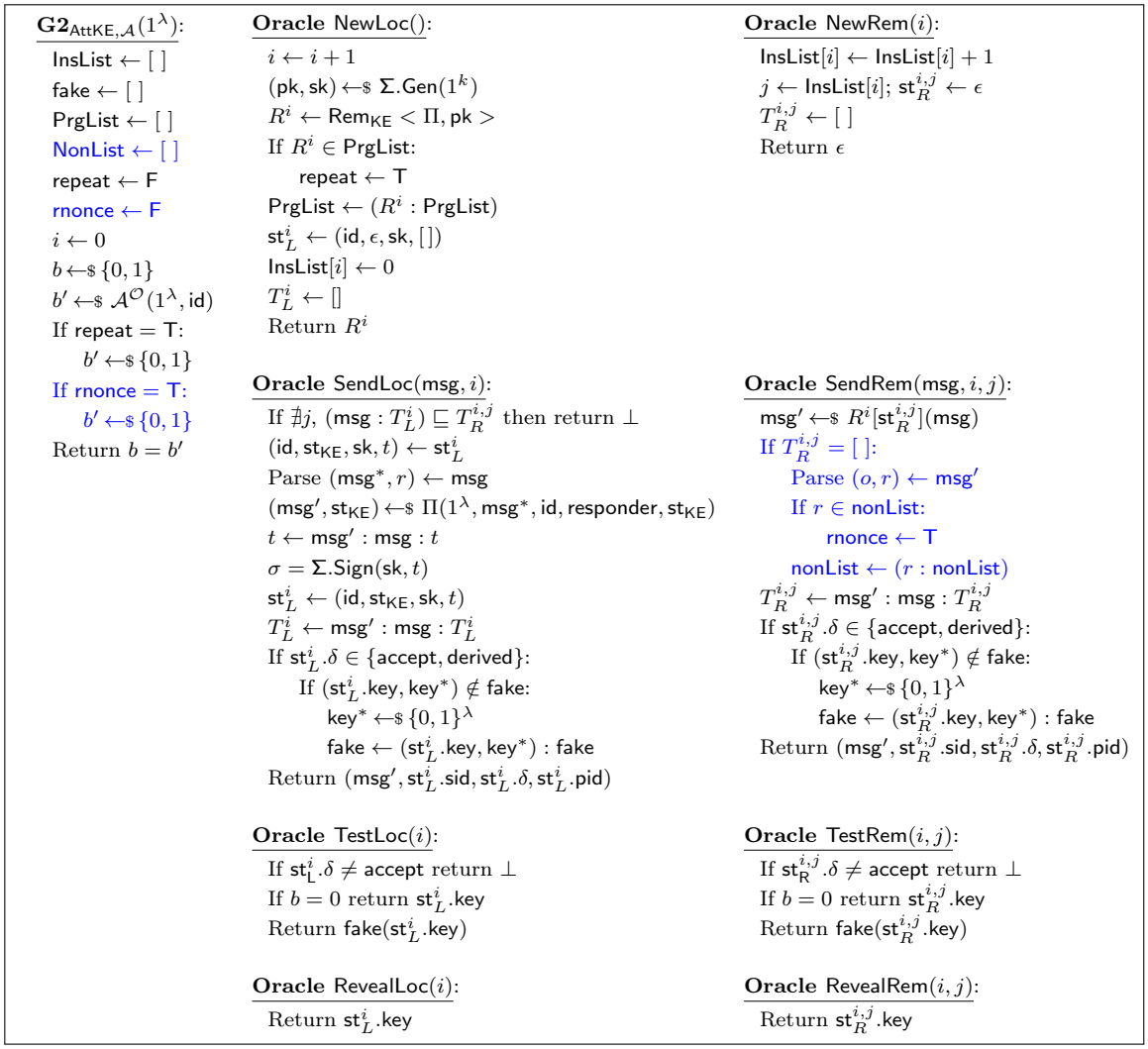


Figure 6.6: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

reject $\wedge\, (o : \mathsf{st}^{i,j}_R) \notin$ sigList. From the construction of R, it is also known that $\mathsf{st}^{i,j}_R.\delta \neq$ reject implies that the provided $((o : \mathsf{st}^{i,j}_R.t), \sigma)$ is a valid message/signature pair for session i. If $i = s$, this is also a valid message/signature pair for $\mathsf{UF}^{\Sigma,\mathcal{C}}$. Now observe that $(o : \mathsf{st}^{s,j}_R.t) \notin$ sigList gives assurance that $(o : \mathsf{st}^{s,j}_R.t)$ was not queried to oracle Sign: rnonce establishes that every nonce is unique, and every execution of SendLocal adds the nonce r to the signed message, so every call to oracle Sign is unique. From

$$\text{If } \nexists j,\ (\mathsf{msg} : T^s_L) \sqsubseteq T^{s,j}_R \text{ then return } \perp$$

and repeat, it follows that if oracle Sign was called for some $(m' : \mathsf{st}^{s,j}_R.t)$, then SendLocal of session s would be responding to the sequence of messages exchanged with the unique instance j, matching $T^{s,j}_R$ that coincides (modulo signatures) with $\mathsf{st}^{s,j}_R.t$. However, by the construction of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, that would imply sigList ← $(\mathsf{st}^{s,j}_R.t$: sigList), and it is already

known that $\mathsf{st}_R^{s,j}.t \notin \mathsf{sigList}$. One can therefore conclude that $(\mathsf{st}_R^{i,j}.t, \sigma)$ is a winning output for game $\mathsf{UF}^{\Sigma,\mathcal{C}}$ if $i = s$. That probability is $1/\mathsf{q}$.

$\mathbf{G3}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
$\mathsf{InsList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{NonList} \leftarrow [\,]$
$\mathsf{sigList} \leftarrow [\,]$
$\mathsf{repeat} \leftarrow \mathsf{F}$
$\mathsf{rnonce} \leftarrow \mathsf{F}$
$\mathsf{forge} \leftarrow \mathsf{F}$
$i \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(1^\lambda, \mathsf{id})$
If $\mathsf{repeat} = \mathsf{T}$:
  $b' \leftarrow\$ \{0,1\}$
If $\mathsf{rnonce} = \mathsf{T}$:
  $b' \leftarrow\$ \{0,1\}$
If $\mathsf{forge} = \mathsf{T}$:
  $b' \leftarrow\$ \{0,1\}$
Return $b = b'$

Oracle NewLoc():
$i \leftarrow i + 1$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\$ \Sigma.\mathsf{Gen}(1^k)$
$R^i \leftarrow \mathsf{Rem}_{\mathsf{KE}} < \Pi, \mathsf{pk} >$
If $R^i \in \mathsf{PrgList}$:
  $\mathsf{repeat} \leftarrow \mathsf{T}$
$\mathsf{PrgList} \leftarrow (R^i : \mathsf{PrgList})$
$\mathsf{st}_L^i \leftarrow (\mathsf{id}, \epsilon, \mathsf{sk}, [\,])$
$\mathsf{InsList}[i] \leftarrow 0$
$T_L^i \leftarrow []$
Return R^i

Oracle SendLoc(msg, i):
If $\nexists j$, $(\mathsf{msg} : T_L^i) \sqsubseteq T_R^{i,j}$ then return $\perp$
$(\mathsf{id}, \mathsf{st}_{\mathsf{KE}}, \mathsf{sk}, t) \leftarrow \mathsf{st}_L^i$
Parse $(\mathsf{msg}^*, r) \leftarrow \mathsf{msg}$
$(\mathsf{msg}', \mathsf{st}_{\mathsf{KE}}) \leftarrow\$ \Pi(1^\lambda, \mathsf{msg}^*, \mathsf{id}, \mathsf{responder}, \mathsf{st}_{\mathsf{KE}})$
$t \leftarrow \mathsf{msg}' : \mathsf{msg} : t$
$\sigma = \Sigma.\mathsf{Sign}(\mathsf{sk}, t)$
$\mathsf{sigList} \leftarrow t : \mathsf{sigList}$
$\mathsf{st}_L^i \leftarrow (\mathsf{id}, \mathsf{st}_{\mathsf{KE}}, \mathsf{sk}, t)$
$T_L^i \leftarrow \mathsf{msg}' : \mathsf{msg} : T_L^i$
If $\mathsf{st}_L^i.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{st}_L^i.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{msg}', \mathsf{st}_L^i.\mathsf{sid}, \mathsf{st}_L^i.\delta, \mathsf{st}_L^i.\mathsf{pid})$

Oracle TestLoc(i):
If $\mathsf{st}_{\mathsf{L}}^i.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathsf{st}_L^i.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}_L^i.\mathsf{key})$

Oracle RevealLoc(i):
Return $\mathsf{st}_L^i.\mathsf{key}$

Oracle NewRem(i):
$\mathsf{InsList}[i] \leftarrow \mathsf{InsList}[i] + 1$
$j \leftarrow \mathsf{InsList}[i]$; $\mathsf{st}_R^{i,j} \leftarrow \epsilon$
$T_R^{i,j} \leftarrow [\,]$
Return ϵ

Oracle SendRem(msg, i, j):
$\mathsf{msg}' \leftarrow\$ R^i[\mathsf{st}_R^{i,j}](\mathsf{msg})$
If $T_R^{i,j} = [\,]$:
  Parse $(o, r) \leftarrow \mathsf{msg}'$
  If $r \in \mathsf{nonList}$:
    $\mathsf{rnonce} \leftarrow \mathsf{T}$
  $\mathsf{nonList} \leftarrow (r : \mathsf{nonList})$
Else:
  Parse $(o, \sigma) \leftarrow \mathsf{msg}$
  If $\mathsf{st}_R^{i,j}.\delta \neq \mathsf{reject} \ \wedge (o : \mathsf{st}_R^{i,j}.t) \notin \mathsf{sigList}$
    $\mathsf{forge} \leftarrow \mathsf{T}$
$T_R^{i,j} \leftarrow \mathsf{msg}' : \mathsf{msg} : T_R^{i,j}$
If $\mathsf{st}_R^{i,j}.\delta \in \{\mathsf{accept}, \mathsf{derived}\}$:
  If $(\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) \notin \mathsf{fake}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{fake} \leftarrow (\mathsf{st}_R^{i,j}.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return $(\mathsf{msg}', \mathsf{st}_R^{i,j}.\mathsf{sid}, \mathsf{st}_R^{i,j}.\delta, \mathsf{st}_R^{i,j}.\mathsf{pid})$

Oracle TestRem(i, j):
If $\mathsf{st}_{\mathsf{R}}^{i,j}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ return $\mathsf{st}_R^{i,j}.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}_R^{i,j}.\mathsf{key})$

Oracle RevealRem(i, j):
Return $\mathsf{st}_R^{i,j}.\mathsf{key}$

Figure 6.7: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

Finally, in game $\mathsf{G4}^{\Pi,\mathcal{A}}$, AttKE is no longer required, and instead a passive adversary and its corresponding oracles $\{\mathsf{Execute}(i,j), \mathsf{Reveal}(i,s), \mathsf{Test}(i,s)\}$ is in place. The intuition is that, at this fifth game, all calls to the compiled AttKE protocol can be perfectly simulated using the original protocol Π. More formally, it is required to show that

$$\Pr[\,\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] = \Pr[\,\mathsf{G4}^{\Pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,].$$

Whenever $\mathsf{G4}^{\Pi,\mathcal{A}}$ is required to respond to SendLocal or SendRemote, instead of executing Π

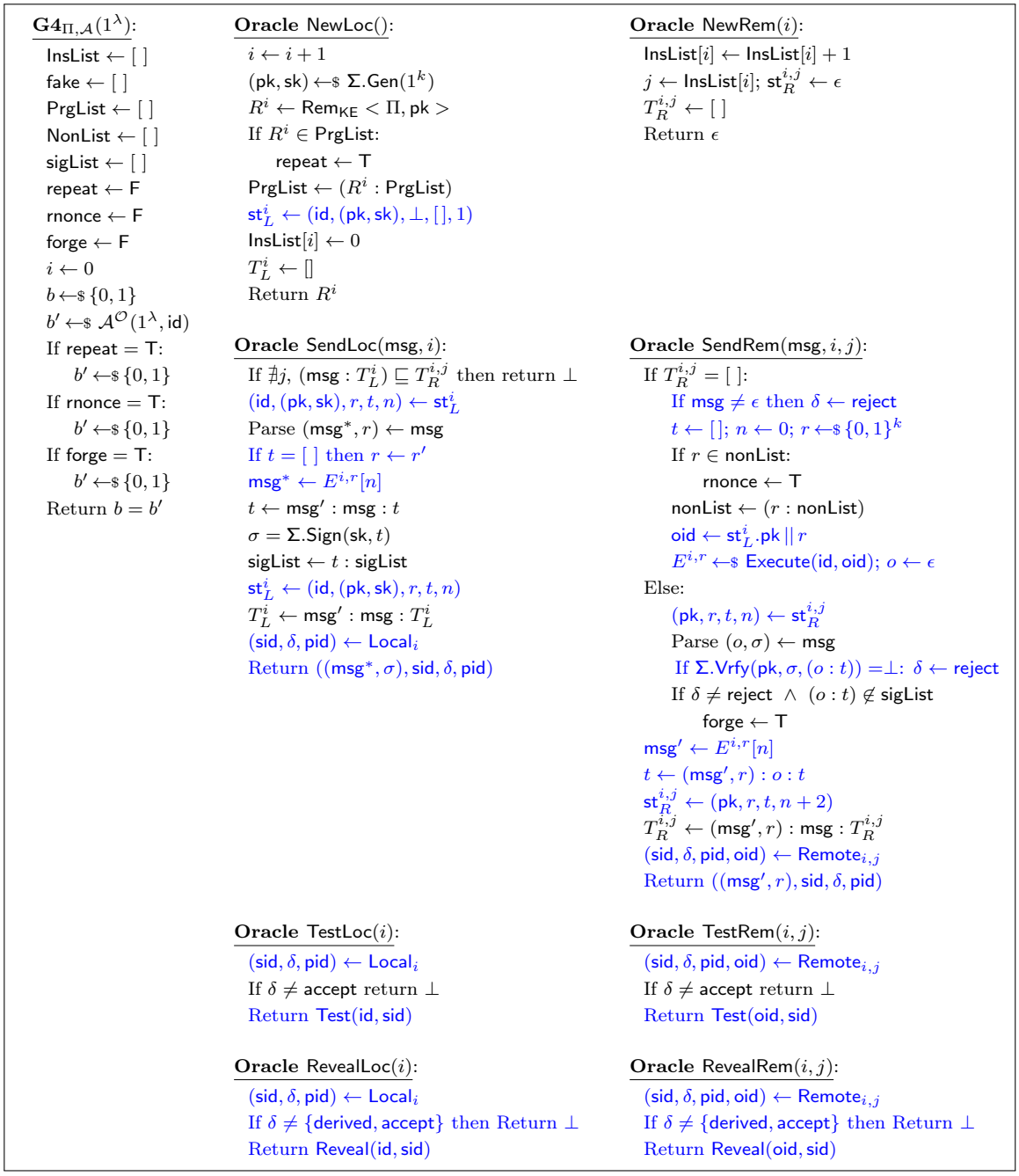


Figure 6.8: Game defining the execution environment for the security analysis of an AttKE scheme for an arbitrary local identity id. $\mathcal{O}$ denotes all oracles associated with the game.

and R (respectively), it will follow a unique Execute(id, oid) that is associated with sessions i and nonce r. Additionally, $\mathsf{G4}^{\Pi,\mathcal{A}}$ tracks when any execution should be rejected, and responds to Reveal/Test with either $\perp$ or with the output of the same oracles from the passive adversary. To help with this, $\mathsf{G4}^{\Pi,\mathcal{A}}$ stores a list of Execute transcripts $E^{i,r}$. Messages exchanged from such transcripts will be tracked locally and remotely with a counter n. From the information

in $(\mathsf{st}_L^i, \mathsf{st}_R^{i,j})$ and transcripts $E^{i,r}$ it is realistic to assume that it is possible to infer $(\mathsf{sid}, \delta, \mathsf{pid})$ and $(\mathsf{sid}, \delta, \mathsf{pid}, \mathsf{oid})$ via Local_i and $\mathsf{Remote}_{i,j}$, respectively.

The changes in NewLocal merely reflect additional information stored in existing structures. In SendLocal, $E^{i,r}[n]$ is used instead of Π to provide the next message. In SendRemote, instead of running R, the specification in Figure 6.3 is followed on the oracle itself, with the exception of calling $E^{i,j}[n]$ instead of using Π to provide the next message. In Reveal/Test, either $\perp$ is returned or the result of another oracle Reveal/Test is given, instead of directly providing a key. In these scenarios, handling of fake keys is delegated to Test queries of the passive adversary. As such, to validate these games as equivalent, it must now be shown that all calls to $E^{i,r}$ correspond to the Π replaced, and that Reveal/Test are responding similarly to $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$.

Observe that, by the construction of $E^{i,r} \leftarrow_\$ \mathsf{Execute}(\mathsf{id}, \mathsf{oid})$,

$$E^{i,r} = [o_1 \leftarrow_\$ \Pi(1^\lambda, \epsilon, \mathsf{oid}, \mathsf{initiator}, st_r), o_2 \leftarrow_\$ \Pi(1^\lambda, o_1, \mathsf{id}, \mathsf{responder}, st_i), \ldots]$$

the transcript $E^{i,r}$ contains a list of specific executions of $\Pi(1^\lambda, \mathsf{msg}, i, \rho, st)$. It must be shown that every $E^{i,j}[n]$ matches to the output of $\Pi(1^\lambda, \mathsf{msg}, i, \rho, st)$ executed in $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$. First note that, from repeat, it is inferred that every i is associated with a different R. This means that every $\mathsf{SendRemote}(\mathsf{msg}, i, j)$ will be associated with a unique st_L^i. Many executions of the same session i may occur, so the nonce associated cannot be determined at NewLocal (hence $\mathsf{st}_L^i \leftarrow (\mathsf{id}, (\mathsf{pk}, \mathsf{sk}), \perp, [\,], 1)$), but it is also known that every first message of AttKE will fix a unique r, so $E^{i,r}$ can be established in the first non-reject call of $\mathsf{SendLocal}(\mathsf{msg}, i)$, and a 1-to-1 relation with instance j is given by

$$\text{If } \nexists j,\ (m : T_L^i) \sqsubseteq T_R^{i,j} \text{ then return } \perp$$

and forge. Let $\mathsf{SendLocal}_k$ and $\mathsf{SendRemote}_k$ be the k-th execution of these Oracles.

- $\mathsf{SendRemote}_1(\mathsf{msg}, i, j)$: If $\delta \neq \mathsf{reject}$ and $r \leftarrow_\$ \{0,1\}^k$, we have that

  $$[\Pi(1^\lambda, \epsilon, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}), \ldots] \leftarrow_\$ \mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$$

  and thus $\Pi(1^\lambda, \epsilon, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}) = E^{i,r}[0] = \mathsf{msg}_1$; $\mathsf{st}_R^{i,j}.t = [(\mathsf{msg}_1, r), \epsilon]$; $\mathsf{st}_R^{i,j}.n = 2$.

- $\mathsf{SendLocal}_2(\mathsf{msg}, i)$: If $\mathsf{st}_L^i.\delta \neq \mathsf{reject}$, then it is implied that $((o, r) : \mathsf{st}_L^i.t)$ is in the prefix trace of a unique instance $\mathsf{SendRemote}_1$ (from (1)). Since $\mathsf{st}_L^i.t = [\,]$, msg was the first message produced by $\mathsf{SendRemote}_1$, one can set r as the nonce of the execution, retrieving the second message from $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$ in $E^{i,r}$, the unique transcript between session i and the remote j that produced r. Therefore, $\Pi(1^\lambda, \mathsf{msg}', \mathsf{id}, \mathsf{responder}, \mathsf{st}) = E^{i,r}[1] = \mathsf{msg}_2$. $\mathsf{st}_L^i.t = [\mathsf{msg}_2, (\mathsf{msg}_1, r'), \epsilon]$; $\mathsf{st}_L^i.n \leftarrow 3$.

- $\mathsf{SendRemote}_n(\mathsf{msg}, i, j)$: If $\delta \neq \mathsf{reject}$, Parse $(o, \sigma) \leftarrow \mathsf{msg}$; $\Sigma.\mathsf{Vrfy}(\mathsf{pk}, \sigma, (o : \mathsf{st}_R^{i,j}.t)) \neq \perp$ and $(o : \mathsf{st}_R^{i,j}.t) \in \mathsf{sigList}$ means that msg was the $(n-1)$-th message produced by

instance $\mathsf{SendLocal}_{n-1}(\mathsf{msg}_{n-2}, i)$ with $\mathsf{st}_L^i.t = [o, \ldots, (\mathsf{msg}_1, r), \epsilon]$, following $(o : \mathsf{st}_R^{i,j}.t)$ (from forge). $(o : \mathsf{st}_R^{i,j}.t)$ was constructed with the first $n-1$ messages in the transcript of $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$: $E^{i,r}$, so given that $E^{i,r}[n]$ provides the n-th message of $E^{i,r}$, then $\Pi(1^\lambda, o, \mathsf{pk}||r, \mathsf{initiator}, \mathsf{st}) = E^{i,r}[n]$. $\mathsf{st}_R^{i,j}.n = n + 2$.

- $\mathsf{SendLocal}_n(\mathsf{msg}, i)$: If $\mathsf{st}_L^i.\delta \neq \mathsf{reject}$, then msg was the $(n-1)$-th message produced by some instance $\mathsf{SendRemote}_{n-1}$ with $([\mathsf{msg}, \ldots, \mathsf{msg}_2, (\mathsf{msg}_1, r), \epsilon])$ in its trace $\mathsf{st}_R^{i,j}.t$ (from (1)). $\mathsf{st}_R^{i,j}.t$ was also constructed with the first $n-1$ messages in the transcript of $\mathsf{Execute}(\mathsf{id}, \mathsf{pk}||r)$: $E^{i,r}$, so given that $E^{i,r}[n]$ provides the n-th message of $E^{i,r}$, one infers that $\Pi(1^\lambda, m', \mathsf{id}, \mathsf{responder}, \mathsf{st}) = E^{i,r}[n]$. $\mathsf{st}_L^i.n = n + 2$.

Regarding Reveal and Test queries, observe that either the instances of i or j have $\delta \neq \{\mathsf{derived}, \mathsf{accept}\}$ or $\delta \neq \mathsf{accept}$, respectively, which given the previous is the same in both scenarios, or $\mathcal{A}$ gets the response from the Oracle of the passive adversary regarding the unique $\mathsf{Execute}(\mathsf{id}, \mathsf{oid})$ associated with sessions i and j, and the output is also the same in both scenarios.

At this point, the behaviour of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ to an adversary $\mathcal{A}$ is indistinguishable to the one provided in $\mathsf{G4}^{\Pi,\mathcal{A}}$ interacting with a passive adversary for the original protocol. This provides the correctness and security guarantees of the passive protocol Π modulo any attacks that would also be possible in the passive scenario. Now observe that, given that rnonce assures unique values for oid, and given that $\mathsf{Local}_{\mathsf{KE}}$ and $\mathsf{Remote}_{\mathsf{KE}}$ have $\rho = \mathsf{responder}$ and $\rho = \mathsf{initiator}$, respectively, it is trivial to infer that this implies the correctness and security of AttKE.

To conclude,

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}} &= \Pr[\mathsf{G}_0^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)] - \Pr[\mathsf{G}_4^{\Pi,\mathcal{A}}(1^\lambda)] \\
&= (\sum_{i=0}^{3} \Pr[\mathsf{G}_i^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)] - \Pr[\mathsf{G}_{i+1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)]) + (\Pr[\mathsf{G}_3^{\mathsf{AttKE},\mathcal{A}}(1^\lambda)] - \Pr[\mathsf{G}_4^{\Pi,\mathcal{A}}(1^\lambda)]) \\
&\leq \Pr[\mathsf{repeat}] + \Pr[\mathsf{rnonce}] + \Pr[\mathsf{forge}] + \mathsf{Adv}^{\mathsf{Att}}_{\Pi,\mathcal{A}}(\lambda) \\
&\leq \frac{3 * \mathsf{Adv}^{\mathsf{UF}}_{\Sigma,\mathcal{D}}(\lambda) * \mathsf{q}}{2} + \frac{\mathsf{q}_R^2}{2^k} + \mathsf{Adv}^{\mathsf{Att}}_{\Pi,\mathcal{A}}(\lambda)
\end{aligned}$$

and Theorem 3 follows.

□

6.2 Utility

Finally, for constructing full-fledged authenticated and private remote attested computation applications, several utility theorems will now be presented, describing precisely the guarantees one obtains when combining an attested computation protocols and labelled attested

computation protocols with AttKE, in various settings. Intuitively, these theorems will state that attestation allows for authentication and secrecy assurances offered by AttKE to be retained when is used to establish session keys with remote IEEs, in the presence of fully active adversaries that control the remote machine, and when the key exchange is composed with arbitrary programs.

One-to-one AttKE utility

The first setting considered is the one presented in Section 5.2. Figure 6.9 shows an idealised game where an adversary must distinguish between two remote machines where an AttKE scheme is executed in combination with an AC scheme. Machine $\mathcal{M}$ is any standard remote machine that is supported by the attested computation protocol, whereas $\mathcal{M}'$ represents a modification of $\mathcal{M}$ where one can tweak the operation of $\mathsf{Rem}_{\mathsf{KE}}$ programs. The differences of $\mathcal{M}'$ with respect to $\mathcal{M}$ are concentrated on the Run interface, which now operates as follows:

- It takes as additional parameters a list fake of pairs of keys and Boolean flag tweak that, when activated, identifies a process that is running an instance of $\mathsf{Rem}_{\mathsf{KE}}$ composed with some program Q. This flag triggers the following modifications with respect to the operations of $\mathcal{M}$.
- When it detects that $\mathsf{Rem}_{\mathsf{KE}}$ has transitioned into $\mathsf{derived}$ or accept state, it will check if the derived key exists in list fake. If not, it generates a new random key^*, and $(\mathsf{key}, \mathsf{key}^*)$ is added to the list.
- When it detects that program Q is set to start executing, rather than using the key as an input to ϕ, it uses $\mathsf{fake}(\mathsf{key})$ instead.

The environment presented to the adversary models a standard attested computation interaction, where it is given total control over the remote machine using oracles Load and Run (these oracles will either give access to $\mathcal{M}$ or to $\mathcal{M}'$, depending on a secret bit b generated in the beginning of the game). The adversary is also able to obtain challenge remote programs using a $\mathsf{NewSession}(Q)$ oracle that uses the attested computation scheme to compile $\mathsf{Rem}_{\mathsf{KE}}$ composed with arbitrary program Q of its choice under a mapping function ϕ_{key} that reveals the relevant parts of the key exchange state (namely the secret key key, the party identifiers oid and pid, the state δ and the session identifier sid). Observe that such arbitrary programs can leak all of the information revealed by ϕ_{key} to the attacker. If the adversary chooses to Load a challenge program, and if $\mathcal{M}'$ is being used in the game, then it will be tweaked as described above. Whenever $\mathsf{NewSession}(Q)$ is called, the environment creates a new local session i that the adversary can interact with using a $\mathsf{Send}(i, \mathsf{msg})$ oracle. The Send oracle uses the Verify algorithm of the attested computation scheme to validate attested outputs and, if they are accepted, feeds them to the $\mathsf{Loc}_{\mathsf{KE}}$ instance (and also ensures that list fake is

updated). Finally, the adversary can explicitly choose to be tested (as opposed to the implicit testing it may trigger using arbitrary programs Q) by calling Test on a local instance. This oracle will either return the true key, if $b = 0$, or the associated random key that is kept in the fake list. As before, the winning event guess is established to occur when $b = b'$ in the end of the game.

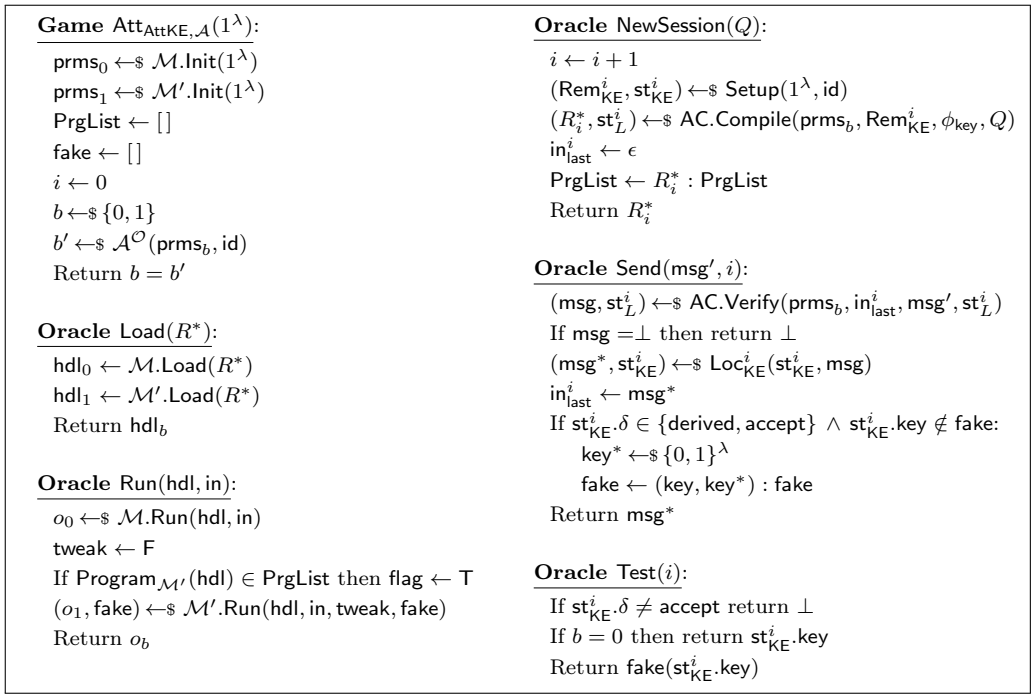


Figure 6.9: Game defining the utility of an AttKE scheme when used in the context of attested computation.

Theorem 4 (One-to-one AttKE utility)**.** *If* AttKE *is correct and secure and the AC protocol is correct, secure and ensures minimum leakage, then for all ppt adversaries in the utility experiment: 1. the probability that the adversary violates AttKE two-sided entity authentication is negligible; and the key secrecy advantage* $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

This proof begins by bounding the possibility for the occurrence of a bad event ($\mathsf{G0}^{\mathsf{AC},\mathcal{A}}$ to $\mathsf{G1}^{\mathsf{AC},\mathcal{A}}$). Then, the machine execution will be replaced by the indistinguishable behaviour of the simulator of minimum leakage game in Figure 5.4 for $\mathsf{G2}^{\mathsf{AC},\mathcal{A}}$. Finally, one argues that, given these circumstances, this scenario is the same as one of $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ using the oracles of key exchange for attested computation in Figure 6.2 modulo any advantage the adversary may gain from the AttKE scheme. The proof consists in a sequence of four games presented in figures 6.10 to 6.13. The first game is simply the utility game in Figure 6.9.

In the second game $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$, the adversary loses whenever a forgeAC event occurs. Intuitively, this event corresponds to the adversary producing an output that is successfully validated, but was not computed using Run, and hence constitutes a forgery with respect to

$\mathbf{G0}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms}_0 \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$\ \{0,1\}$
$b' \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b, \mathsf{id})$
Return $b = b'$

Oracle $\mathsf{Load}(R^*)$:
If $b = 0$
  Return $\mathcal{M}.\mathsf{Load}(R^*)$
Return $\mathcal{M}'.\mathsf{Load}(R^*)$

Oracle $\mathsf{NewSession}(Q)$:
$i \leftarrow i + 1$
$(\mathsf{Rem}^i_{\mathsf{KE}}, \mathsf{st}^i_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$(R^*_i, \mathsf{st}^i_L) \leftarrow\$\ \mathsf{AC.Compile}(\mathsf{prms}_b, \mathsf{Rem}^i_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{PrgList} \leftarrow ((R^*_i, \mathsf{Rem}^i_{\mathsf{KE}}, Q) : \mathsf{PrgList})$
Return R^*_i

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:
$\mathsf{flag} \leftarrow \mathsf{F}$
If $\mathsf{Program}_{\mathcal{M}'}(\mathsf{hdl}) \in \mathsf{PrgList}$ then $\mathsf{flag} \leftarrow \mathsf{T}$
If $b = 0$ return $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
$(o, \mathsf{fake}) \leftarrow\$\ \mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{flag}, \mathsf{fake})$
Return o

Oracle $\mathsf{Send}(\mathsf{msg}', i)$:
$(\mathsf{msg}, \mathsf{st}^i_L) \leftarrow\$\ \mathsf{AC.Verify}(\mathsf{prms}_b, \mathsf{in}^i_{\mathsf{last}}, \mathsf{msg}', \mathsf{st}^i_L)$
$(\mathsf{msg}^*, \mathsf{st}^i_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Loc}^i_{\mathsf{KE}}(\mathsf{st}^i_{\mathsf{KE}}, \mathsf{msg})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{msg}^*$
If $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
  $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return msg^*

Oracle $\mathsf{Test}(i)$:
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ then return $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key})$

Figure 6.10: Game defining the utility of an AttKE scheme when used in the context of attested computation.

$\mathbf{G1}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda)$:
$\mathsf{prms}_0 \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$\mathsf{forgeAC} \leftarrow \mathsf{F}$
$i \leftarrow 0$
$b \leftarrow\$\ \{0,1\}$
$b' \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b, \mathsf{id})$
If $\mathsf{forgeAC} = \mathsf{T}$:
  $b' \leftarrow \{0,1\}$
Return $b = b'$

Oracle $\mathsf{Load}(R^*)$:
If $b = 0$
  Return $\mathcal{M}.\mathsf{Load}(R^*)$
Return $\mathcal{M}'.\mathsf{Load}(R^*)$

Oracle $\mathsf{NewSession}(Q)$:
$i \leftarrow i + 1$
$(\mathsf{Rem}^i_{\mathsf{KE}}, \mathsf{st}^i_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$(R^*_i, \mathsf{st}^i_L) \leftarrow\$\ \mathsf{AC.Compile}(\mathsf{prms}_b, \mathsf{Rem}^i_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{PrgList} \leftarrow ((R^*_i, \mathsf{Rem}^i_{\mathsf{KE}}, Q) : \mathsf{PrgList})$
$T^i_L = [\,]$
Return R^*_i

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:
$\mathsf{flag} \leftarrow \mathsf{F}$
If $\mathsf{Program}_{\mathcal{M}'}(\mathsf{hdl}) \in \mathsf{PrgList}$ then $\mathsf{flag} \leftarrow \mathsf{T}$
If $b = 0$: $o \leftarrow \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
Else: $(o, \mathsf{fake}) \leftarrow\$\ \mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{flag}, \mathsf{fake})$
Return o

Oracle $\mathsf{Send}(\mathsf{msg}', i)$:
$(\mathsf{msg}, \mathsf{st}^i_L) \leftarrow\$\ \mathsf{AC.Verify}(\mathsf{prms}_b, \mathsf{in}^i_{\mathsf{last}}, \mathsf{msg}', \mathsf{st}^i_L)$
If $b = 0$ then $\mathcal{M} \leftarrow \mathcal{M}$ else $\mathcal{M} \leftarrow \mathcal{M}'$
If $\mathsf{msg} \neq \perp \ \wedge\ \nexists \mathsf{hdl}$ s.t. $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R^*_i$
 $\mathsf{Rev}(\mathsf{msg}' : T^i_L) \sqsubseteq \mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl})$:
  $\mathsf{forgeAC} \leftarrow \mathsf{T}$
$(\mathsf{msg}^*, \mathsf{st}^i_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Loc}^i_{\mathsf{KE}}(\mathsf{st}^i_{\mathsf{KE}}, \mathsf{msg})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{msg}^*$; $T^i_L \leftarrow \mathsf{msg}^* : \mathsf{msg}' : T^i_L$
If $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
  $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return msg^*

Oracle $\mathsf{Test}(i)$:
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$ return $\perp$
If $b = 0$ then return $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key}$
Return $\mathsf{fake}(\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key})$

Figure 6.11: First hop of the utility proof.

AC. Let Rev reverse a list given as input[2]. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G0}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{forgeAC}\,].$$

Let q be the maximum number of calls to NewSession allowed, and N the number of messages exchanged in $\mathsf{Rem}^i_{\mathsf{KE}}$. The distance between these two games is upper bound, by constructing an adversary $\mathcal{B}$ against the security of AC such that

$$\Pr[\,\mathsf{forgeAC}\,] \;\leq\; \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AC},\mathcal{B}}(\lambda) * \mathsf{q} * \lceil N/2 \rceil\,.$$

[2]This is for handling a technicality, in which the trace of $\mathcal{M}$ and the constructed T^i_L are in reverse order.

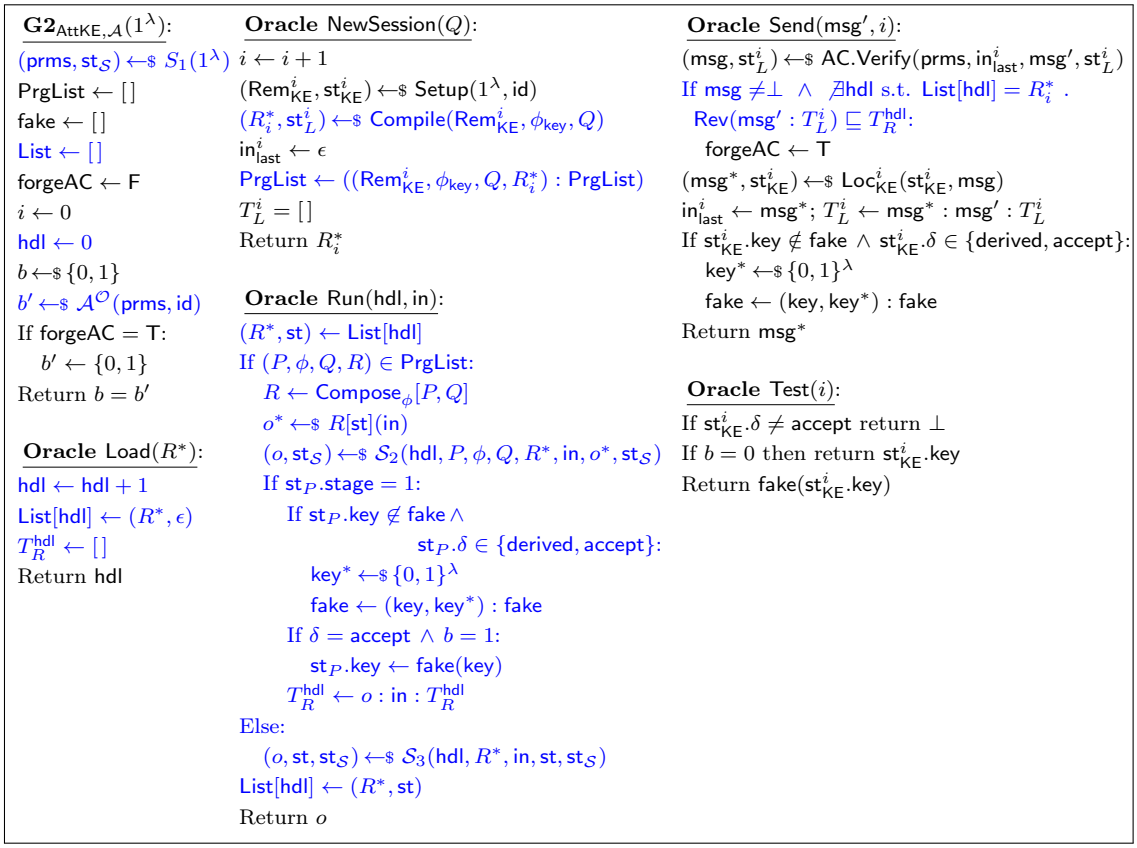


Figure 6.12: Second hop of the utility proof.

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ as follows: at the beginning of the game, $\mathcal{B}$ has to try and guess which session will have a forged message, and which message of the protocol it will be. As such, it samples uniformly from $[1..\mathsf{q}]$ a session s and from $[1..\lceil N/2 \rceil]$ a message k. During NewSession such that $i = s$, $\mathcal{B}_1$ will output $(\mathsf{Rem}^i_{\mathsf{KE}}, \phi_{\mathsf{key}}, Q, k, \bot)$ to $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, store the produced R^* and all hdl output by $\mathsf{Load}(R^*)$ from there on. Afterwards, all calls to $\mathcal{M}.\mathsf{Load}(R^*)$ and $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, m)$ or $\mathcal{M}'.\mathsf{Load}(R^*)$ and $\mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, m, \mathsf{flag}, \mathsf{fake})$ with the same calls on $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$. Whenever $\mathsf{Send}(\mathsf{msg}', i)$ is called, $i = s$ and $\mathsf{msg} \neq \bot$, let n be the number of messages sent for session s:

- If $n = 0$, $\mathcal{B}_2$ outputs $(\bot, \mathsf{msg}', \mathsf{msg}^*)$.
- If $n > 0 \wedge n < k$, $\mathcal{B}_2$ outputs $(\mathsf{st}_{\mathcal{B}}, \mathsf{msg}', \mathsf{msg}^*)$

It remains to show that, when forgeAC is set, $\mathcal{B}$ wins $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$ with probability $1/(\mathsf{q} * \lceil N/2 \rceil)$. When the game ends and $\mathsf{forgeAC} = \mathsf{T}$,

$$\mathsf{msg} \neq \bot \ \wedge \ \nexists \mathsf{hdl} \text{ s.t. } \mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R^*_i \ . \ (\mathsf{msg}' : T^i_L) \sqsubseteq \mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl})$$

From the construction of Send, $\mathsf{msg} \neq \bot$ means that this message has been successfully validated by AC.Verify. Furthermore, this matches the verifications in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, considering

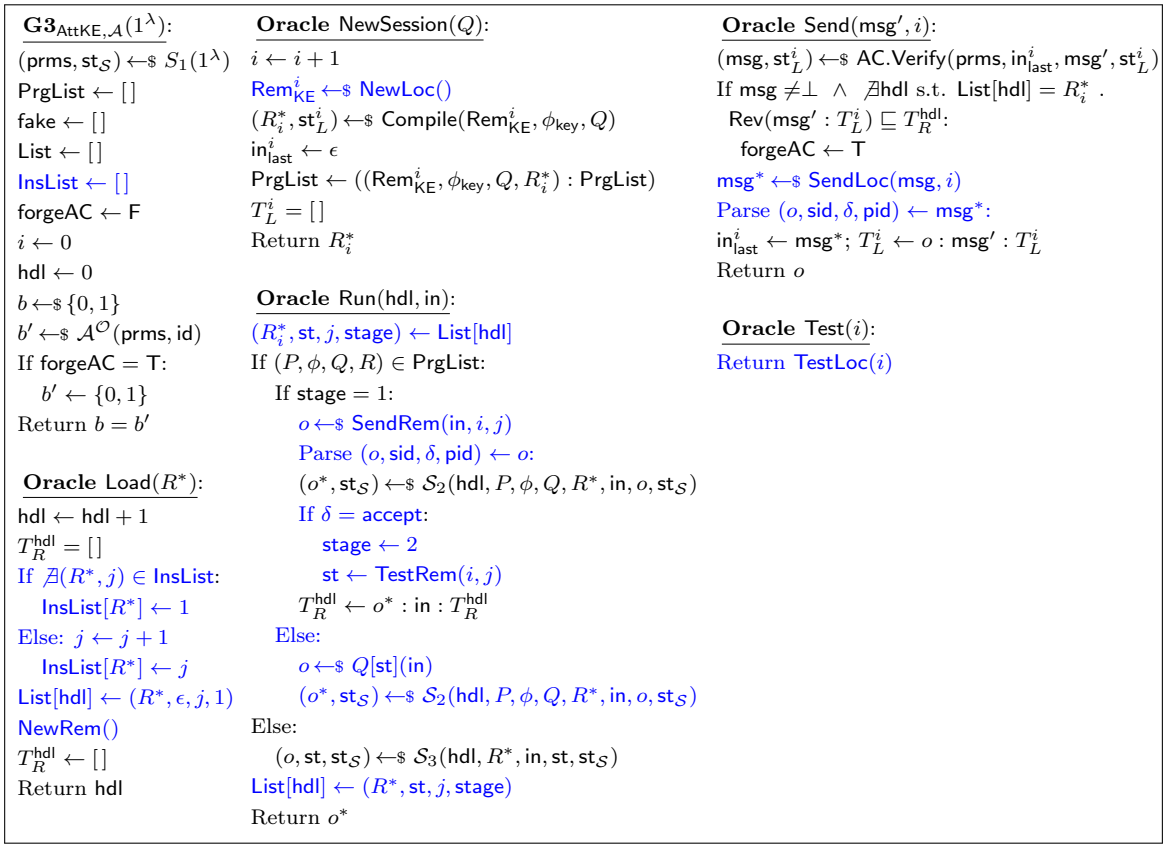


Figure 6.13: Third hop of the utility proof.

that $\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{msg}^*$ in Send, so it is known that all verifications that succeed in $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ also do so in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$. If forgeAC is set upon receiving message k, the final check is reached. All calls that produce handles such that $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = R_i^*$ are also performed in $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$, so it remains to show that $\mathsf{Rev}(\mathsf{msg}' : T_L^i)$ matches T.

It is known that $T \leftarrow \mathsf{ATrace}_{R[\mathsf{st};\mathsf{Coins}_\mathcal{M}(\mathsf{hdl}^*)]}(i_1, \ldots, i_n)$. From the construction of Send and the behaviour of the adversary, it can also be concluded that this is constructed with the outputs given by $\mathcal{B}_2$. It will be shown, inductively, why $\mathsf{Rev}(\mathsf{msg}' : T_L^i)$ matches the k outputs of $\mathcal{B}_2$.

- Initially, given that $(\perp, \mathsf{msg}'_0, \mathsf{msg}^*_0) \leftarrow \mathcal{B}_2$:

$$\mathsf{Rev}(\mathsf{msg}'_0 : T_L^i) = \mathsf{Rev}[\mathsf{msg}'_0, \perp] = [\perp, \mathsf{msg}'_0]$$

- For all subsequent messages up to k, $(\mathsf{msg}^*_{n-1}, \mathsf{msg}'_n, \mathsf{msg}^*_n) \leftarrow \mathcal{B}_2$, so

$$\mathsf{Rev}(\mathsf{msg}' : T_L^i) = \mathsf{Rev}[\mathsf{msg}'_n, \mathsf{msg}^*_{n-1}, \ldots, \mathsf{msg}'_0, \perp] = [\perp, \mathsf{msg}'_0, \ldots, \mathsf{msg}^*_{n-1}, \mathsf{msg}'_n]$$

As such, when forgeAC is set on session s, and in the k-th message input on Send,

$$\nexists \mathsf{hdl} \text{ s.t. } \mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = R_i^* \ .\ (\mathsf{msg}' : T_L^i) \sqsubseteq \mathsf{ATrace}_\mathcal{M}(\mathsf{hdl})$$

which results in a winning output for $\mathsf{Att}^{\mathsf{AC},\mathcal{B}}$ with probability $1/(\mathsf{q} * \lceil N/2 \rceil)$.

In the third game $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, when the adversary loads and runs code, it is no longer interacting with machine $\mathcal{M}$, but rather with a simulator that is given the trace of a legitimate execution. Furthermore, $\mathcal{S}$ handles the different behaviour of $\mathcal{M}$ and $\mathcal{M}'$ following the description in Section 6.2. Intuitively, this difference corresponds to the indistinguishable scenario presented in the minimum leakage game of Figure 5.4. Given the presented differences, it follows that

$$\Pr[\,\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \\ \Pr[\,\mathsf{Leak\text{-}Real}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Leak\text{-}Ideal}^{\mathsf{AC},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,].$$

One must now argue that the difference between the games is bound by the adversary's advantage of breaking minimal leakage. In both possibilities for the bit b, the transformation in Load is exactly the same as the one in the minimum leakage game. Furthermore, T_R^{hdl} will always correspond to the $\mathsf{ATrace}_{\mathcal{M}}(\mathsf{hdl})$. First, consider $b = 0$. The behaviour of Run is exactly the same as the one in the minimum leakage game, modulo the generation of the fake key, which will not be taken into consideration since it is only used in Test when $b = 1$. Now consider $b = 1$. The behaviour of Run is exactly the same as the one in the minimum leakage game, modulo generating and setting the fake key. However, these additional operations match the described behaviour expected from $\mathcal{M}'$ for establishing the fake key.

As such, the advantage of the adversary in $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ with respect to $\mathsf{G1}^{\mathsf{AttKE},\mathcal{A}}$ is limited by its advantage of breaking the minimum leakage.

Finally, in game $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, the key exchange part of the code is no longer executed. Instead, the game makes use of AttKE and its corresponding oracles $\{\mathsf{NewRem}, \mathsf{NewLoc}, \mathsf{SendRem}, \mathsf{SendLoc}, \mathsf{TestRem}, \mathsf{TestLoc}\}$. The intuition is that, at this fourth game, calls to the attested part of the protocol can be provided using a key exchange for attested computation AttKE. More formally, it must be shown that

$$\Pr[\,\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}}\,.$$

Whenever $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ creates a local or a remote session (respectively, $\mathsf{NewSession}$ or Load), it will initialize accordingly on the AttKE protocol. On the remote side, a new $\mathsf{InsList}$ will also keep track of how many remote sessions are created (just like oracle NewRem in AttKE). On the local side, algorithm Setup will be replaced by a call to NewLoc that by its construction will return the same value as in $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$. As such, it must now be argued that, every time the game is required to produce either an output from the local/remote machine or a key, the response given by the corresponding oracle is only distinguishable by an adversary that breaks AttKE security.

- For $\mathsf{stage} = 1 \ \wedge \ \delta \neq \mathsf{accept}$: Locally, SendLoc replaces calling $\mathsf{Loc}^{i}_{\mathsf{KE}}$, which by the oracle behaviour implies the same result. Remotely, instead of producing output via $R[\mathsf{st}](\mathsf{in})$, oracle SendRem is now directly executed, which by construction holds and updates st in the same way as $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$.
- For $\mathsf{stage} = 1 \ \wedge \ \delta = \mathsf{accept}$: In $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$, SendLoc would replace the real key with a fake key according to decision bit b. In $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$, the same thing happens, but according to AttKE decision bit. Remotely, the behaviour of $\mathsf{st} \leftarrow \mathsf{TestRem}$ also differs in the same manner.
- For $\mathsf{stage} = 2$: $\mathsf{Compose}_{\phi}[P, Q]$ implies that the state st considered for Q only maintains the key. As such, from thereon, the only difference between $\mathsf{G2}^{\mathsf{AttKE},\mathcal{A}}$ and $\mathsf{G3}^{\mathsf{AttKE},\mathcal{A}}$ is the key depending on the decision bit taken into consideration.

In this final game, the adversary's decision is based on a bit that is unrelated to any information obtainable via the oracles. Let

$$\mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} = \Pr[\,\mathsf{Leak\text{-}Real}^{\mathsf{AC},\mathcal{A}}(1^{\lambda}) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Leak\text{-}Ideal}^{\mathsf{AC},\mathcal{A}}(1^{\lambda}) \Rightarrow \mathsf{T}\,]$$

Finally

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{Ut}}_{\mathsf{AttKE},\mathcal{A}} &= \Pr[\,\mathsf{G}_0^{\mathsf{AttKE},\mathcal{A}}(1^{\lambda})\,] - \Pr[\,\mathsf{G}_3^{\mathsf{AttKE},\mathcal{A}}(1^{\lambda})\,] \\
&= \sum_{i=0}^{3} \Pr[\,\mathsf{G}_i^{\mathsf{AttKE},\mathcal{A}}(1^{\lambda})\,] - \Pr[\,\mathsf{G}_{i+1}^{\mathsf{AttKE},\mathcal{A}}(1^{\lambda})] \\
&\leq \Pr[\mathsf{forgeAC}] + \mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} + \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}} \\
&\leq \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AC},\mathcal{B}}(\lambda) * \mathsf{q} * \lceil N/2 \rceil + \mathsf{Adv}^{\mathsf{Leak}}_{\mathsf{AC},\mathcal{A}} + \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{AttKE},\mathcal{A}}
\end{aligned}$$

and Theorem 4 follows.

□

One-to-many AttKE utility

Attested Computation can also be employed when several instances of attested key exchange running in parallel. The one-to-many utility theorem validates that, if the authentication and secrecy assurances offered by AttKE are retained when used to establish keys between a single client and a remote IEE, then this is also the case for multiple clients establishing keys with a remote IEE. This setting considers the presence of a fully active adversary, in control of the machine and allowed to select k honest participants (controlling the remaining $n - k$), as well as the possibility of composing the key exchanges with any arbitrary function F.

Figure 6.15 shows an idealized experiment similar to the approach taken for the previous Utility theorem. The adversary is challenged to distinguish between two remote machines,

where several instances of an AttKE scheme are being combined and executed with an AC scheme. Machine $\mathcal{M}$ represents the standard remote machine supported by the AC protocol, while machine $\mathcal{M}'$ is a modification of $\mathcal{M}$ in which the operation of the first k (honest) $\mathsf{Rem}_{\mathsf{KE}}$ programs is tweaked. The differences between these machines are mostly concentrated on the Run interface, which operates as follows:

- It takes, as additional parameters, a list of key pairs fake, a Boolean flag t and the number of honest participants k. The activation of this flag identifies an IEE running the selected parallel execution of $\mathsf{Rem}_{\mathsf{KE}}$ composed with program Q, and triggers the following modifications with respect to the operations in $\mathcal{M}$:
  - When it detects that any of the first k executions of $\mathsf{Rem}_{\mathsf{KE}}$ has transitioned into the derived or accept state, it records the derived key and checks if it exists in the provided list fake. If that is not the case, it generates a new random key^* and adds $(\mathsf{key}, \mathsf{key}^*)$ to the list.
  - When it detects that program Q is executing for the first time, rather than using all keys as input to ϕ_{key}, it replaces the first k as the respective $\mathsf{fake}(\mathsf{key})$ (from $k+1$ to n, the behaviour is similar to $\mathcal{M}$).

To capture the notion of parallel execution of several instances of AttKE in the AC setting, ParComp is introduced, described in Figure 6.14. This program construction takes n programs $\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}$ and produces a program that is built to expect every input to consist in an array of n inputs. It parses the received input, runs every instance $\mathsf{Rem}^i_{\mathsf{KE}}$ with the i-th input and constructs an output array with the outputs of the executions.

Program $\mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, ..., \mathsf{Rem}^n_{\mathsf{KE}} \rangle (\mathsf{msg}, \mathsf{st})$:

If $\mathsf{st} = \epsilon$:
  For $i \in [1..n]$: $\mathsf{st}[i] \leftarrow \epsilon$
For $i \in [1..n]$:
  $o_i \leftarrow\$\ \mathsf{Rem}^i_{\mathsf{KE}}(\mathsf{msg}[i], \mathsf{st}[i])$
$\mathsf{out} \leftarrow o_1, \ldots, o_n$
Return out

Figure 6.14: Details for running n parallel key exchange protocols.

The provided environment models the adversary power in a standard attested computation interactions, where it is given access to the oracles Load and Run, matching to $\mathcal{M}$ or $\mathcal{M}'$, depending on the sampled secret bit b. The adversary can also initialize challenge programs by running $\mathsf{NewSession}(\phi, Q)$, which initializes n executions, compiles the KE programs and composes them with program Q, having the set of exchanged keys mapped to its initial state via ϕ. Note that, additionally to the adversary having access to the machine, its power is extended to control $n-k$ participants, by providing him their internal states $\mathsf{st}^{i(k+1)}_{\mathsf{KE}}, \ldots, \mathsf{st}^{in}_{\mathsf{KE}}$. Upon calling NewSession, the environment creates new session i, with which the adversary

Game $\mathsf{Att}_{\mathsf{OtmAttKE},\mathcal{A}}(1^\lambda, \mathcal{I}, k)$:
$\mathsf{prms}_0 \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$\ \{0,1\}$
$b' \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b)$
Return $b = b'$

Oracle $\mathsf{Test}(i)$:
$\mathsf{Keys} \leftarrow [\,]$
For $j \in [1..k]$:
 If $\mathsf{st}^{ij}_{\mathsf{KE}}.\delta \neq \mathsf{accept}$ return $\perp$
 If $b = 0$ then
  $\mathsf{Keys} \leftarrow \mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key} : \mathsf{Keys}$
 Else
  $\mathsf{Keys} \leftarrow \mathsf{fake}(\mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key}) : \mathsf{Keys}$
Return Keys

Oracle $\mathsf{NewSession}(\phi, Q)$:
$i \leftarrow i + 1$
For $j \in [1..n]$:
 $(\mathsf{st}^{ij}_{\mathsf{KE}}, \mathsf{Rem}^j_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathcal{I}[j])$
 $\mathsf{in}^{ij}_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{C} := \mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle$
For $j \in [1..n]$:
 $(R^*_i, \mathsf{st}^{ij}_L) \leftarrow\$\ \mathsf{Compile}(\mathsf{prms}_b, \mathsf{C}, \phi, Q)$
$\mathsf{PrgList} \leftarrow R^*_i : \mathsf{PrgList}$
Return $(R^*_i, \mathsf{st}^{i(k+1)}_{\mathsf{KE}}, \ldots, \mathsf{st}^{in}_{\mathsf{KE}})$

Oracle $\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$:
$o_0 \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, \mathsf{in})$
$\mathsf{t} \leftarrow \mathsf{F}$
If $\mathsf{Program}_{\mathcal{M}'}(\mathsf{hdl}) \in \mathsf{PrgList}$:
 then $\mathsf{t} \leftarrow \mathsf{T}$
$(o_1, \mathsf{fake}) \leftarrow\$\ \mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, \mathsf{in}, \mathsf{t}, \mathsf{fake}, k)$
Return o_b

Oracle $\mathsf{Send}(\mathsf{msg}', i, j)$:
$(\mathsf{ins}, \mathsf{outs}^*) \leftarrow \mathsf{msg}'$
$(\mathsf{outs}, \mathsf{st}^{ij}_L) \leftarrow \mathsf{Verify}(\mathsf{prms}, \mathsf{ins}, \mathsf{msg}', \mathsf{st}^{ij}_L)$
If $\mathsf{outs} = \perp$ Return $\perp$
If $\mathsf{in}^{ij}_{\mathsf{last}} \neq \mathsf{ins}[j]$ Return $\perp$
$(\mathsf{msg}^*, \mathsf{st}^{ij}_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}^{ij}_{\mathsf{KE}}, \mathsf{outs}[j])$
$\mathsf{in}^{ij}_{\mathsf{last}} \leftarrow \mathsf{msg}^*$
If $\mathsf{st}^{ij}_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge$
 $\mathsf{st}^{ij}_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
 $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
 $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return msg^*

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl}_0 \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'.\mathsf{Load}(R^*)$
Return hdl_b

Figure 6.15: Game defining the one-to-many utility of a AttKE scheme when used in the context of attested computation. Compile and Verify refer to methods of the underlying AC scheme.

can interact with in behalf of participant j by using $\mathsf{Send}(\mathsf{msg}', i, j)$. The Send oracle mimics the behavior of a local participant by using the Verify algorithm of the AC scheme to validate if the output was correctly attested, and if the j-th input (the input provided by the local participant) is consistent with what was provided. If both of these conditions are met, the result is fed to the $\mathsf{Loc}_{\mathsf{KE}}$ instance. Finally, the adversary can challenge an explicit session i to test, by calling $\mathsf{Test}(i)$. This oracle will return either the honest participant (the initial k) set of true keys, if $b = 0$, or the associated random keys kept in the fake list. The winning event guess is defined to occur when $b = b'$ in the end of the game.

Theorem 5 (One-to-many AttKE utility)**.** *If the* AttKE *is correct and secure, and the* AC *protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the utility experiment: the probability that the adversary violates the* AttKE *two-sided entity authentication is negligible; and the key secrecy advantage* $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

The intuition behind this proof very closely follows the formula to the one of One-to-one utility theorem. A sketch is now presented. There is a sequence of four games, beginning with the original experiment in Figure 6.15. On the second game, the possibility for the adversary to forge an attested output us restricted, which can be bound as the advantage for attacking the original AC scheme. Afterwards, the behaviour of the machine is replaced with a simulator, which is also indistinguishable if the underlying AC scheme provides minimal leakage. Finally, all calls to the key exchange are replaced by calls to oracles for the AttKE scheme, therefore reducing to attacks on the AttKE scheme. In the final game, the adversary's

decision is based on a bit that is unrelated to any information obtainable via the oracles. All these hops match the ones described in the proof for Theorem 4, but in a setting generalised for multiple local participants.

Labelled AttKE utility

Labelled AttKE is a flavour of utility relying on a LAC scheme. Here, it is allowed for the key exchange to be composed in parallel with other arbitrary programs and then sequentially with another program, where the execution is being attested according to specific labels. The utility experiment intuitively states that the adversary cannot distinguish between a derived key and a random key, whenever the key exchange has been performed between an honest party and a remote machine running $\mathsf{Rem}_{\mathsf{KE}}$ within an IEE, and $\mathsf{Rem}_{\mathsf{KE}}$ is composed with other arbitrary programs as described above. The reason this parallel/sequential composition pattern does not harm security is that the parts of the state belonging to different parallel-composed programs are disjoint, and sequential composition only reveals controlled information to other programs. Indeed, sequential composition in the utility theorem is restricted to pass only specific parts of the state of the key exchange program to the following phase: mapping function ϕ_{key} passes on the derived key, the session and party identifiers, and the state ($\mathsf{derived}$ or accept) of the key exchange. Contrary to AC utility, this is not enough to define the mapping function, as other programs composed in parallel with the remote key exchange need to pass states to the next phase as well. To that extent, if $\phi_1, \ldots, \phi_n$ are mapping functions, let $\phi_1^{l_1} | \ldots | \phi_n^{l_n}$ be defined as $\phi^* := \mathsf{st}.l_i \mapsto \phi_i(\mathsf{st}.l_i)$. If the state comes from the program $\langle P_1 | \ldots | P_n \rangle_{(l_1,\ldots,l_n)}$, this mapping function maps the state belonging to each P_i using ϕ_i. In this composition context, the ϕ_i is taken according to the key exchange to be ϕ_{key}. This ensures that only the key is transmitted to the following stage of the protocol, and not information supposed to be local to the key exchange protocol and hence not intended for further use.

In the experiment in Figure 6.16, the adversary has to distinguish between an ideal machine and a real world machine where an AttKE is run in parallel with other programs in the first phase of a LAC-compiled protocol. The machine $\mathcal{M}$ represents the remote machine expected by the LAC protocol and the machine $\mathcal{M}'$ is a modification of machine $\mathcal{M}$ in which the key derived by a key-exchange session is magically replaced by a fresh key. In order to maintain consistency between the tested keys and the keys used in $\mathcal{M}'$, oracle $\mathcal{M}'.\mathsf{Run}$ takes two additional parameters: a list fake of pairs of keys and a flag tweak. If the flag is activated, the following modifications in the behaviour of $\mathcal{M}$ occur in $\mathcal{M}'$:

- It expects the sub-program being activated due to input label l to be a key exchange $\mathsf{Rem}_{\mathsf{KE}}$ instance. After running its transition function, $\mathcal{M}'$ checks if it has reached the $\mathsf{derived}$ or accept state. If so, it retrieves the derived key and if there is no association

(key, _) in fake it generates a fresh key key^* and appends $(\mathsf{key}, \mathsf{key}^*)$ to fake.

- Furthermore, if the key exchange process has entered accept state, it performs $\mathsf{st}.l.\mathsf{key} \leftarrow \mathsf{fake}(\mathsf{key})$, i.e., it replaces the derived key with a fake random one. Note that this will cause the fake key to be passed to the next stage of the sequentially composed program.

The oracles provided to the adversary offer access to the remote machine. Additionally, the adversary can create new sessions of the key exchange using the NewSession oracle, where the remote key exchange is composed in parallel (with label l^*) with programs $P_1, \ldots, P_n$, followed with Q and compiled for LAC. Parallel composition of $P_1, \ldots, P_n$, and its sequential composition with Q follow the respective composition notions for labelled programs as described in Figure 5.1. Note that, given the structure of the presented parallel composition, the position in which a program is listed in the composition expression is irrelevant. The adversary makes the local part of the key exchange progress by using the Send oracle, provided that the message passes the LAC verification step for the relevant label. Finally the adversary can challenge a session by executing the Test oracle, which return either the real key of a fake key according to b (provided that the key exchange has reached a derived or accept state).

Game $\mathsf{Att}_{\mathsf{AttKE},\mathcal{A}}(1^\lambda, \mathsf{id})$:

$\mathsf{prms}_0 \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$\mathsf{prms}_1 \leftarrow\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{PrgList} \leftarrow [\,]$
$\mathsf{fake} \leftarrow [\,]$
$i \leftarrow 0$
$b \leftarrow\$\ \{0,1\}$
$b' \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{prms}_b)$
Return $b = b'$

Oracle $\mathsf{Test}(i)$:

If $\mathsf{st}^i_{\mathsf{KE}}.\delta \neq \mathsf{accept}$: Return $\bot$
If $b = 0$: Return $\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key}$
Else: Return $\mathsf{fake}(\mathsf{st}^i_{\mathsf{KE}}.\mathsf{key})$

Oracle $\mathsf{Send}(o^*, i)$:

$o \leftarrow \mathsf{LAC.Verify}[\mathsf{st}^i_V](\mathsf{prms}, l^*_i, \mathsf{in}^i_{\mathsf{last}}, o^*)$
If $o = \bot$: Return $\bot$
$\mathsf{msg}^* \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}[\mathsf{st}^i_{\mathsf{KE}}](o)$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{msg}^*$
If $\mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
  $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
  $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
Return msg^*

Oracle $\mathsf{NewSession}(P_1, l_1, \phi_1, \ldots, P_n, l_n, \phi_n, l^*, Q, L^*)$:

If $\exists j, k$ such that $j \neq k \wedge l_j = l_k$: Return $\bot$
If $(p, (l^*, \epsilon)) \notin L^*$: Return $\bot$
$i \leftarrow i + 1$
$l^*_i \leftarrow (p, (l^*, \epsilon))$
$(\mathsf{st}^i_{\mathsf{KE}}, \mathsf{Rem}^i_{\mathsf{KE}}) \leftarrow\$\ \mathsf{Setup}(1^\lambda, \mathsf{id})$
$\mathsf{in}^i_{\mathsf{last}} \leftarrow \epsilon$
$\mathsf{RemComp} := \langle \mathsf{Rem}^i_{\mathsf{KE}} | P_1 | \ldots | P_n \rangle_{(l^*, l_1, \ldots, l_n)}$
$\phi^* := \phi^{l^*}_{\mathsf{key}} | \phi^{l_1}_1 | \ldots | \phi^{l_n}_n$
$R_i := \langle\, \mathsf{RemComp}\,;\, Q \,\rangle_{\phi,p,q}$
$R^*_i \leftarrow\$\ \mathsf{LAC.Compile}(\mathsf{prms}_b, R_i, L^*)$
$\mathsf{st}^i_V \leftarrow (R^*_i, L^*)$
$\mathsf{PrgList} \leftarrow (R^*_i, l^*_i) : \mathsf{PrgList}$
Return R^*_i

Oracle $\mathsf{Load}(R^*)$:

$\mathsf{hdl}_0 \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
$\mathsf{hdl}_1 \leftarrow \mathcal{M}'.\mathsf{Load}(R^*)$
Return hdl_b

Oracle $\mathsf{Run}(\mathsf{hdl}, l, x)$:

$o_0 \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, l, x)$
$\mathsf{tweak} \leftarrow \mathsf{F}$
If $(\mathsf{Program}_{\mathcal{M}'}(\mathsf{hdl}), l) \in \mathsf{PrgList}$ then $\mathsf{tweak} \leftarrow \mathsf{T}$
$(o_1, \mathsf{fake}) \leftarrow\$\ \mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, l, x, \mathsf{tweak}, \mathsf{fake})$
Return o_b

Figure 6.16: Utility of adversarially composed AttKE.

Theorem 6 (Labelled AttKE utility)**.** *If the AttKE is correct and secure, and the LAC protocol is correct, secure and ensures minimal leakage, then for all ppt adversaries in the*

labelled utility experiment: the probability that the adversary violates the AttKE two-sided entity authentication is negligible; and the key secrecy advantage $2 \cdot \Pr[\mathsf{guess}] - 1$ *is negligible.*

The previously presented proof for Theorem 4 consists in a sequence of 4 games $\mathsf{G0}_{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}_{\mathsf{AttKE},\mathcal{A}}$. The first hop removes the possibility of an AC forgery, thus ensuring that messages are delivered properly from the remote program executing the key exchange to the local party. The second hop, using minimal leakage, replaces execution in the remote machine by a simulated execution based on a real execution of the compiled program. The final game hop replaces the execution of the remote key exchange by call to the AttKE security oracles, and concludes immediately by security of the AttKE. Due to the similarities of both proofs, the whole proof is not rewritten, instead the differences coming the different notion of attestation and the more general composition patterns are highlighted. For brevity, the original proof will be referred to as the *one-to-one* proof.

The presented proof is a sequence of four games $\mathsf{G0}'_{\mathsf{AttKE},\mathcal{A}}$ to $\mathsf{G3}'_{\mathsf{AttKE},\mathcal{A}}$. The following description relates how the sequence of games can be constructed with respect to the one-to-one proof.

First game hop The first game $\mathsf{G0}'_{\mathsf{AttKE},\mathcal{A}}$ is the utility game presented in Figure 6.16. In the one-to-one proof, the first hop consists in adding a forgeAC event in the Send oracle to ensure that the initial segment of the trace witnessed by local party matches the initial segment of a valid execution of the distant protocol. Here, similarly, a forgeLAC event is added, making sure that the subtrace corresponding to the appropriate label matches a remote execution, the Send oracle is replaced by the one described in Figure 6.17.

Oracle $\mathsf{Send}(o^*, i)$:
- $o \leftarrow \mathsf{LAC.Verify}[\mathsf{st}^i_V](\mathsf{prms}, l^i, \mathsf{in}^i_{\mathsf{last}}, o^*)$
- If $o = \perp$: Return $\perp$
- If $b = 0$ then $\mathcal{M} \leftarrow \mathcal{M}$ else $\mathcal{M} \leftarrow \mathcal{M}'$
- If $o \neq \perp \ \wedge \ \nexists \mathsf{hdl}$ s.t. $\mathsf{Program}_{\mathcal{M}}(\mathsf{hdl}) = R^*_i$.
  $\mathsf{Rev}(o : T^i_L) \sqsubseteq \mathsf{filter}[l^i]\mathsf{Trace}_{\mathcal{M}}(\mathsf{hdl})$: $\mathsf{forgeLAC} \leftarrow \mathsf{T}$
- $\mathsf{msg}^* \leftarrow\!\$\ \mathsf{Loc}_{\mathsf{KE}}[\mathsf{st}^i_{\mathsf{KE}}](o)$
- $\mathsf{in}^i_{\mathsf{last}} \leftarrow \mathsf{msg}^*; T^i_L \leftarrow \mathsf{msg}^* : o : T^i_L$
- If $\mathsf{st}^i_{\mathsf{KE}}.\delta \in \{\mathsf{derived}, \mathsf{accept}\} \wedge \mathsf{st}^i_{\mathsf{KE}}.\mathsf{key} \notin \mathsf{fake}$:
  - $\mathsf{key}^* \leftarrow\!\$\ \{0,1\}^\lambda$
  - $\mathsf{fake} \leftarrow (\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
- Return msg^*

Figure 6.17: Send oracle from $\mathsf{G1}'_{\mathsf{AttKE},\mathcal{A}}$

The correctness of this game hop follows from the same arguments as the one-to-one proof.

Second game hop The second game hop in the one-to-one proof consists in replacing the remote machine by the simulator, provided by the minimum leakage property. The minimal leakage property is exactly the same in AC and LAC, and this second game hop is exactly the same.

This second game enables reasoning on the semantic of the original code instead of the compiled code executed in an IEE. This allows for taking advantage of semantics for parallel and sequential composition in the next game hop.

Third game hop In the third game hop, the crucial point is emulating a run of the protocol using the oracles from the AttKE security game. As in the one-to-one proof, a list of instances of key exchanges related to the various programs is kept, updated in the Load oracle. The NewSession oracle creates a new instance of $\mathsf{Rem}_{\mathsf{KE}}$ using the NewLoc oracle and the Send oracle uses the SendLoc oracle, exactly as in the one-to-one proof. The crucial modifications appear in the Run oracle and are presented in Figure 6.18.

Oracle Run(hdl, (l, in)):
  $(R_i^*, \mathsf{st}, j, \mathsf{stage}) \leftarrow$ List[hdl]
  If $(R_i^*, l_0) \in$ PrgList and $R_i^* = \mathsf{LAC.Compile}(\langle\langle P|P_1 \ldots |P_l\rangle_{(l_0,l_1,\ldots,l_n)}; Q\rangle_{(\phi_{\mathsf{key}}|\phi_1|\ldots|\phi_n),p,q})$:
    If stage $= 1$:
      If $l = (p, l_0, \epsilon)$:
        If st.finished.l: Return (F, ϵ)
        $o \leftarrow\$$ SendRem(in, i, j)
        Parse $(o, \mathsf{sid}, \delta, \mathsf{pid}) \leftarrow o$:
        If $\delta =$ accept:
          st.finished.$l \leftarrow$ T
          st.l.key $\leftarrow$ TestRem(i, j)
      Else If $l = (p, l_k, l')$
        $o \leftarrow P_k[\mathsf{st}.l_k](l', \mathsf{in})$
        st.finished.$l_k \leftarrow o$.finished
        If st.finished.l_k: st.$l_k \leftarrow \phi_k(\mathsf{st}.l_k)$
      $o \leftarrow (\wedge_{i=0}^{n}\mathsf{st.finished}.l_i, o)$
      $(o^*, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_2(\mathsf{hdl}, R^*, l, \mathsf{in}, o, \mathsf{st}_{\mathcal{S}})$
      If $(\wedge_{i=0}^{n}\mathsf{st.finished}.l_i, o)$: stage $= 2$
      $T_R^{\mathsf{hdl}} \leftarrow o^* : \mathsf{in} : T_R^{\mathsf{hdl}}$
    Else:
      $o \leftarrow\$ Q[\mathsf{st}](l, \mathsf{in})$
      $(o^*, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_2(\mathsf{hdl}, P, \phi, Q, R^*, \mathsf{in}, o, \mathsf{st}_{\mathcal{S}})$
  Else:
    $(o, \mathsf{st}, \mathsf{st}_{\mathcal{S}}) \leftarrow\$ \mathcal{S}_3(\mathsf{hdl}, R^*, \mathsf{in}, \mathsf{st}, \mathsf{st}_{\mathcal{S}})$
  List[hdl] $\leftarrow (R^*, \mathsf{st}, j, \mathsf{stage})$
  Return o^*

Figure 6.18: Run oracle from $\mathsf{G3}'_{\mathsf{AttKE},\mathcal{A}}$

Note that that for this last game hop to be valid, it is necessary that only the key and relevant parts of the key exchange state are passed through ϕ, which is ensured by the fact that the mapping function in the sequential composition is $(\phi_{\mathsf{key}}|\phi_1|\ldots|\phi_n)$. Additionally, the state of the key exchange has to be completely independent from the state of the other programs composed in parallel in order for one to be able to emulate it using the SendRem oracle. This property is ensured by the semantics of the parallel composition. With these remarks, it should be noted that the semantics of this third game is exactly the same as the semantics of $\mathsf{G2}'_{\mathsf{AttKE},\mathcal{A}}$, in a similar way to the one-to-one proof. Finally, observe that the adversary wins $\mathsf{G3}_{\mathsf{AttKE},\mathcal{A}}$ if it wins the AttKE security game (modulo the reduction simulating all non-AttKE

oracles), which concludes the proof.

Chapter 7

Secure Outsourced Computation

In the previous chapters, we presented formalisations that encapsulate the security guarantees provided by the usage of trusted hardware, and rigorously defined how one can employ these mechanisms for the establishment of secure channels between local participants and remote IEEs. We now propose and analyse the direct application of Secure Outsourced Computation relying on these hardware mechanisms.

7.1 Secure Outsourced Computation

The setting of secure outsourced computation considers a single local user, willing to outsource computation to a remote participant equipped with an IEE. Informally, two properties are required: i) that only the legitimate local user can pass inputs to the outsourced program and ii) that the I/O of the remote program is secret from any observer (even an actively malicious one). As such, first syntax will be provided for the protocols that solve this problem, then the formal definitions for the properties outlined above will be proposed, concluding with a generic construction combining a key-exchange for attestation, a scheme for attested computation and an authenticated encryption scheme.

A security definition is proposed for each one of these properties. It is also shown that it is easy to satisfy these definitions by building on top of an AttKE and an AC scheme. The main intuition here is using an AttKE over the AC protocol to established a shared authenticated encryption key between the (compiled) remote instance of the program and the local machine. It is then enough to use this shared key to build a secure channel between the remote and the local machine.

Syntax

A *Secure Outsourced Computation* scheme (SOC) for a remote machine $\mathcal{M}$ is defined by the following algorithms:

- $\mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$ is the program compilation algorithm. On input public parameters prms, a program P and a party identifier id, it outputs a compiled program P^*, together with an initial state st_l for the local side algorithms. It is assumed that initially $\mathsf{st}_l.\mathsf{accept} = \perp$. Note that unlike the AC compilation algorithm, this algorithm only takes one program as input, as this scheme is intended for providing guarantees for the whole trace and not only for an initial segment.

- $\mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$ is the client side initialization algorithm. On input public parameters prms, o (presumably the last message from the remote machine) and local state st_l, it returns the next message i to be delivered to the remote machine in the bootstrapping step, together with the updated local state. It is assumed that $\mathsf{BootStrap}$ sets an accept flag to T when the initialization process successfully terminates.

- $\mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$ is the verification algorithm. It fulfils the same function as the AC verification algorithm. Note that, as all the inputs are provided by the local machine, it is not necessary to feed it the last input, as it can be stored in the state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{T}$.

- $\mathsf{Encode}(\mathsf{prms}, i, \mathsf{st}_l)$ is the encoding algorithm. On input the public parameters, local state and the next intended input for P, it returns the next input i^* for P^* together with the updated local state. It is expected to return $\perp$ if $\mathsf{st}_l.\mathsf{accept} \neq \mathsf{T}$.

- $\mathsf{Attest}(\mathsf{prms}, \mathsf{hdl}, i)$ is, as in an AC scheme, the (untrusted) attestation algorithm.

A party A with identifier id who wants to outsource program P to the remote machine first compiles P with his id, thus obtaining P^* and some secret data st_l. It then loads P^* on the remote machine using some untrusted protocol. As it is, the program P^* is not ready to receive inputs intended for P: an initial bootstrapping phase (until $\mathsf{BootStrap}$ sets the accept flag) is necessary to establish some shared secrets between the IEE in which P^* is executed and A. Then when A wants to send an input to the remote execution, it encodes it using Encode, sends it (using Attest) and verifies the output provided by Attest using Verify.

In this section, for simplicity reasons, it is assumed that the program P is deterministic. However, it would be easy to extend all the definitions to a non-deterministic program.

Input Integrity

While security of attested computation aims at ensuring that a trace was honestly produced on the remote side, it does nothing to restrict the provenance of the inputs received.

It is now provided a stronger notion named *input integrity* which, intuitively, ensures that if a program is compiled by a party with identifier id, then only that party may use the remote compiled program. This property is ensured by making sure that the local and remote views coincide (up to the last message exchanged, which may not have yet been delivered). The following formula Ψ which relates two input/output traces captures this intuition.

$$\Psi(T, T') := T = T' \ \vee \ \exists o.\big(T = o :: T'\big)\ \exists i.\big(T' = i :: T\big)$$

The formalization that provided in Figure 7.1 is as follows. The adversary chooses a program P that is compiled with an honest party's id yielding P^* (which is given to the adversary). The adversary is given access to two oracles. A bootstrapping oracle that simply executes $\mathsf{BootStrap}$ honestly; and a send oracle that verifies the last (presumed) output of the remote program and encodes the next input (which is provided by the adversary), while keeping track of the local view of the trace. The goal of the adversary is then create a mismatch between the local and remote view of the trace.

The adversary is allowed in the experiment to use a local agent as oracle. This local agent initializes the remote process (using $\mathsf{BootStrap}$) then checks the last output of the remote machine and prepares the next input.

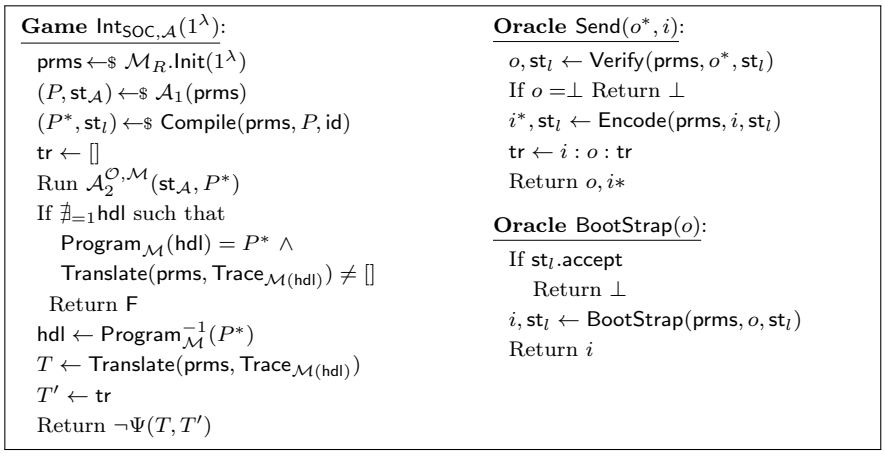


Figure 7.1: Input integrity of a SOC scheme

Definition 27 (Input Integrity)**.** *It is defined that a SOC scheme satisfies* input integrity *if there exists a polynomial time algorithm* $\mathsf{Translate}$ *such that for all ppt* $\mathcal{A}$ *the experiment described in Figure 7.1 returns true with probability negligible in the security parameter.*

Input privacy

Privacy of I/O is defined with an indistinguishability game. One important point here is that the class of programs considered is *length-uniform* (written lu) programs. A program is length uniform if the length of its outputs depends only on the length of its inputs. Intuitively, this is because the encryption scheme is allowed to leak the length of the messages, which in turn would leak information about the inputs for a non lu program.

The formalization described in Figure 7.2 is as follows. First, a bit b is chosen, that will determine whether the adversary will be talking with the left send oracle or the right send oracle (described later). As for input integrity, the adversary then chooses a program P. It is then compiled for an honest party's identifier and given the resulting P^* to the adversary. The adversary is also given access to the bootstrapping oracle. In addition, he is given access to a left or right send oracle. This oracle, on a request with the last candidate output of the remote machine and two inputs i_0 and i_1, verifies the last candidate output and, depending on the bit b, encodes either i_0 or i_1 and returns the result. The goal of the adversary is to guess the bit b with non-negligible bias from 1/2.

Game $\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
- $b \leftarrow\$ \{0,1\}$
- $\mathsf{prms} \leftarrow\$ \mathcal{M}_R.\mathsf{Init}(1^\lambda)$
- $(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
- If $\neg\mathsf{lu}(P)$ Return $b' \leftarrow\$ \{0,1\}$
- $(P^*, \mathsf{st}_l) \leftarrow\$ \mathsf{Compile}(\mathsf{prms}, P, \mathsf{id})$
- $b' \leftarrow \mathcal{A}_2(\mathsf{st}_\mathcal{A}, P^*)^{\mathcal{O},\mathcal{M}}$
- Return $b = b'$

Oracle $\mathsf{Send}_b(o^*, i_0, i_1)$:
- $o, \mathsf{st}_l \leftarrow \mathsf{Verify}(\mathsf{prms}, o^*, \mathsf{st}_l)$
- If $|m_0| \neq |m_1|$ Return $\perp$
- $i^*, \mathsf{st}_l \leftarrow \mathsf{Encode}(\mathsf{prms}, i_b, \mathsf{st}_l)$
- Return i^*

Oracle $\mathsf{BootStrap}(o)$:
- If $\mathsf{st}_l.\mathsf{accept}$
  - Return $\perp$ // (1 init max)
- $i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
- Return i

Figure 7.2: Input privacy of a SOC scheme

Definition 28. *A SOC scheme is defined to satisfy* input privacy *if, for all ppt $\mathcal{A}$, the experiment in Figure 7.2 returns true with probability* 1/2 *up to a negligible function.*

This definition ensures that there exist no two traces (with messages of the same length) played by an honest party over a SOC protocol that are distinguishable for an (active) adversary. This means that no adversary can gain information on the inputs sent out by a local machine using a SOC scheme, besides the length of the messages exchanged, achieving the established goal of hiding the honest party's inputs.

Definition 29. *A SOC scheme is said to be* secure *if it satisfies both input privacy and input integrity.*

7.2 An implementation of a secure SOC scheme

Having defined the security expected from a SOC scheme, it is now defined a scheme that satisfies these requirements. This construction is based on an AttKE, and an AC scheme. The main idea is using the AttKE to establish a key between the party agent and the IEE, and then communicate with the IEE over the secure channel established with this key.

Formally, let $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ be an AC scheme, $(\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}})$ be an AttKE and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an authenticated encryption scheme. Figure 7.3 defines a SOC scheme. The most important part is the compilation part, which uses the AC scheme compilation to compile the composition of the $\mathsf{Rem}_{\mathsf{KE}}$ program generated by Setup together with program P running over a secure channel (denoted by $C(P)$). The initial local state is the union of the state provided by the AC compilation and the AttKE setup. The program $C(P)$ simply decrypts the message it receives checks that the sequence number of the message matches its view the passes the decrypted message to P. It then retrieves the output of P, appends the corresponding next sequence number and outputs it. This mechanism ensures that all messages received (resp. sent out) by P^* after the bootstrapping phase have the form $\mathsf{Enc}(k, i\#m)$ where i is the position of the message in the trace, m is the message intended to (resp. produced by) P, and k is the key established by the AttKE.

On the local side, the bootstrapping mechanism simply consists of running the local KE over the AC protocol as already described in the utility definition. Once the key has been established, the local state keeps track of the local view of the sequence number. Verifying an output consists in decrypting it and checking that the sequence number against the local view of it. Encoding an input, is just appending the correct sequence number and encrypting it with the shared key.

Theorem 7. *If* $(\mathsf{Compile}, \mathsf{Attest}, \mathsf{Verify})$ *is an AC scheme ensuring correctness, security and minimal leakage,* $\{\mathsf{Setup}, \mathsf{Loc}_{\mathsf{KE}}, \mathsf{Rem}_{\mathsf{KE}}\}$ *is a secure AttKE and* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a secure authenticated encryption scheme, then the SOC presented in Figure 7.3 is secure.*

Integrity

The proof consists of two game hops described in Figure 7.4. The first hop from G_0 to G_1 consists in using the utility of the key exchange to replace the shared key by a magically shared fresh key. The second game hop from G_1 to G_2 is simply using sequence numbers and integrity of the encryption scheme to ensure that the local and remote traces actually match. The details of each game hop are now provided.

Let $\mathcal{A}$ be an adversary against G_0 or G_1, it is now built an adversary $\mathcal{B}$ against $\mathsf{Att}_{\mathsf{AttKE}}$. The machine $\mathcal{B}$ simulates $\mathcal{A}$ giving $\mathsf{NewSession}(P)$ as input to $\mathcal{A}_2$ instead of $\mathsf{Compile}^{\mathrm{sec}}(\mathsf{prms}, P, \mathsf{id})$, answering the oracle calls as follows:

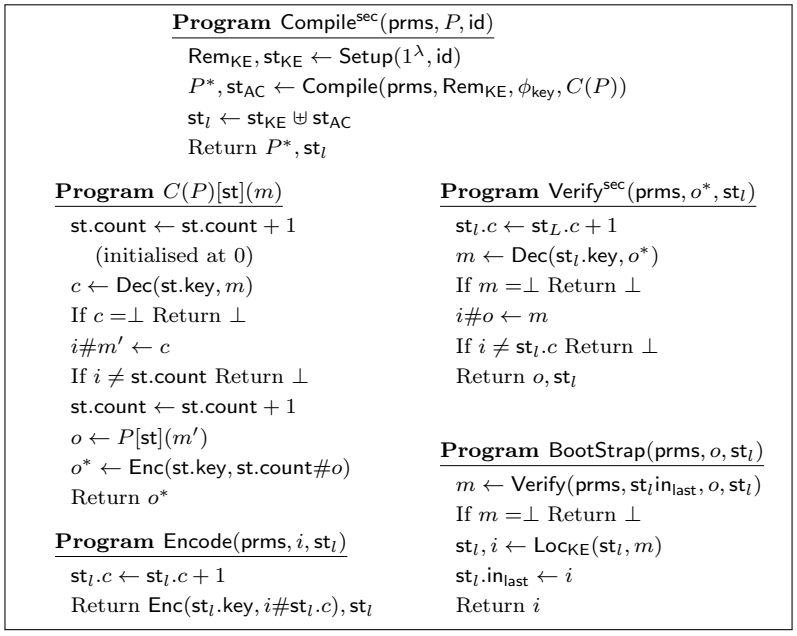


Figure 7.3: SOC algorithms

$\mathsf{BootStrap}(o)$: Return $\mathsf{Send}(o, 0)$

$\mathsf{Send}(o^*, i)$: $k \leftarrow \mathsf{Test}(0)$
$o, c \leftarrow \mathsf{Dec}(k, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(k, \mathsf{count}_l \# i)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

$\mathsf{Load}(R^*)$: Return $\mathsf{Load}(R^*)$

$\mathsf{Run}(\mathsf{hdl}, i)$: Return $\mathsf{Run}(\mathsf{hdl}, i)$

Observe that if the bit b chosen in the utility game is 0, the game being played by $\mathcal{B}$ is exactly the integrity game. On the other hand, if the bit chosen is 1, either $\mathcal{B}$ violates entity authentication or $\mathcal{B}$ behaves as $\mathcal{A}$ playing against G_1. It can therefore be concluded that

$$|G_0\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_1\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)| \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

First note that the only difference between G_1 and G_2 occurs when the forge event is raised (the other differences are simple rewritings). Now let $\mathcal{A}$ be an adversary against G_2. An adversary $\mathcal{B}$ is now built against the INT-CTXT game by simulating $\mathcal{A}$ playing G_2 using the encryption/decryption oracles provided by the INT-CTXT game. Note that the event forge

is raised if and only if $\mathcal{B}$ wins the INT-CTXT game. Indeed, remarking that the counters are strictly increasing ensure that no encryption can be accepted twice as input to Run or Send. This entails the fact that if a message is accepted, it was the last message produced. It can then be concluded that

$$|G_1\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_2\text{-}\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)| \leq \mathsf{Adv}^{\text{INT-CTXT}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec}),\mathcal{B}}(1^\lambda)$$

It is now enough to remark that in G_2, the Run oracle and the Send oracle agree on inputs and outputs (unless forge is raised). As G_2 always returns true, it follows that

$$\mathsf{Adv}^{\mathsf{Int}}_{\mathsf{SOC}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE}}(1^\lambda) + \mathsf{Adv}^{\text{INT-CTXT}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})}(1^\lambda)$$

Privacy

The proof consists of, as previously in replacing the derived key by a magically shared key. The second game hop, as in the integrity property, makes sure that the inputs received on both sides coincide. The game G_2 obtained reduces then quite simply to IND-CPA of the encryption scheme. The game hop is presented in Figure 7.5. The reduction from G_0 to G_1 is exactly the same as the one in the integrity proof. As previously (for some PPT $\mathcal{B}$),

$$\left|G_0\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_1\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)\right| \leq \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

Let $\mathcal{A}$ be an adversary against G_1 or G_2. It is now built an adversary $\mathcal{B}$ against the INT-CTXT game for $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$. The machine $\mathcal{B}$ simulates $\mathcal{A}$ playing the game G_1, with the exception of not drawing k and using the encryption and decryption oracle of the INT-CTXT game. Remark that G_1 and G_2 behave differently only if $\mathcal{A}$ is able to submit a forged encryption to either the send or the run oracle. Indeed, if the encryption submitted is neither the last produced encryption nor a forgery, the sequence number makes sure that the corresponding oracle return $\perp$. It follows that

$$\left|G_1\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - G_2\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)\right| \leq \mathsf{Adv}^{\text{INT-CTXT}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec}),\mathcal{B}}(1^\lambda)$$

The key point in the proof is reducing G_2 to the security of the authenticated encryption scheme. Let us call Chal the IND-CPA challenge oracle and Enc the encryption oracle in the IND-CPA game for $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$. Let $\mathcal{A}$ be an adversary against G_2-Priv. The following adversary $\mathcal{B}$ against the IND-CPA game for $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ is built. The machine $\mathcal{B}$ simulates the game G_2, without drawing the key k or the bit b and using the IND-CPA oracles to perform encryptions. In the Send oracle, instead of computing $\mathsf{Enc}(\mathsf{count}_l\#i_b, k)$, $\mathcal{B}$ calls $\mathsf{Chal}(\mathsf{count}_l\#i_0, \mathsf{count}_l\#i_1)$. Similarly, in the Run oracle, instead of computing $\mathsf{Enc}(\mathsf{count}\#o_b, k)$, $\mathcal{B}$ calls $\mathsf{Chal}(\mathsf{count}\#o_0, \mathsf{count}\#o_1)$. All calls are well formed as at each call of Send, the length of the two inputs is required to be identical and, in the Run

Game G_0-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:

$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}_\mathcal{M}^{-1}(P^*)$
$T \leftarrow \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}(\mathsf{hdl})})$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(i):

If st_l.accept Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):

$o, c \leftarrow \mathsf{Dec}(\mathsf{st}_l.\mathsf{key}, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(\mathsf{st}_l.\mathsf{key}, \mathsf{count}_l \# i)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):

$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i):

$o \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i)$
Return o

Game G_1-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:

$k \leftarrow\$\ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}_\mathcal{M}^{-1}(P^*)$
$T \leftarrow \mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}(\mathsf{hdl})})$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(o):

If st_l.accept Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):

If $\neg\mathsf{st}_l$.accept Return $\perp$
$o, c \leftarrow \mathsf{Dec}(k, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(k, \mathsf{count}_l \# i)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):

$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i^*):

If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\neg\mathsf{st}_\mathcal{M}(\mathsf{hdl})$.accept
  $o \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_\mathcal{M}(\mathsf{hdl})$.accept
  raise twoPartners
$i \# c \leftarrow \mathsf{Dec}(k, i^*)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
Return $\mathsf{Enc}(k, \mathsf{count} \# o)$

Game G_2-$\mathsf{Int}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:

$k \leftarrow\$\ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{tr}_R \leftarrow []$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$\ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
Run $\mathcal{A}_2^{\mathsf{BootStrap},\mathsf{Send},\mathsf{Run},\mathsf{Load}}(st_\mathcal{A}, P^*)$
If $\nexists_{=1}\mathsf{hdl}$ such that
  $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^* \wedge$
  $\mathsf{Translate}(\mathsf{prms}, \mathsf{Trace}_{\mathcal{M}(\mathsf{hdl})}) \neq []$
 Return F
$\mathsf{hdl} \leftarrow \mathsf{Program}_\mathcal{M}^{-1}(P^*)$
$T \leftarrow \mathsf{tr}_R$
$T' \leftarrow \mathsf{tr}$
Return $\neg\Psi(T, T')$

Oracle BootStrap(o):

If st_l.accept Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle Send($o*, i$):

If $\neg\mathsf{st}_l$.accept Return $\perp$
$o, c \leftarrow \mathsf{Dec}(k, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
If $o \neq \mathsf{last}_o$ Raise forge
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(k, \mathsf{count}_l \# i)$
$\mathsf{last}_i \leftarrow i$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle Load(R^*):

$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle Run(hdl, i^*):

If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) \neq P^*$
  $o^* \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\neg\mathsf{st}_\mathcal{M}(\mathsf{hdl})$.accept
  $o \leftarrow\$\ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
  Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$
    $\wedge \mathsf{st}_\mathcal{M}(\mathsf{hdl})$.accept
  raise twoPartners
$i \# c \leftarrow \mathsf{Dec}(k, i^*)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
If $i \neq \mathsf{last}_i$ Raise forge
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$\mathsf{last}_o \leftarrow o$
$\mathsf{tr}_R \leftarrow o : i : \mathsf{tr}_R$
Return $\mathsf{Enc}(k, \mathsf{count} \# o)$

Figure 7.4: Game hops for integrity of the SOC scheme

oracle, P is assumed length-uniform. It is now enough to remark that $\mathcal{B}$ wins the IND-CPA game if and only if $\mathcal{A}$ wins G_0 to conclude

$$\left| G_2\text{-}\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda) - \frac{1}{2} \right| \leq \mathsf{Adv}^{\text{IND-CPA}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec}),\mathcal{B}}(1^\lambda)$$

From this result, everything can be summed up to obtain the advantage of an adversary against the privacy game:

$$\mathsf{Adv}^{\mathsf{Priv}}_{\mathsf{SOC}}(1^\lambda) \leq \mathsf{Adv}^{\text{IND-CPA}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec}),\mathcal{B}}(1^\lambda) + \mathsf{Adv}^{\text{INT-CTXT}}_{(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec}),\mathcal{B}}(1^\lambda) + \mathsf{Adv}^{\mathsf{utility}}_{\mathsf{AttKE},\mathcal{B}}(1^\lambda)$$

Game G_0-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
$o, c \leftarrow \mathsf{Dec}(\mathsf{st}_l.\mathsf{key}, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(\mathsf{st}_l.\mathsf{key}, \mathsf{count}_l \# i_b)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}(\mathsf{hdl}, i^*)$:
$o^* \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
Return o^*

Game G_1-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
If $\neg\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$o, c \leftarrow \mathsf{Dec}(k, o^*)$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
If $c \neq \mathsf{count}_l$ Return $\perp$
$\mathsf{count}_l \leftarrow \mathsf{count}_l + 1$
$i* \leftarrow \mathsf{Enc}(k, \mathsf{count}_l \# i_b)$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}(\mathsf{hdl}, i^*)$:
If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) \neq P^*$
$o^* \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
Return o^*
If $\neg\mathsf{st}_\mathcal{M}(\mathsf{hdl}).\mathsf{accept}$
$o \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i^*)$
Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$
$\wedge \mathsf{st}_\mathcal{M}(\mathsf{hdl}).\mathsf{accept}$
raise twoPartners
$i \# c \leftarrow \mathsf{Dec}(k, i^*)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
If $c \neq \mathsf{count}_R$ Return $\perp$
$o \leftarrow P[\mathsf{st}_P](i)$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
Return $\mathsf{Enc}(k, \mathsf{count} \# o)$

Game G_2-$\mathsf{Priv}_{\mathsf{SOC},\mathcal{A}}(1^\lambda)$:
$k \leftarrow\$ K(1^\lambda)$
$\mathsf{st}_P \leftarrow \emptyset$
$\mathsf{count}_R \leftarrow 0$
$\mathsf{count}_l \leftarrow 0$
$\mathsf{prms} \leftarrow\$ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(P, \mathsf{st}_\mathcal{A}) \leftarrow\$ \mathcal{A}_1(\mathsf{prms})$
$P^*, \mathsf{st}_l \leftarrow \mathsf{Compile}^{\mathsf{sec}}(\mathsf{prms}, P, \mathsf{id})$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}}(st_\mathcal{A}, P^*)$
Return $b = b'$

Oracle $\mathsf{BootStrap}(o)$:
If $\mathsf{st}_l.\mathsf{accept}$ Return $\perp$
$i, \mathsf{st}_l \leftarrow \mathsf{BootStrap}(\mathsf{prms}, o, \mathsf{st}_l)$
Return i

Oracle $\mathsf{Send}_b(o*, i_0, i_1)$:
If $|i_0| \neq |i_1|$ Return $\perp$
If $o^* \neq \mathsf{last}_{o^*}$ Return $\perp$
$i* \leftarrow \mathsf{Enc}(k, \mathsf{count}_l \# i_b)$
$\mathsf{last}_{i_0} \leftarrow i_0$
$\mathsf{last}_{i_0} \leftarrow i_1$
$\mathsf{last}_{i^*} \leftarrow i^*$
$\mathsf{tr} \leftarrow i : o : \mathsf{tr}$
Return $o, i*$

Oracle $\mathsf{Load}(R^*)$:
$\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
Return hdl

Oracle $\mathsf{Run}_b(\mathsf{hdl}, i^*)$:
If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) \neq P^*$
$o^* \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i)$
Return o^*
If $\neg\mathsf{st}_\mathcal{M}(\mathsf{hdl}).\mathsf{accept}$
$o \leftarrow\$ \mathcal{M}.\mathsf{Run}(\mathsf{hdl}, i)$
Return o^*
If $\exists \mathsf{hdl}_2 \neq \mathsf{hdl}.\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$
$\wedge \mathsf{st}_\mathcal{M}(\mathsf{hdl}).\mathsf{accept}$
raise twoPartners
If $i \neq \mathsf{last}_{i^*}$ Return $\perp$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$o_0 \leftarrow P[\mathsf{st}_P^0](\mathsf{last}_{i_0})$
$o_1 \leftarrow P[\mathsf{st}_P^1](\mathsf{last}_{i_1})$
$\mathsf{count}_R \leftarrow \mathsf{count}_R + 1$
$o^* \leftarrow \mathsf{Enc}(k, \mathsf{count} \# o_b)$
$\mathsf{last}_{o^*} \leftarrow o^*$
Return o^*

Figure 7.5: Game hops in the privacy proof

Chapter 8

Secure Multiparty Computation

In the previous chapter, we focused on the one-to-one application of securely outsourcing program execution to a potentially untrusted machine. On this chapter, we further explore the applicability of Attested Computation on the multiple user settings of Secure Function Evaluation and Secure Multiparty Computation.

8.1 Secure Function Evaluation

The combination of Attested Computation with multiple parallel executions of AttKE (Theorem 5) enables for the deployment of a Secure Function Evaluation protocol. More specifically, this section considers multiple local participants willing to execute some arbitrary function over their private inputs, and obtain the resulting computation output.

Functionalities and execution model

Let n be the number of protocol participants. The goal is to securely execute a functionality $\mathcal{F}$ defined as $(\mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$, as follows.

- F is a (stateless) function to be evaluated. More precisely $(o_1, \ldots, o_n) \leftarrow \mathsf{F}(i_1, \ldots, i_n)$ is a function with n inputs and n outputs, one for each party.
- $\mathsf{Lin}(i, k)$ defines the public leakage that can be revealed by a protocol from a given input i by party k.
- $\mathsf{Lout}(o_1, \ldots, o_n)$ defines the public leakage associated with the outputs of F.

This setting considers a machine $\mathcal{M}$ allowing for the execution of non-labelled programs on isolated execution environments, as defined in Section 5.1. The machine $\mathcal{M}$ is assumed to be

adversarially controlled, with the isolation guarantees enforced by the IEEs. All the code that is run in $\mathcal{M}$, but outside these execution environments is also considered to be adversarially controlled. The goal is to guarantee that a set of parties can use the IEE capabilities of $\mathcal{M}$ securely, bar the possibility that $\mathcal{M}$ refuses to allow the protocol to proceed (this would amount to a denial-of-service attack, which is considered to be outside the scope of this work).

A protocol π for functionality $\mathcal{F}$ is a six-tuple of algorithms as follows:

- Setup – This is the party local set-up algorithm. Given the security parameter, the public parameters prms for machine $\mathcal{M}$, the party's identifier id, it returns the party's initial state st (including its secret key material) and its public information pub.
- Init – This is the party local protocol initialization algorithm. Given the party's state st produced by Setup, the description of a functionality F and the public information of all participants Pub, it prepares the code to be securely evaluated, compiles it and outputs an accordingly updated state st. Note that a party can choose to engage in a protocol by checking if the public parameters of all parties are correct and assigned to roles in the protocol that match the corresponding identities.
- AddInput – This is the party local input providing algorithm. Given the party's current state st and an input in, it outputs an updated state st.
- Process – This is the party local message processing algorithm. Given its internal state st, and an input message msg, it runs the next protocol stage, updates the internal state and returns output message msg'. Protocol termination will be locally signalled with an output message $\mathsf{msg}' = \perp$.
- Output – This is the party local output retrieval algorithm. Given internal state st, it returns the computation output o.
- Remote – This is the untrusted code that will be run in $\mathcal{M}$ and which ensures the correctness of the protocol by controlling its scheduling. It has oracle access to $\mathcal{M}$, and is run on public parameters prms, the handle to the IEE in which the compiled program is running and input message msg; it returns the output message msg' and the identity id of the party for which msg' is intended, as well as the participant position pos expected for the next message.

Security

The presented security definition is based on the experiments shown in Figure 8.1, where $\mathcal{O}$ represents access to all oracles except Fun, i.e., the adversary has oracle access to Run, Setup, SetInput, GetOutput and Send in both games.

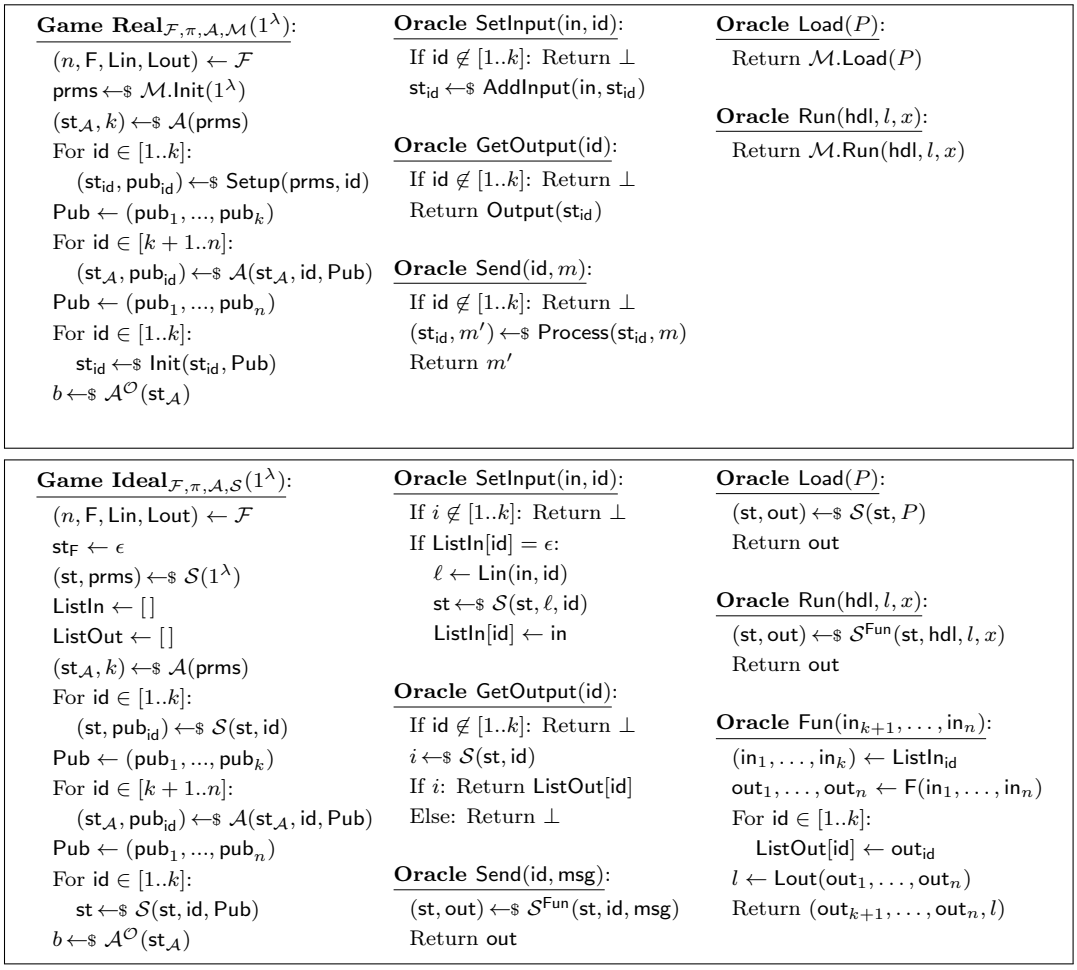


Figure 8.1: Real and Ideal security games for SFE. Fun can only be run once.

Definition 30. *π is said to be secure for $\mathcal{F}$ if, for any ppt adversary $\mathcal{A}$, there exists a ppt simulator $\mathcal{S}$ such that the following definition of advantage is a negligible function in the security parameter.*

$$|\,\mathsf{Pr}[\mathbf{Real}^{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow b = 1] - \mathsf{Pr}[\mathbf{Ideal}^{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow b = 1]\,|$$

As is customary in secure computation models, security analysis is done via the ideal world versus real world approach. The real world considers a remote machine $\mathcal{M}$, which is under the control of the adversary. A fixed number of n parties are also considered, where the first k of which are honest, and the rest corrupt. The experiment begins with the trusted setup of the machine, the attacker selectively choosing the number of corrupt parties, and the trusted initialization of the initial states of all honest parties. The adversary gets the public parameters of all honest parties (including the machine) and chooses those of the corrupt parties. All public parameters are assumed to be authenticated and available to all (e.g., via a PKI). Note that the adversary has all the information it needs to honestly execute the protocol and, in particular, it may properly compile and run Remote on its own, or choose

to arbitrarily deviate from it. The honest parties will initialise correctly and will therefore be able to take advantage of the attestation guarantees provided by the remote machine. Intuitively, this is the underlying setup assumption that enables secure computation to be done efficiently.

The adversary then takes control of the experiment execution and it has access to a series of oracles that allow it to fully control the remote machine (Load and Run) and interact with the honest parties: setting their inputs via SetInput, checking the outputs via GetOutput and delivering messages via Send. In this sense, the real world experiment is similar to the Universal Composability framework: the adversary represents the UC environment that controls party inputs and outputs combined with a *dummy adversary* that conveys communications between parties and the environment. When the adversary terminates, it outputs a bit b with its guess of whether it is executing in the real or ideal worlds.

As in the real world experiment, the ideal experiment begins with the global set-up procedure. The simulator is given full control of the remote machine, which is always assumed to be honest, and this includes the ability to generate its global parameters. The simulator is also given the ability to control the honest party parameter generation. The remaining parties are initialized in the standard way, and the adversary is run on an equivalent set of parameters as in the real world.

The central component in the ideal experiment is the functionality oracle Fun. This oracle represents an idealized execution, very much like the ideal functionality in the UC setting. The adversary can provide inputs and read outputs from the functionality via the SetInput and GetOutput oracles; this ability applies only to the inputs of honest parties. The scheduling of the computation of the functionality F is controlled by the simulator, as is the setting of the inputs corresponding to corrupt parties. As in the UC framework, this can be interpreted as an idealization of the operation of the functionality where some of its participants are dishonest: from this perspective, the simulator should be seen as an adversary *attacking* the ideal functionality and, as such, it should control the dishonest inputs.

Succinctly, the proposed security model is inspired in the UC framework, and can be derived from it when natural restrictions are imposed: PKI, static corruptions, and a distinguished non-corruptible party modeling an SGX-enabled machine.[1] A security proof for a protocol in this model can be interpreted as translation of any attack against the protocol in the real world, as an attack against the ideal functionality in the ideal world. The simulator performs this translation by presenting an execution environment to the adversary that is consistent with what it is expecting in the real world. It does this by simulating the operations of the Load, Run and Send oracles, which represent the operation of honest parties in the protocol. While the adversary is able to provide the inputs and read the outputs for honest parties

[1] This particular choice in this model has implications for the composability properties of these results, as will be discussed in a later section.

Program $\mathsf{Box}\langle \mathsf{F}, \Lambda \rangle(\mathsf{msg}_1, \ldots, \mathsf{msg}_n)$:
For $i \in [1..n]$:
$\mathsf{inp}_i \leftarrow \Lambda.\mathsf{Dec}(\mathsf{key}_i, \mathsf{msg}_i)$
$(\mathsf{out}_1, \ldots, \mathsf{out}_n) \leftarrow \mathsf{F}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)$
For $i \in [1..n]$:
$o_i \leftarrow \Lambda.\mathsf{Enc}(\mathsf{key}_i, \mathsf{out}_i)$
Return $(o_1, \ldots, o_n)$

Figure 8.2: Boxing using Authenticated Encryption

directly from the functionality, the simulator is only able obtain partial leakage about this values via the Lin function, which in this work is assumed to just reveal the lengths of their inputs. Conversely, it can obtain the functionality outputs for corrupt parties and leakage of honest outputs via the Fun oracle (which relies on Lout for output leakage, also assumed to reveal nothing but output length).

Similarly to the real world, the adversary will finish the interaction by outputting a bit b containing its guess of which world it is executing in.

Boxing using authenticated encryption

After the bootstrapping stage, the protocol will be running the ideal functionality within an isolated execution environment, and using secure channels to communicate with each participating party. The availability of these secure channels will follow from Theorem 5. As such, it is now formalized the concept of *boxing a functionality*, which defines a program that executes the functionality code, but receives inputs and delivers outputs using secure channels. This is done in the form of the Box construction presented in Figure 8.2. This construction takes a functionality $\mathcal{F}$ for n parties and a secure authenticated encryption encryption scheme Λ. This allows for $\mathsf{Box}\langle \mathcal{F}, \Lambda \rangle$ to be constructed, whose initial state is assumed to contain n symmetric keys compatible with scheme Λ, denoted key_1 to key_n (one for each participating party). Upon inputs $\mathsf{msg}_1, \ldots, \mathsf{msg}_n$, the program decrypts msg_i with sk_i to obtain input inp_i. It then passes all decrypted inputs to the functionality to produce outputs $\mathsf{out}_1, \ldots, \mathsf{out}_n$, and subsequently encrypts them with the respective key. Finally, the encrypted outputs are returned.

Generic construction

It can now be presented a construction of a protocol that relies on AC and AttKE to bootstrap secure function evaluation. The main property of this construction is that messages are bundled between all local clients and the remote IEE executing the protocol. All n local participants receive messages that encompass the responses $(\mathsf{msg}_1, \ldots, \mathsf{msg}_n)$, and are identified by their position $\mathsf{pos} \in [1..n]$. Their respective input is to be accessed by Process,

according to the participant's position in the protocol. Remote is responsible for collecting every individual message. As it receives all n inputs, it concatenates them and feeds it to the IEE executing the protocol.

As previously explained, Remote is untrusted code to run in $\mathcal{M}$ to ensure the correctness of the protocol. The joint execution of Setup, Init and Process is responsible for preventing adversarial tampering of the trace. The presented protocol is composed of six algorithms: Setup, Init, Process, AddInput, GetOutput and Remote. These algorithms are expressed in Figure 8.3 and will now be described in brief.

- $\mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathsf{id}, \mathsf{pos})$ starts participant of identifier id and position pos in the protocol, by generating its key exchange program $\mathsf{Pub}_{\mathsf{id}} = \mathsf{Rem}_{\mathsf{KE},\mathsf{id}}$ and initializing its state st. The local state is now ready to execute Init ($\mathsf{stage} \leftarrow 1$).
- $\mathsf{Init}(\mathsf{st}, \mathsf{Pub})$ composes the set of key exchange programs in Pub, prepares the code F to be securely evaluated, and compiles it to R^*. The internal state st is then updated with the computed verification sate st_V. The local state is now ready to receive AddInput ($\mathsf{stage} \leftarrow 2$). Note that this assumes that all parties agree on all other parties' public parameters prior to executing this algorithm.
- $\mathsf{AddInput}(\mathsf{st}, \mathsf{in})$ will simply update the local state st with the provided input in. The local state is now ready to execute Process ($\mathsf{stage} \leftarrow 3$).
- $\mathsf{Process}(\mathsf{msg}, \mathsf{st})$ while the key exchange is executing ($\mathsf{stage} = 3$), verifies the attestation for the input received and if the included input of its position matches the last input provided ($\mathsf{in}_{\mathsf{last}}$), generates the next response for the key exchange and updates its state. After completing the key exchange ($\mathsf{stage} = 4$), Process extracts input in from its state, encrypts it with the exchanged key key to produce an encrypted output. Finally, after sending its input ($\mathsf{stage} = 5$), Process expects an output encrypted using the same shared key key, decrypting it and storing the result in out. From this point onwards, Process will not execute ($\mathsf{stage} = 6$).
- $\mathsf{Output}(\mathsf{st})$ will simply return the value stored in variable out. This means that, before Process reaching its completion, the output will be $\perp$, otherwise it will be the result of securely evaluating F.
- $\mathsf{Remote}(\mathsf{st}_R, \mathsf{prms}, \mathsf{Pub}, \mathsf{msg})$ begins by composing and compiling the code, similarly to what is performed in Init, and loading the result into $\mathcal{M}$. Remote also keeps track of the messages produced by $\mathcal{M}$ and received from all local participants in $\mathsf{in}[\mathsf{pos}]$, as well as the next message to produce p. p defines the position of the participant that will receive the next message, so the behaviour highly depends on the value of p:
  - $p = 1$ means that a message must be provided for the first participant. This triggers Remote to collect the input received into $\mathsf{in}[n]$ (if the next participant is

the first, then the received message came from the last participant, i.e. n), run the IEE via $\mathsf{AC.Attest}$ for the collected inputs in (ϵ for the first execution) and update the record of the last inputs provided lin. The received output will be stored in out, the internal state st_R is updated and the message out is returned alongside the last input given to the IEE (lin), signalled to be delivered to participant p.

– $p \in [2, \ldots, n]$ stores the received input in $\mathsf{in}[p-1]$ (if the next response goes to p, then the received message was from $p-1$), updates its state and returns the previously computed $(\mathsf{out}, \mathsf{lin})$ flagged for participant p.

Algorithm $\mathsf{Setup}(1^\lambda, \mathsf{prms}, \mathsf{id}, \mathsf{pos})$:
$(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{st}_V \leftarrow \epsilon$; $\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$; $\mathsf{in} \leftarrow \epsilon$; $\mathsf{out} \leftarrow \epsilon$
$\mathsf{st} \leftarrow (\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, 1)$
Return $(\mathsf{st}, \mathsf{Rem}_{\mathsf{KE}})$

Algorithm $\mathsf{AddInput}(\mathsf{st}, \mathsf{msg})$:
$(\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, \mathsf{stage}) \leftarrow \mathsf{st}$
If $\mathsf{stage} = 2$:
  $\mathsf{st} \leftarrow (\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{msg}, \mathsf{out}, 3)$
  Return st
Else: Return $\perp$

Algorithm $\mathsf{Process}(\mathsf{msg}, \mathsf{st})$:
$(\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, \mathsf{stage}) \leftarrow \mathsf{st}$
$(\mathsf{id}, \mathsf{st}_{\mathsf{KE}}, \mathsf{sk}, t) \leftarrow \mathsf{st}_L$
If $\mathsf{stage} = 3$:
  $(\mathsf{ins}, \mathsf{outs}^*) \leftarrow \mathsf{msg}$
  $(\mathsf{outs}', \mathsf{st}_V) \leftarrow \mathsf{AC.Verify}(\mathsf{prms}, \mathsf{ins}, \mathsf{msg}, \mathsf{st}_V)$
  If $\mathsf{outs}' = \perp$: Return $\perp$
  $(\mathsf{ins}, \mathsf{outs}) \leftarrow \mathsf{outs}'$
  If $\mathsf{in}_{\mathsf{last}} \neq \mathsf{ins}[\mathsf{pos}]$: Return $\perp$
  $(o, \mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_L, \mathsf{outs}[\mathsf{pos}])$
  $\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $(\mathsf{st}_{\mathsf{KE}}.\delta) \in \{\mathsf{derived}, \mathsf{accept}\}$
    $\mathsf{stage} \leftarrow 4$
Else If $\mathsf{stage} = 4$:
  $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_{\mathsf{KE}}.\mathsf{key}, \mathsf{in})$
  $\mathsf{stage} \leftarrow 5$
Else If $\mathsf{stage} = 5$:
  $(\mathsf{ins}, \mathsf{outs}) = \mathsf{msg}$
  $\mathsf{out} \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{st}_{\mathsf{KE}}.\mathsf{key}, \mathsf{outs})$
  $\mathsf{stage} \leftarrow 6$; $o \leftarrow \epsilon$
Else: Return $\perp$
$\mathsf{st} \leftarrow (\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, \mathsf{stage})$
Return (st, o)

Algorithm $\mathsf{Init}(\mathsf{st}, \mathsf{Pub}, \mathsf{in})$:
$(\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, \mathsf{stage}) \leftarrow \mathsf{st}$
If $(\mathsf{stage} \neq 1)$:
  Return $\perp$
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$\mathsf{RemComp} := \mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle$
$Q := \mathsf{Box}\langle \mathsf{F}, \Lambda \rangle$
$(R^*, \mathsf{st}_V) \leftarrow \mathsf{AC.Compile}(\mathsf{prms}, \mathsf{RemComp}, \phi_{\mathsf{key}}, Q)$
$\mathsf{st} \leftarrow (\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, 2)$
Return st

Algorithm $\mathsf{Remote}^{\mathcal{M}}(\mathsf{st}_R, \mathsf{prms}, \mathsf{Pub}, \mathsf{msg})$:
$n \leftarrow \mathsf{Length}(\mathsf{Pub})$
If $\mathsf{st}_R = \epsilon$:
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $\mathsf{RemComp} := \mathsf{ParComp}\langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle$
  $Q := \mathsf{Box}\langle \mathsf{F}, \Lambda \rangle$
  $(R^*, \mathsf{st}_V) \leftarrow \mathsf{AC.Compile}(\mathsf{prms}, \mathsf{RemComp}, \phi_{\mathsf{key}}, Q)$
  $\mathsf{hdl} \leftarrow \mathcal{M}.\mathsf{Load}(R^*)$
  For $i \in [1..n]$:
    $\mathsf{in}[i] \leftarrow \epsilon$
  $p = 1$
Else: $(\mathsf{hdl}, \mathsf{in}, \mathsf{lin}, \mathsf{out}, p) \leftarrow \mathsf{st}_R$
If $p = 1$:
  $\mathsf{in}[n] \leftarrow \mathsf{msg}$
  $\mathsf{out} \leftarrow\$ \mathsf{AC.Attest}^{\mathcal{M}}(\mathsf{prms}, \mathsf{hdl}, \mathsf{in})$
  $\mathsf{lin} \leftarrow \mathsf{in}$
Else $\mathsf{in}[p-1] \leftarrow \mathsf{msg}$
If $(p = n)$ $p' = 1$
Else $p' \leftarrow p + 1$
$\mathsf{st}_R \leftarrow (\mathsf{hdl}, \mathsf{in}, \mathsf{lin}, \mathsf{out}, p')$
Return $(\mathsf{st}_R, p, (\mathsf{lin}, \mathsf{out}))$

Algorithm $\mathsf{Output}(\mathsf{st})$:
$(\mathsf{prms}, \mathsf{st}_L, \mathsf{id}, \mathsf{pos}, \mathsf{st}_V, \mathsf{in}_{\mathsf{last}}, \mathsf{in}, \mathsf{out}, \mathsf{stage}) \leftarrow \mathsf{st}$
Return out

Figure 8.3: SFE Protocol algorithms.

Theorem 8. *If* LAC *is a correct and secure LAC scheme,* AttKE *is a secure AttKE scheme and* Λ *a secure authenticated encryption scheme, then the protocol in Figure 8.3 is correct and secure for any functionality* $\mathcal{F}$ *that leaks input/output length.*

It is now presented a proof sketch. The following section proposes a more refined protocol generalised for MPC, whose full proof follows the same intuition as this one. The proof consists of two game hops for the real world scenario, reaching a real world setting for which it is trivial for a simulator to present an accordingly indistinguishable view.

It begins with the real world described in this section. The first hop is to an alternative scenario in which the exchanged keys between all honest participants and the remote IEE are replaced with randomly generated ones, whose correctness can be reduced to Theorem 5. In a second hop, the encryption of inputs from honest participants and outputs to honest participants are replaced by encryptions of zeros, which given that the adversary has no access to the associated keys, is given by the IND-CPA property of the encryption scheme used.

This scenario is now trivial for the simulator to emulate, since it no longer requires knowledge of information to which the simulator does not have access to, namely the honest parties' inputs and outputs. The simulator can, therefore, simply initialize and execute the protocol according to its description, modulo some additional safety mechanisms (e.g. assuring that no inputs are provided before initialization and that no outputs are delivered before protocol termination).

8.2 Multiparty Computation from LAC

In Section 8.1, we proposed an application for Attested Computation for the design of a secure SFE protocol. However, the usage of "vanilla" Attested Computation implied that every participant must be given the full trace of the computation for validating outputs produced by the IEE. We now extend the goal to providing a protocol for secure MPC, now taking the alternative design path of employing the Labelled Attested Computation described in Section 5.2. The idea is to ensure the same level of security against powerful active adversaries, while requiring for a much more scalable amount of data for local verification of attested outputs. This is achieved by having LAC ensure a secure channel between the IEE and all participants, where the labelled cryptographic material required for every participant is restricted to its own interaction with the IEE.

Functionalities and execution model

The notions presented in Section 8.1 are here refined to consider reactive functionalities. More specifically, the presented definition of $\mathcal{F}$ is the same, with the caveat of now being seen as a deterministic stateful transition function that takes inputs of the form (id, i), where id is a party identifier, an integer in the range $[1..n]$ where n denotes the number of participating parties. On each transition, F produces an output that is intended for party id, as well as

an updated state. Also refining the previous approach, F is associated two leakage functions $\mathsf{Lin}(k, i, \mathsf{st})$ and $\mathsf{Lout}(k, o, \mathsf{st})$ which define the public leakage that can be revealed by a protocol about a given input i or output o for party k, respectively; for the sake of generality, both functions may depend on the internal state st of the functionality, although this is not the case in the examples provided in this work. Arbitrary reactive functionalities formalized in the Universal Composability framework can be easily recast as transition functions such as these. The upside of the presented approach is that one obtains a precise code-based definition of what the functionality should do (which is central to this thesis since these descriptions give rise to concrete programs); the downside is that the code-based definitions may be less clear to a human reader, as one cannot ignore the tedious *book-keeping* parts of the functionality.

The execution model considered is the same as the one introduced in Section 8.1, but now one considers machine $\mathcal{M}$ to allow for running labelled programs on isolated execution environments. In secure computation terms, this machine should *not* be seen as an ideal functionality that enables some hybrid model of computation, but rather as an additional party that comes with a specific setup assumption, a fixed internal operation, and which cannot be corrupted.[2] The same adversarial power over $\mathcal{M}$ is also considered in this setting.

Since this setting considers the usage of LAC and reactive functionalities, the definition of protocol π for functionality $\mathcal{F}$ must also be refined to the seven-tuple of algorithms as follows:

- Setup – This is the party local set-up algorithm. Given the security parameter, the public parameters prms for machine $\mathcal{M}$ and the party's identifier id, it returns the party's initial state st (including its secret key material) and its public information pub.

- $\mathsf{Compile}$ – This is the (deterministic) code generation algorithm. Given the description of a functionality F, and the public parameters $(\mathsf{prms}, \mathsf{Pub})$ for both the remote machine and the entire set of public parameters for the participating parties, it generates the instrumented program that will run inside an IEE.

- Remote – This is the untrusted code that will be run in $\mathcal{M}$ and which ensures the correctness of the protocol by controlling its scheduling and input collection order. It has oracle access to $\mathcal{M}$, and is run on public parameters prms, the handle to the IEE in which the compiled program is running and input message msg; it returns the output message msg', the identity id of the party for which msg' is intended, and a flag inreq that indicates whether party id is expected to provide an input at this step of the protocol. Its initial state describes the order in which inputs of different parties should be provided to the functionality.

[2] Relating this to the Universal Composability framework, this special party $\mathcal{M}$ does not take inputs or outputs, and is accessible only via its communications tape, which is assumed to be controlled by the adversary.

- Init – This is the party local protocol initialization algorithm. Given the party's state st produced by Setup and the public information of all participants Pub it outputs an updated state st. Note that a party can choose to engage in a protocol by checking if the public parameters of all parties are correct and assigned to roles in the protocol that match the corresponding identities.
- AddInput – This is the party local input providing algorithm. Given the party's current state st and an input in, it outputs an updated state st.
- Process – This is the party local message processing algorithm. Given its internal state st, and an input message msg, it runs the next protocol stage, updates the internal state and returns output message msg'. Protocol termination will be locally signalled with an output message $\mathsf{msg}' = \perp$.
- Output – This is the party local output retrieval algorithm. Given internal state st, it returns the current output o.

Correctness

The following definition formalizes the notion of n users *correctly running a function evaluation protocol* π for F.

Definition 31. *π is said to be correct for functionality $\mathcal{F}$ on m inputs in r rounds if, for all λ, and all adversaries $\mathcal{A}$, the experiment in Figure 8.4 always returns* T*.*

This correctness definition considers an honest execution environment, but includes a correctness adversary that is in charge of finding problematic inputs for the protocol and potentially erroneous execution schedules. It is parametrized by a number of inputs m and a number of rounds r.

The first stage of the experiment executes the Setup and Init algorithms that initialise both the remote machine and the parties' local states, and collects the public parameters for all of these participants (which are assumed to be authenticated throughout the thesis, e.g., using a PKI). In the second part of the experiment, the adversary chooses a sequence of m inputs for the functionality, interleaving different parties in an arbitrary way. The sequence $(\mathsf{id}_1, \ldots, \mathsf{id}_m)$ implicitly defines a schedule for the execution of the proposed protocol, which should ensure that the inputs of each party are provided to the functionality in precisely this order.

The last stage of the correctness experiment emulates the protocol execution, alternating between local steps and remote steps. The Remote algorithm commands the scheduling of message exchanges; this algorithm is always invoked first and its output indicates the next party to be activated, the message this party will receive, and whether or not the party is

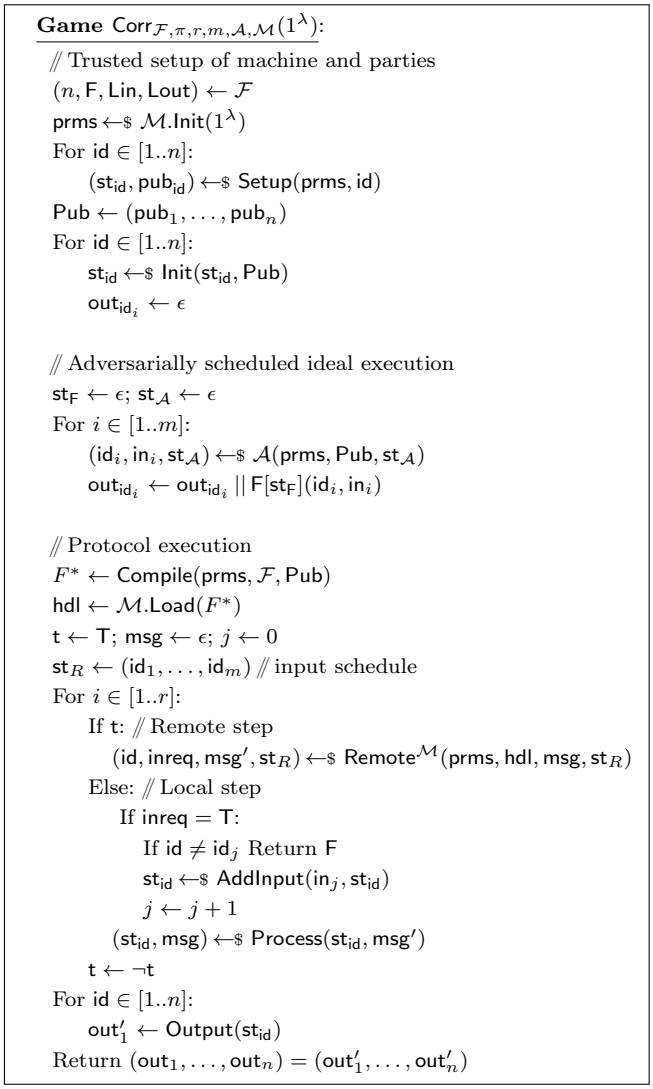


Figure 8.4: Game defining protocol correctness.

expected to provide an input. The protocol is run for r rounds, at which point its outputs are retrieved via Output. The adversary wins the game if it can force the game to produce a set of outputs that wouldn't be obtained by simply running the functionality F with the given inputs in the provided order.

The correctness experiment shows the crucial scheduling role of the Remote algorithm, which is run in an untrusted environment in the remote machine. Here, the work deviates from the standard approach in the UC setting, where the simulation-based definition of security is taken as providing sufficient detail to evaluate correctness of the protocol. Indeed, as other simulation-based definitions, the security experiment below will impose some input/output consistency conditions on the protocol. Intuitively, these must hold for any adversarially chosen Remote scheduling algorithm, as the adversary has full control of the remote machine and scheduling can be arbitrarily controlled by the attacker. However, the author believes

that there is added value in including a separate correctness definition, where the scheduling tasks of the non-security critical parts of the protocol can be specified as a first class feature of the protocol syntax. This also clarifies the envisioned execution model and makes it explicit that untrusted code running in an adversarially run machine is only relevant for correctness purposes.

Security

The presented security definition is based on the experiments shown in Figure 8.5, where $\mathcal{O}$ represents access to all oracles except Fun, i.e., the adversary has oracle access to Run, Setup, SetInput, GetOutput and Send in both games.

Definition 32. *π is said to be secure for $\mathcal{F}$ if, for any ppt adversary $\mathcal{A}$, there exists a ppt simulator $\mathcal{S}$ such that the following definition of advantage is a negligible function in the security parameter.*

$$|\,\mathsf{Pr}[\mathbf{Real}^{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow b = 1] - \mathsf{Pr}[\mathbf{Ideal}^{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow b = 1]\,|$$

The security definition is presented in Figure 8.5, which also refines the SFE definition presented in Figure 8.1. This model must now consider labelled execution and reactive functionalities. Modulo the usage of labels at programs run on $\mathcal{M}$, the most meaningful difference relates to the functionality oracle Fun, and how inputs are gathered via SetInput and GetOutput.

Since it is no longer the case that F is a one-shot function, taking one input for every participant and producing one output respectively, SetInput and GetOutput will manage lists of inputs/outputs for every participant. These will be collected in SetInput and provided in GetOutput. For the latter, while in Section 8.1 the simulator can simply block a participant's output to match the real world execution, it is now the case that $\mathcal{S}$ can select how many outputs can be revealed at every call to GetOutput. Furthermore, $\mathcal{S}$ is also able to control the rate and order in which all inputs are provided to the functionality. Were this not the case, the adversary would be able to distinguish the two worlds by manipulating scheduling in such a way that the simulator could not possibly match.

Boxing for MPC

The notion for boxing presented in Figure 8.2 captured a SFE functionality, in the sense that it is expected to run only once with all inputs, producing all outputs. Boxing reactive functionalities requires a more detailed notion for handling stateful execution, as follows. After the bootstrapping stage, the protocol will be running the ideal functionality within an isolated execution environment, and using secure channels to communicate with each

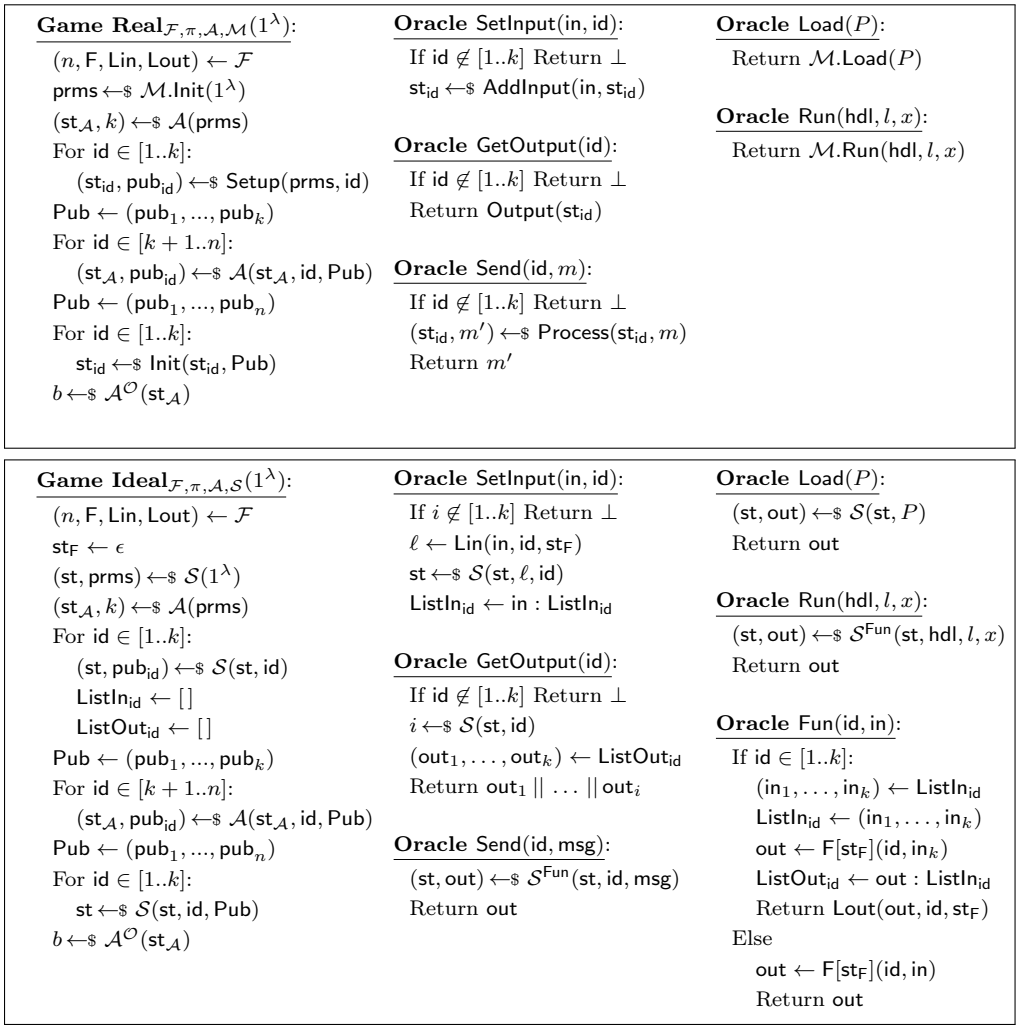


Figure 8.5: Real and Ideal security games.

participating party. The availability of these secure channels will follow from Theorem 6. The updated notion of *boxing a functionality* defines a program that executes the functionality code, but receives inputs and delivers outputs using secure channels. This is done in the form of the Box construction presented in Figure 8.6. This construction takes a functionality $\mathcal{F}$ for n parties and a secure authenticated encryption encryption scheme Λ. In this context, labelled program $\mathsf{Box}\langle\mathcal{F}, \Lambda\rangle$ is constructed, whose initial state is assumed to contain n symmetric keys compatible with scheme Λ, denoted key_1 to key_n (one for each participating party) and the empty initial state for the functionality st_F. To avoid replays of encrypted messages, one sequence number $\mathsf{seq}_{\mathsf{id}}$ per communicating party id is maintained. On input (i^*, l), the program interprets the label as party identity $\mathsf{id} = l$, decrypts input i^* with $\mathsf{sk}_{\mathsf{id}}$ to obtain input $i = (\mathsf{in}, \mathsf{seq})$, an checks the corresponding sequence number. It then passes the decrypted input to the functionality, and subsequently encrypts the output of the functionality back to the same user, together with the updated sequence number. The updated state includes

Program $\mathsf{Box}\langle\mathcal{F},\Lambda\rangle[\mathsf{st}](i^*,l)$:
- $(n,\mathsf{F},\mathsf{Lin},\mathsf{Lout}) \leftarrow \mathcal{F}$
- $\mathsf{id} \leftarrow l$
- If $\mathsf{id} \notin [1..n]$: Return $\perp$
- If $\mathsf{st.seq_{id}} = \epsilon$:
  - $\mathsf{st.seq_{id}} \leftarrow 0$
- $i \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st.key_{id}}, \mathsf{msg})$
- If $\mathsf{msg} = (\mathsf{in}, \mathsf{st.seq_{id}})$:
  - $o \leftarrow F[\mathsf{st.st_F}](\mathsf{id},\mathsf{in})$
  - $\mathsf{st.seq_{id}} \leftarrow \mathsf{st.seq_{id}} + 1$
  - $c \leftarrow\$\ \Lambda.\mathsf{Enc}(\mathsf{st.key_{id}}, (\mathsf{seq}, o))$
  - $\mathsf{st.seq_{id}} \leftarrow \mathsf{st.seq_{id}} + 1$
  - Return c
- Else: Return $\perp$

Figure 8.6: Boxing using Authenticated Encryption

the updated sequence number and the updated state of the functionality. The transition function finally externalizes this updated state and the encrypted output.

An MPC protocol from SGX

It is now described a secure computation protocol based on LAC that works for any functionality. This protocol starts by running bootstrap code in an isolated execution environment in the remote machine, which exchanges keys with each of the participants in the protocol. These key exchange programs are composed in parallel, as defined in Figure 5.1. Once this bootstrap stage is concluded, the code of the functionality, which is composed sequentially with the bootstrap stage, starts executing and it uses the secure channels established with each party to ensure that the collection of inputs and delivery of outputs is secure. The protocol builds upon the Labelled AttKE utility Theorem 6.16, for the usage of key exchange in the context of attestation.

Building on top of a LAC scheme, an AttKE scheme and the proposed Box construction, Figure 8.7 defines a general secure multiparty computation protocol that works for any (possibly reactive) functionality F. The core of the protocol is the execution of an AttKE for each participant in parallel, followed by the execution of the functionality F on the remote machine, under a secure channel with each participant as specified in the updated Box construct. More precisely:

- Setup derives the code for a remote key exchange program $\mathsf{Rem_{KE}}$ using the AttKE setup procedure. This code (which intuitively includes cryptographic public key material) is set to be the public information for this party. The algorithm also stores various parameters in the local state for future usage.
- Compile uses the LAC compilation algorithm on a program that results from the

parallel composition of all the remote key exchange programs for all parties, which is then sequentially composed with the boxed functionality. Sequential composition uses the special ϕ^*_{key} function that maps the keys derived by *all* the key exchange $\mathsf{Rem}_{\mathsf{KE}}$ instances into the initial state of the Box construction. The set of attested labels is restricted to those of form $(p, (\mathsf{id}_i, \epsilon))$, corresponding to the AttKE subprograms.

- Init locally recomputes the program that is intended for remote execution, as this is needed for attestation verification. The set of labels that define the locally recovered trace is set to those of the form $\{(p, (\mathsf{id}, \epsilon)), (q, \mathsf{id}))\}$, which correspond to those exactly matching the parts of the remote trace that are relevant for this party, namely its own key exchange and its own input/output relation with the functionality. Various parts of the local state that are used by Process are also initialized.

- Process is split into two stages. In the first stage it uses LAC with attested labels of the form $(p, (\mathsf{id}, \epsilon))$ to execute AttKE protocol and establish a secure channel with the remote program. In the second stage, it uses non-attested labels of the form (q, id), and it provides inputs to the remote functionality (on request) and recovers the corresponding outputs when they are delivered. The input sending process is initiated by passing an empty message into the algorithm, which triggers the encryption of the next input using the derived secret key from the first stage. A non-empty message input to this stage will trigger decryption and recovery of an output.

- Output reads the output in the state of the participant and returns it.

- AddInput adds an input to the end of the list of inputs that have to be transmitted by the participant.

The (untrusted) scheduling algorithm is shown in Figure 8.8. It is in charge of dispatching messages to/from the remote machine IEE using the Attest algorithm provided by the LAC, and animating the protocol to generate a correct execution for an arbitrary sequence of input-party interactions provided externally as an input schedule which is stored in its initial state. During the bootstrap stage, the Remote procedure interacts with one party at a time,[3] moving from one party to the next when the previous party has moved to its second stage. When all parties have completed the key exchange, the Remote procedure detects this in the output of the IEE (consistently with the properties of sequential composition), and moves to the functionality execution stage.

In this second stage, the algorithm simply follows the provided input schedule. Moving to the next input is triggered by feeding the algorithm with an empty input provided by the previous party (this is syntactic book-keeping to match the presented correctness requirement, and it

[3]Other options were possible for implementing Remote, and the core of the proposed protocol is actually compatible with a totally asynchronous scheduling. Dealing with such issues is out of the scope of this thesis.

algorithm $\mathsf{Setup}(\mathsf{prms}, \mathsf{id})$:
$\mathsf{st}_{\mathsf{id}}.\mathsf{id} \leftarrow \mathsf{id}$; $\mathsf{st}_{\mathsf{id}}.\mathsf{prms} \leftarrow \mathsf{prms}$
$(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L \leftarrow \mathsf{st}_L$; $\mathsf{st}_{\mathsf{id}}.\mathsf{pub} \leftarrow \mathsf{Rem}_{\mathsf{KE}}$
Return $(\mathsf{st}_{\mathsf{id}}, \mathsf{st}_{\mathsf{id}}.\mathsf{pub})$

algorithm $\mathsf{Compile}(\mathsf{prms}, \mathcal{F}, \mathsf{Pub})$:
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n} ; \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
$L^* \leftarrow \{(p,(1,\epsilon)), \ldots, \{(p,(n,\epsilon))\}$
$P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L^*)$
Return P^*

algorithm $\mathsf{Init}(\mathsf{st}_{\mathsf{id}}, \mathsf{Pub})$:
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} \leftarrow 0$;
$\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow 0$; $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow 1$; $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$
If $\mathsf{Pub}[\mathsf{st}_{\mathsf{id}}.\mathsf{id}] \neq \mathsf{st}_{\mathsf{id}}.\mathsf{pub}$: Return $\perp$
$(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
$P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n} ; \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
$L \leftarrow \{(p,(\mathsf{st}_{\mathsf{id}}.\mathsf{id},\epsilon)), (q,\mathsf{st}_{\mathsf{id}}.\mathsf{id})\}$
$\mathsf{st}_{\mathsf{id}}.\mathsf{st}_V \leftarrow (P, L)$
Return $\mathsf{st}_{\mathsf{id}}$

algorithm $\mathsf{AddInput}(\mathsf{in}, \mathsf{st}_{\mathsf{id}})$:
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList} + [\mathsf{in}]$
Return $\mathsf{st}_{\mathsf{id}}$

algorithm $\mathsf{Output}(\mathsf{st}_{\mathsf{id}})$:
Return $\mathsf{st}_{\mathsf{id}}.\mathsf{out}$

algorithm $\mathsf{Process}(\mathsf{st}_{\mathsf{id}}, \mathsf{msg})$:
// Bootstrap (attested labels)
if $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 0$:
  $(i, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_{\mathsf{id}}.\mathsf{prms}, (p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), \mathsf{in}_{\mathsf{last}}, \mathsf{msg}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, i)$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{st}_{\mathsf{KE}}.\delta) = \mathsf{accept}$: Then $\mathsf{stage} \leftarrow 1$
  $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
  Return $(\mathsf{st}_{\mathsf{id}}, \mathsf{msg}')$

// Execution (non-attested labels)
if $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 1$:
  If $\mathsf{msg} = \epsilon$: // Input requested
    $\mathsf{in} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
    $o \leftarrow\$\ \Lambda.\mathsf{Enc}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \mathsf{in}))$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} + 2$
    $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
    Return $(\mathsf{st}_{\mathsf{id}}, \mathsf{msg}')$
  Else: // Process received output
    $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, \mathsf{msg})$
    If $\mathsf{msg}' = (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}, \mathsf{out}')$:
      $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} + 2$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{out} \leftarrow \mathsf{out}'$
      $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)$
      Return $(\mathsf{st}_{\mathsf{id}}, \mathsf{msg}')$
    Else: Return $\perp$

Figure 8.7: General SMPC protocol.

signals the fact that the previous output was correctly delivered to the previous party). The consequence of such an action is that Remote signals that a new input should be requested from the next party in the schedule. When an actual input is received, this is passed into the IEE using an unnattested label of the form (q, id). The output is sent back to the same party.

For proving security, functionalities are restricted to consider to a particular leakage function: size of inputs/outputs. It is then established that a functionality $(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout})$ leaks size if it is such that Lin and Lout return the length of the inputs/outputs (i.e. $\mathsf{Lin}(k, x, \mathsf{st}) = \mathsf{Lout}(k, x, \mathsf{st}) = |x|$ for every k, x, st).

Theorem 9. *If* LAC *is a correct and secure LAC scheme,* AttKE *is a secure AttKE scheme and* Λ *a secure authenticated encryption scheme, then the protocol in Figure 8.7 and Figure 8.8 is correct and secure for any functionality* $\mathcal{F}$ *that leaks input/output length.*

This proof is made by simulation. First, the construction of simulator $\mathcal{S}$ is presented, with the task of interacting with $\mathcal{A}$ on behalf of honest participants of the protocol, i.e., $\mathcal{M}.\mathsf{Load}$,

```
algorithm Remote^M(prms, hdl, msg, st_R):
 // Initial message
 If msg = ϵ :
    msg ← (0, 1, ϵ) // Force bootstrap start
    st_R.IdList ← st_R // Input schedule
    st_R.stage ← 0

 // Bootstrap (attested labels)
 If st_R.stage = 0 :
    (stage_id, id, i) ← msg
    o ← LAC.Attest^M(prms, hdl, (p, (id, ϵ)), i)
    If o.stage = 1 : // IEE just finished bootstrap
      st_R.stage = 1; inreq ← T; msg' ← ϵ
      (id_1, ..., id_k) ← st_R.IdList; id = id_1
      st_R.IdList ← (id_2, ..., id_k)
      Return (id, inreq, msg', st_R)
    Else: // Just continue bootstrap
       If stage_id = 1 : // This id finished bootstrap
         id ← id + 1
         o ← LAC.Attest^M(prms, hdl, (p, (id, ϵ)), ϵ)
         inreq ← F; msg' ← o
         Return (id, inreq, msg', st_R)
       Else:
         inreq ← F; msg' ← o
         Return (id, inreq, msg', st_R)

 // Execution (non-attested labels)
 If st_R.stage = 1 :
   (stage_id, id, i) ← msg
   If i = ϵ : // Move to next input (empty incoming message)
     (id_1, ..., id_k) ← st_R.IdList
     inreq ← T; msg' ← ϵ
     id = id_1; st_R.IdList ← (id_2, ..., id_k)
     Return (id, inreq, msg', st_R)
   Else: // Process input and send output
     o ← M.Run(hdl, (q, id), i)
     inreq ← F; msg' ← o
     If IdList = [ ] : Then st_R.stage ← 2 // No additional inputs
     Return (id, inreq, msg', st_R)
```

Figure 8.8: SMPC protocol untrusted scheduler.

$\mathcal{M}$.Run and Send for parties 1 to k. Afterwards, arguments are presented for why adversary $\mathcal{A}$ cannot distinguish between this displayed interaction and the real world protocol execution.

Observe that, according to the experiment in Figure 8.5, despite being used in different contexts (e.g. the same $\mathcal{S}$ for emulating the machine and the presentation of outputs), the simulator can always distinguish to which call it is responding to. This is because it receives different inputs in different occasions, with exception of honest party initialization and output retrieval, whose orders are predictable (GetOutput will always provide $\mathcal{S}$ with an already initialized id). As such, for clarity of presentation, the simulator is described in Figure 8.9 (local participants) and Figure 8.10 (remote machine) with different behaviour for different

$\mathcal{S}(1^\lambda)$: // parameter initialization
  $\mathsf{prms}, \mathsf{st.sk} \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
  $\mathsf{st}.\lambda \leftarrow 1^\lambda$; $\mathsf{st.hdl} \leftarrow 0$; $\mathsf{st.fake} \leftarrow [\,]$
  Return $(\mathsf{st}, \mathsf{st.prms})$

$\mathcal{S}(\mathsf{st}, \mathsf{id})$: // party setup
  $(\mathsf{st}_L, \mathsf{Rem}_{\mathsf{KE}}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(\mathsf{st}.\lambda, \mathsf{id})$
  $\mathsf{st.id.st}_L \leftarrow \mathsf{st}_L$
  $\mathsf{st.stage} \leftarrow 0$
  Return $(\mathsf{st}, \mathsf{Rem}_{\mathsf{KE}})$

$\mathcal{S}(\mathsf{st}, \mathsf{id})$: // party initialization
  $\mathsf{st.id.InList} \leftarrow [\,]$; $\mathsf{st.id.stage} \leftarrow 0$;
  $\mathsf{st.id.seq_{in}} \leftarrow 0$; $\mathsf{st.id.seq_{out}} \leftarrow 1$; $\mathsf{st.id.in_{last}} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $\mathsf{st}.P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}; \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}}, p, q}$
  $\mathsf{st}.L \leftarrow \{(p, (\mathsf{id}, \epsilon)), (q, \mathsf{id})\}$
  $\mathsf{st.id.st}_V \leftarrow (\mathsf{st}.P, \mathsf{st}.L)$
  $L^* \leftarrow \{(p, (1, \epsilon)), \ldots, \{(p, (n, \epsilon))\}$
  $\mathsf{st}.P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, \mathsf{st}.P, L^*)$
  $\mathsf{st.id.InLeak} \leftarrow [\,]$; $\mathsf{st.id.InList} \leftarrow [\,]$
  Return st

$\mathcal{S}(\mathsf{st}, l, \mathsf{id})$: // add inputs
  $\mathsf{st_{id}.InLeak[id]} \leftarrow \mathsf{st_{id}.InLeak[id]} + [l]$
  Return st

$\mathcal{S}(\mathsf{st}, \mathsf{id})$: // output retrieval
  Return $(\mathsf{st.id.seq_{out}}/2) + 1$

$\mathcal{S}^{\mathsf{Fun}}(\mathsf{st}, \mathsf{id}, \mathsf{msg})$: // emulate local participant id
  If $\mathsf{st.id.stage} = 0$:
    $(i, \mathsf{st.id.st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st.prms}, (p, (\mathsf{id}, \epsilon)), \mathsf{st.id.in_{last}}, \mathsf{msg}, \mathsf{st.id.st}_V)$
    If $i = \perp$: Return $\perp$
    $(o, \mathsf{st.id.st}_L) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st.id.st}_L, i)$
    $\mathsf{st.id.in_{last}} \leftarrow o$
    If $\mathsf{st.id.st}_L.\mathsf{key} \notin \mathsf{st.fake} \wedge \mathsf{st.id.st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
      $\mathsf{key}^* \leftarrow\$ \{0,1\}^{\mathsf{st}.\lambda}$
      $\mathsf{st.fake} \leftarrow (\mathsf{st.id.st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
    If $\mathsf{st.id.st}_L.\delta = \mathsf{accept}$: $\mathsf{stage} \leftarrow 1$
    $\mathsf{msg}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, o)$
    Return $(\mathsf{st}, \mathsf{msg}')$
  If $\mathsf{st.id.stage} = 1$:
    If $\mathsf{msg} = \epsilon$:
      $l \leftarrow \mathsf{st.id.InLeak}[0]$
      $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st.id.InLeak}$
      $\mathsf{st.id.InLeak} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
      $\mathsf{in} \leftarrow \{0\}^l$
      $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st.id.st}_L.\mathsf{key}), (\mathsf{st.id.seq_{in}}, \mathsf{in}))$
      $\mathsf{st.id.InList}[\mathsf{st.id.seq_{in}}] \leftarrow o$
      $\mathsf{st.id.in_{last}} \leftarrow o$
      $\mathsf{st.id.seq_{in}} \leftarrow \mathsf{st.id.seq_{in}} + 2$
      $\mathsf{msg}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, o)$
      Return $(\mathsf{st}, \mathsf{msg}')$
    Else:
      $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st.id.st}_L.\mathsf{key}), \mathsf{msg})$
      If $\mathsf{msg}' = (\mathsf{st.id.seq_{out}}, \mathsf{out}')$:
        $\mathsf{st.id.seq_{out}} \leftarrow \mathsf{st.id.seq_{out}} + 2$
        $\mathsf{msg}' \leftarrow (\mathsf{st.id.stage}, \mathsf{id}, \epsilon)$
        Return $(\mathsf{st}, \mathsf{msg}')$
      Else: Return $\perp$

Figure 8.9: Description of simulator $\mathcal{S}$ with respect to emulating local participants.

calls. Notice that in this scenario there is no $\mathcal{M}$, however the simulator perfectly follows the description of $\mathcal{M}$ to emulate its behaviour. Following its description in Section 5.2, let $\mathsf{SMInit}(1^\lambda)$ be the initialization function of the security module, producing public parameters prms and internal state sk, and let $P^*[\mathsf{hdl_{st}}, \mathsf{sk}](l, i)$ be the execution of compiled P^* given the internal state $\mathsf{hdl_{st}}$ and private parameters sk, according to the description of the security module, producing (possibly attested) output o^*.

The behaviour detailed in $\mathcal{S}$ does not trivially entail indistinguishability from the real world on all cases. The two main differences between how the simulator handles calls and how the same instructions would be executed in the real world are highlighted in the presented figures, and are now further detailed.

- The simulator is replacing the exchanged keys associated with honest participants with randomly generated ones (fake), and using them throughout the second stage of the protocol.

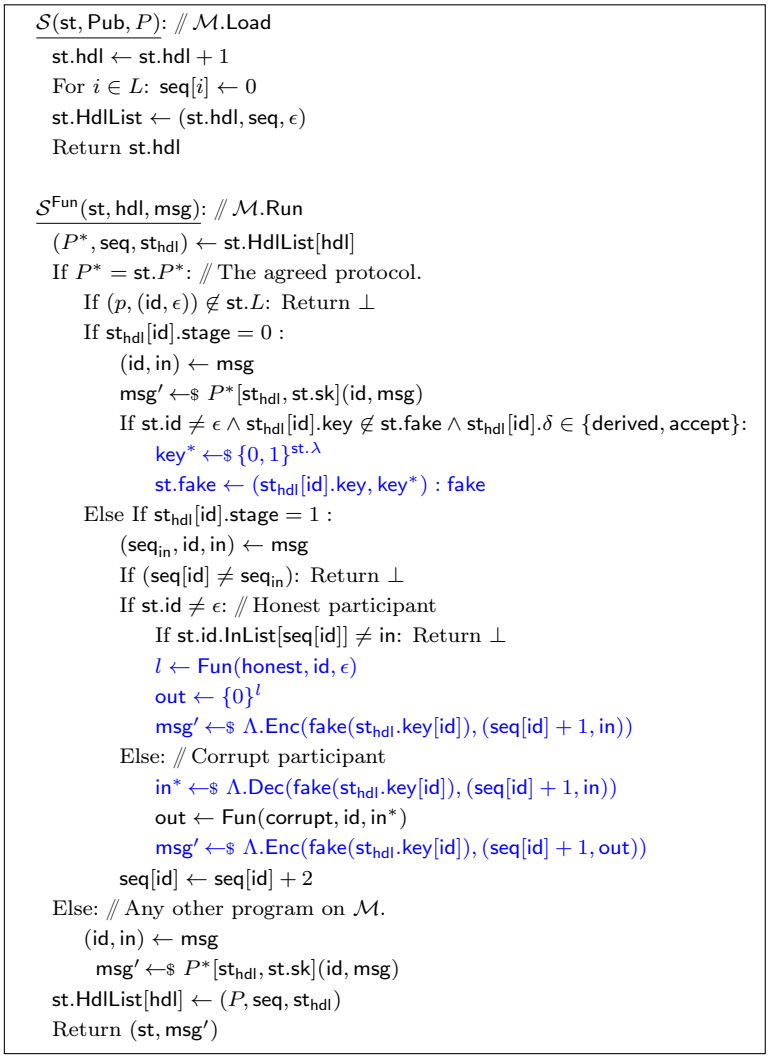


Figure 8.10: Description of simulator $\mathcal{S}$ with respect to emulating the remote machine.

- Instead of the honest participant's inputs and outputs, the simulator is encrypting strings of 0s with the same length as the real-world values (obtained by Lin and Lout).

It is now argued that, nevertheless, this provides an indistinguishable view for any $\mathcal{A}$. This is proven in three game hops from the real world, from Figure 8.11 to Figure 8.14. The first hop will replace $\mathcal{M}$ with the slightly different $\mathcal{M}'$, which replaces keys exchanged by honest participants by freshly generated keys (in exactly the same way the simulator is doing it). The correctness of this hop follows from the utility theorem, using a hybrid argument to replace keys of all k honest parties. Afterwards, the encrypted inputs/outputs of honest parties are replaced by encrypting dummy payloads of the correct length. The correctness of this hop follows from the indistinguishability of the underlying authenticated encryption scheme. Finally, the possibility of $\mathcal{A}$ to produce a forged encryption is restricted, by accordingly establishing a *bad* event. The correctness of this final hop follows from the unforgeability of

$\mathbf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}.\mathsf{Init}(1^\lambda)$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$\ \mathcal{A}(\mathsf{prms})$
For $\mathsf{id} \in [1..k]$:
  $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, \mathsf{st}_{\mathsf{id}}.\mathsf{pub}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
For $\mathsf{id} \in [k+1..n]$:
  $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_{\mathsf{id}}) \leftarrow\$\ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
For $\mathsf{id} \in [1..k]$:
  $\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} \leftarrow 0$;
  $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow 0$; $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow 1$; $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}\,;$
    $\mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
  $L \leftarrow \{(p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_{\mathsf{id}}.\mathsf{id})\}$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_V \leftarrow (P, L)$
$b \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{st}_\mathcal{A})$

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{M}.\mathsf{Load}(P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$:
Return $\mathcal{M}.\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
Return $\mathsf{st}_{\mathsf{id}}.\mathsf{out}$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{msg})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 0$:
  $(i, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_{\mathsf{id}}.\mathsf{prms}, (p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), \mathsf{in}_{\mathsf{last}}, \mathsf{msg}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, i)$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{st}_{\mathsf{KE}}.\delta) = \mathsf{accept}$: Then $\mathsf{stage} \leftarrow 1$
  $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
  Return msg'
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 1$:
  If $\mathsf{msg} = \epsilon$:
    $\mathsf{in} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
    $o \leftarrow\$\ \Lambda.\mathsf{Enc}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \mathsf{in}))$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} + 2$
    $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
    Return msg'
  Else:
    $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, \mathsf{msg})$
    If $\mathsf{msg}' = (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}, \mathsf{out}')$:
      $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} + 2$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{out} \leftarrow \mathsf{out}'$
      $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)$
      Return msg'
Else: Return $\perp$

Figure 8.11: Real world expanded.

the underlying authenticated encryption scheme.

The first game (Figure 8.11) is simply the real game expanded with the protocol instantiation. In this setting, whenever the adversary sets k as 0, i.e. corrupts all participants, the simulator already produces an indistinguishable view. In this case there are no honest inputs/outputs, so the simulator has access to all information and can therefore execute the protocol without replacing any keys and without encrypting any dummy payloads ($\mathsf{st.id} = \epsilon$ for all id), executing Fun whenever a corrupt input is provided to produce the corresponding output. As such, the following steps will only refer to situations in which $k \neq 0$, where indistinguishability is not yet established.

In the second game $\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$ (Figure 8.12), the machine in the ideal world is replaced with the machine M' of the Utility game for which $b = 1$. This machine performs exactly what the simulator is doing with the list fake, i.e., replacing keys for the first k participants whenever they finish the first stage of the protocol (the key exchange). This is possible via

$\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$\mathsf{prms} \leftarrow\$\ \mathcal{M}'.\mathsf{Init}(1^\lambda)$
$\mathsf{fake} \leftarrow [\,]$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$\ \mathcal{A}(\mathsf{prms})$
For $\mathsf{id} \in [1..k]$:
  $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, \mathsf{st}_{\mathsf{id}}.\mathsf{pub}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
For $\mathsf{id} \in [k+1..n]$:
  $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_{\mathsf{id}}) \leftarrow\$\ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
For $\mathsf{id} \in [1..k]$:
  $\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} \leftarrow 0$;
  $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow 0$; $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow 1$; $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n}\ ;\ \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
  $L \leftarrow \{(p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_{\mathsf{id}}.\mathsf{id})\}$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_V \leftarrow (P, L)$
  $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{st}_\mathcal{A})$

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Load}(P)$:
Return $\mathcal{M}'.\mathsf{Load}(P)$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$:
$\mathsf{flag} \leftarrow \mathsf{F}$
If $\mathsf{Program}_\mathcal{M}(\mathsf{hdl}) = P^*$: $\mathsf{flag} \leftarrow \mathsf{T}$
Return $\mathcal{M}'.\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg}, \mathsf{flag}, \mathsf{fake})$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{msg})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 0$:
  $(i, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_{\mathsf{id}}.\mathsf{prms}, (p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), \mathsf{in}_{\mathsf{last}}, \mathsf{msg}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L) \leftarrow\$\ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, i)$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
    $\mathsf{key}^* \leftarrow\$\ \{0,1\}^\lambda$
    $\mathsf{st}.\mathsf{fake} \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  If $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{st}_{\mathsf{KE}}.\delta) = \mathsf{accept}$: Then $\mathsf{stage} \leftarrow 1$
  $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
  Return msg'
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 1$:
  If $\mathsf{msg} = \epsilon$:
    $\mathsf{in} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
    $o \leftarrow\$\ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \mathsf{in}))$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} + 2$
    $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
    Return msg'
  Else:
    $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), \mathsf{msg})$
    If $\mathsf{msg}' = (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}, \mathsf{out}')$:
      $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} + 2$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{out} \leftarrow \mathsf{out}'$
      $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)$
      Return msg'
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
Return $\mathsf{st}_{\mathsf{id}}.\mathsf{out}$

Figure 8.12: First hop of the proof.

two steps.

Fix identity $\mathsf{id} = 1$. The behaviour of $\mathcal{M}$ in $\mathbf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)$ regarding this participant can be replaced using the Utility Theorem 6.16, for which $l^* = \mathsf{id}$. In this scenario, the key (both in $\mathcal{M}'.\mathsf{Run}$ and in Send) for that particular participant will be replaced by a fake one and stored in fake (as described in $\mathbf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)$, but only for $\mathsf{id} = 1$). The advantage gained by the adversary in this intermediate step is bound by its advantage in winning the experiment in Figure 6.16, by providing

$$\mathsf{Rem}^2_{\mathsf{KE}}, (p, (2, \epsilon)), (q, 2), \ldots \mathsf{Rem}^n_{\mathsf{KE}}, (p, (n, \epsilon)), (q, n), (p, (1, \epsilon)), \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle, \phi_{\mathsf{key}}$$

To $\mathsf{NewSession}$ on every call.

Now observe that, for any scenario in which m participants have had their keys replaced by

fake ones, it is possible to apply the same Utility theorem for replacing the keys of $m+1$ participants. In order to replace all k keys, it are therefore required to apply the same Utility theorem k times, and thus

$$\Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \mathsf{Adv}^{\mathsf{UT}}_{\mathsf{AttKE},\mathcal{A}}(\lambda) * \mathsf{k}.$$

In the third game $\mathbf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ (Figure 8.13), machine M' is opened and its behaviour is altered for instances of running program P^* on the second stage as follows:

- Upon receiving an honest participant input, instead of using it for computing Fun instead uses the first element of ListIn.
- When producing an honest participant output, instead of returning an encryption of the value received from F, it stores the value from F on an output list for this identity $\mathsf{OutList}$ and returns an encryption of zeros of the same length as the output.

Similarly, on the local side for instances running the second stage:

- When called for presenting the input, instead of encrypting the actual input, it stores it on a list of inputs InList and encrypts a string of zeros of the same length.
- Upon receiving an output, instead of decrypting and storing it on $\mathsf{ListOut}$, it retrieves the value of $\mathsf{OutList}$ and stores it on $\mathsf{ListOut}$.

The distance between these two games is upper bound, by constructing an adversary $\mathcal{B}$ against the indistinguishability of encryption scheme Λ such that

$$\Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{M}'}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G2}_{\mathcal{F},\pi}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I$$

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ as follows: it first has to try and guess which message will be used to distinguish. Let I be the maximum number of inputs adversary $\mathcal{A}$ chooses to input for any participant. It samples uniformly from $[1..k]$ a participant p, and from $[1..(I*2)]$ a message m. Since every input produces an output, one establishes that

- If $m \in [1..I]$, $\mathcal{B}$ picked the m-th input.
- If $m \in [I+1,\ldots,(I*2)]$, $\mathcal{B}$ picked the $\frac{m}{I}$-th output.

and proceed accordingly. $\mathcal{B}$ replaces all calls for encryption/decryption for inputs/outputs of participant p with similar calls to $\Lambda.\mathsf{Enc}$ and $\Lambda.\mathsf{Dec}$, with exception of the following. If $m \in [1..I]$, whenever $\mathsf{Send}(\mathsf{id},\mathsf{msg})$ for $\mathsf{id} = p$ is called for the m-th time on the second stage, $\mathcal{B}$ challenges $\mathsf{IND}^{\Lambda,\mathcal{B}}(1^\lambda)$ with message

$$((\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \mathsf{in}), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \{0\}^{|\mathsf{in}|}))$$

Otherwise, whenever $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$ for $l = p$ and $P = P^*$ (the agreed protocol) is called for the $\frac{m}{l}$-th time on the second stage, $\mathcal{B}$ challenges $\mathsf{IND}^{\Lambda,\mathcal{B}}(1^\lambda)$ with message

$$((\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}), (\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$$

Observe that any advantage $\mathcal{B}$ acquires in this transformation can be effectively used to distinguish between $\mathsf{G1}_{\mathcal{F},\pi,\mathcal{M}'}(1^\lambda)$ and $\mathsf{G2}_{\mathcal{F},\pi}(1^\lambda)$, since the only difference between the two games is the encryption of either the first message (the real value) or the second (the dummy payload with the same length). The two games are identical modulo this difference.

In the fourth game $\mathbf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ (Figure 8.14), the adversary loses whenever $\mathsf{authForge}$ event occurs. Intuitively, this event corresponds to the adversary producing an encryption that was not produced by either Send or $\mathcal{M}.\mathsf{Run}$ for $\mathsf{id} \in [1..k]$ (honest participant), and hence constitutes a forgery with respect to Λ. Given that the two games are identical until this event occurs, it follows that

$$\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda) \Rightarrow \mathsf{T}\,] \leq \Pr[\,\mathsf{authForge}\,].$$

The distance between these two games is upper bound, by constructing an adversary $\mathcal{C}$ against the existential unforgeability of encryption scheme Λ such that

$$\Pr[\,\mathsf{authForge}\,] \leq \mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{C}}(\lambda) * k$$

Adversary $\mathcal{C}$ simulates the environment of $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$ as follows: it first has to try and guess which session will produce the forgery. As such, it samples uniformly from $[1..k]$ a participant p and replaces the key generated for honest participant p (before adding to fake) with the key generated by $\Lambda.\mathsf{Gen}$. From there on, every time an encryption/decryption is requested for p, the same operation will be requested to $\Lambda.\mathsf{Enc}$ and $\Lambda.\mathsf{Dec}$, respectively.

When $\mathsf{authForge}$ is set, according to the rules of $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$, algorithm $\mathcal{C}$ outputs candidate encryption msg. It remains to show that this is a valid forgery. To see this, first observe that this is indeed a valid encryption, as decryption is performed on this value immediately before $\mathsf{authForge}$ occurs. It suffices to establish that message msg could not have been queried from the Λ oracle. Access to this oracle is only permitted on the encryption of inputs for this participant, and on outputs to this participant (when executing Run). From the construction of these operations and the sequence numbers they entail, it is known that producing such an encryption would only occur via the inclusion of msg in $\mathsf{authList}$. Since one can infer this is not the case, msg could not have been queried to the encryption oracle. It is therefore concluded that $\mathcal{C}$ outputs a valid forgery whenever $\mathsf{authForge}$ occurs.

Finally, it is argued that the behaviour displayed by the simulator is indistinguishable to what adversary $\mathcal{A}$ observes in game $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$. This is the case because the simulator no longer has private information to which he has no access to. In $\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$, only the length

of honest inputs and outputs is required to emulate their private inputs and outputs, to which $\mathcal{S}$ has access to via Lin and Lout. Additionally, the simulator uses the same message sequence numbers to prevent $\mathcal{A}$ from forcing an execution that deviates from the order in which inputs are provided and outputs are retrieved. Since the executions are the same for all other aspects (including key replacements to fake and exclusion of forged encryptions), $\mathcal{A}$ is provided the same view in both worlds.

Let

$$\mathsf{Adv}^{\mathsf{Distinguish}}_{\mathcal{F},\mathcal{A}} = \Pr[\,\mathsf{Real}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda) \Rightarrow \mathsf{T}\,] - \Pr[\,\mathsf{Ideal}_{\mathcal{F},\pi,\mathcal{A},\mathcal{S}}(1^\lambda) \Rightarrow \mathsf{T}\,]$$

To conclude,

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{Distinguish}}_{\mathcal{F},\mathcal{A}} &= \Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,] \\
&= (\Pr[\,\mathsf{G0}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}}(1^\lambda)\,] - \Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)\,]) + (\Pr[\,\mathsf{G1}_{\mathcal{F},\pi,\mathcal{A},\mathcal{M}'}(1^\lambda)\,] - \\
&\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,]) + (\Pr[\,\mathsf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,] - \Pr[\,\mathsf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)\,]) \\
&\leq \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{UT},\mathcal{A}}(\lambda) * k + \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I + \Pr[\mathsf{forgeAuth}] \\
&\leq \mathsf{Adv}^{\mathsf{Att}}_{\mathsf{UT},\mathcal{A}}(\lambda) * k + \mathsf{Adv}^{\mathsf{IND}}_{\Lambda,\mathcal{B}}(\lambda) * k * 2I + \mathsf{Adv}^{\mathsf{UF}}_{\Lambda,\mathcal{C}}(\lambda) * k
\end{aligned}$$

and Theorem 9 follows.

8.3 Relation to the UC approach

For completeness, this section now summarizes how the presented results map to the standard Universal Composability approach [Can01, CCL15], as well as to other approaches to formalise hardware guarantees.

Secure computation from theoretical tamper-proof hardware

Results in [CF01] demonstrate that, within the "plain" model, secure computation of general UC functionalities is impossible. In [Kat07], Katz tackles this issue by introducing a new setup assumption based on physical hardware requirements that demonstrably circumvents these impossibility results. This work was inspired by a somewhat similar approach in [CP92, Bra93, CP93], where *observers* can be used as trusted hardware for the deployment of e-cash applications. The central idea is that a bank could issue observers as smartcards for each user, which implemented an e-cash functionality. These entities would then behave as intermediaries between user and bank server, simultaneously enforcing passive behaviour on the users and ensuring user anonymity in the presence of a malicious bank.

Katz models this hardware as a wrapper functionality $\mathcal{F}_{\mathsf{wrap}}$ with two central operations: Creation and Execution. Creation allows for participants to take some software and compile it

insider this theoretical tamper-proof hardware token, alongside the participants' identities. After this initial step, the hardware token is sent to the other participant(s) to run Execution, which gives black-box access to the software embedded on the token. The following hardware assumptions are required to be met, in order for the proposed protocols to ensure security.

- The party running Creation is aware of the code that is to be run securely.
- The hardware token must ensure that the user running Execution cannot learn anything about the underlying code that would not be revealed by black-box access to its functionality. This also requires for the assumption that said hardware has a secure source of randomness.
- The party running Creation should no longer be able to send messages to the token, once it is sent to the remaining participant(s).

One can immediately find similarities between what is required from these theoretical hardware tokens, and what is necessary for one to make use of IEE-based guarantees as discussed in this thesis. The described Creation mechanism encapsulates both the machine initialization of $\mathcal{M}$.Init, and the execution of $\mathcal{M}$.Load for some specific protocol, making it functionally similar, albeit more restrictive in the sense that a malicious user cannot load/run additional (potentially parallel) programs into the hardware token. The first requirement is closely related to what is imposed for AC and LAC, where the program code is central in validating attested outputs via Verify, being either stored on the verification state as compilation output, or directly fed as algorithm parameter. The second requirement is similar to the minimal leakage property required for both flavours of attestation (Figures 5.4 and 5.9), intuitively necessary to ensure that a malicious user in possession of the hardware is unable to extract its secret cryptographic material. However, in the IEE abstraction, it is not necessary that the only information extractable from the underlying code is the one given by black-box access, as the code itself is allowed to be public. The third requirement is not as necessary for this thesis' setting, as the hardware provider is not considered to be a participant of the protocol, and thus the process of $\mathcal{M}$.Init is abstracted to be run securely before any protocol interaction. For real-world deployment of security-critical applications, however, it should certainly be the case that IEE providers are unable to establish covert channels with access to the secure hardware and its protected resources.

The work also proposes how to deploy a secure multiple commitment protocol using these hardware tokens. A core aspect of this design is that Creation must load a functionality that has a hard-wired key-pair $(\mathsf{pk}, \mathsf{sk})$, where sk will be used by the token to sign produced outputs, and pk will be used by the token creator to validate said outputs. This is also very similar to how IEEs are proposed to attest I/Os, where sk is seen throughout the thesis as the device key, and pk as public parameters prms. The model proposed for IEEs, however, considers trusted hardware that is not aware of any of the participant's identity,

since it is initialized by a third party. As such, contrary to Katz's approach where Creation fundamentally enables for the subsequent runs of Execution to produce outputs associated with the identity of the other participant, code loaded into IEEs must be adapted to include this information, which is what motivates the key exchange for attested computation (as described in Figure 6.3) to be parametrized with the public key of the participant attesting the execution.

Composability in the UC model

Another closely related approach is the one by Pass et al. [PST16]. In this work, the authors propose an abstraction to the general behaviour of modern hardware enabling for attested execution, and present several theoretical results, among which is the demonstration that achieving UC protocols for MPC without additional setup assumptions is impossible, unless all participants are equipped with IEE hardware.

The motivation presented is closely similar to ours. A formal approach to the security properties ensured by IEE hardware can benefit both users and software developers in: i.) unifying guarantees provided by different attestation hardware, ii.) enabling for a rigorous design and validation for security-critical software in a non-trivial paradigm such as the one made available by these technologies, and iii.) providing relevant implementation/deployment feedback for both the designers of trusted hardware, and for the developers of software to be run in said environments.

Behaviour of remote attestation hardware is abstracted as functionality $\mathcal{G}_{\mathsf{att}}$. $\mathcal{G}_{\mathsf{att}}$ is programmed to run a key generation upon initialization, producing $(\mathsf{pk}, \mathsf{sk})$ as a result – sk being the device key and pk the public material for verifying attestation. The functionality then makes $\mathsf{getpk}()$ available, allowing for the public parameters to be accessible at any time. Calls to enclaves are install, used to load programs into IEEs, and resume, used to run IEEs (or resume executions) with some provided input. Initialization matches the proposed $\mathcal{M}.\mathsf{Init}$, where the public key is directly provided to all participants in the protocol, and resume and install are reincarnations of the expected machine "regular" behaviour $\mathcal{M}.\mathsf{Load}$ and $\mathcal{M}.\mathsf{Run}$, respectively.

The major difference between the two approaches lies on how one represents the behaviour of IEE machines on the MPC security model, in particular in their initialization procedure. $\mathcal{G}_{\mathsf{att}}$ is seen as a global functionality, over which the simulator has no control over (other than basic interactions via its API). Our work considers that simulators are allowed to initialize our machine, as is the case with the security analysis of Functional Encryption [LOS+10] protocols, which enables the extraction of outputs in an intuitive way.

[PST16] trusted hardware is abstracted as a global functionality $\mathcal{G}_{\mathsf{att}}$. On the ideal world, simulator $\mathcal{S}$ must emulate the protocol behaviour with only access to the public pa-

rameters initialized by $\mathcal{G}_{\mathsf{att}}$, which it can retrieve via $\mathsf{getpk}()$. Without trapdoors on the programs run nor additional setup options, output extraction is demonstrably not possible.

This work trusted hardware is abstracted as a local functionality $\mathcal{M}$. On the ideal world, simulator $\mathcal{S}$ must emulate the protocol behaviour, but can program $\mathcal{M}$ in a way that allows for the extraction of corrupt outputs, which can be intuitively achieved by having $\mathcal{S}$ performing $\mathcal{M}.\mathsf{Init}$ and resorting to the secret cryptographic material.

The impossibility of output extraction for [PST16] is intuitively similar to the impossibility of UC-commitments without a common reference string (CRS) [CF01]. Consider a real-world corrupt committer participant $\mathcal{A}$ and a honest receiver $\mathcal{R}$. The simulation proof of this protocol requires for the simulator $\mathcal{S}$ to present $\mathcal{A}$ with an indistinguishable transcript, which involves extracting the committed value and forwarding it to the associated global functionality $\mathcal{F}$. However, allowing for the simulator to extract a value from $\mathcal{F}$ implies that the real-world $\mathcal{R}$ must also be able to extract that commitment, thus violating the hiding property of commitments. This is circumvented by considering a CRS as a local functionality, which can programmed by $\mathcal{S}$, allowing him to embed a trapdoor on the CRS and thus trivially extract the commitment without that being possible on the real world (since $\mathcal{R}$ does not possess such trapdoor).

This is closely translated for the approach on using $\mathcal{G}_{\mathsf{att}}$ for two-party secure computation, as the programs being loaded to the trusted hardware are extended with a trapdoor, allowing for $\mathcal{S}$ to emulate corrupt outputs without implying security vulnerabilities on the real-world protocol execution. More specifically, after a key exchange has been performed to obtain key key, the enclave's program that will compute the functionality with the two participants' inputs $\mathsf{F}(\mathsf{inp}_0, \mathsf{inp}_1)$ can either produce a signed legitimate output, or a signed output forgery. More specifically, the enclave's program is extended with a backdoor parameter v in its execution, as follows

$\mathsf{compute}(\mathsf{ct}, v)$:

$\mathsf{inp}_1 \leftarrow \mathsf{AE.Dec}(\mathsf{key}, \mathsf{ct})$ // private input is received and decrypted

If $v \neq \perp$: // checks for forgery flag

Return v // if $\perp$ return forged output

Else: Return $\mathsf{F}(\mathsf{inp}_0, \mathsf{inp}_1)$ // otherwise, produce legitimate output

this has no impact in the real-world, as honest participants will always run with $v = \perp$, and enables $\mathcal{S}$ to specifically request a forgery for the corrupt output provided by the ideal functionality. This mechanism can be extended to the MPC setting analogously, where outputs for corrupt participants can be programmed given the respective participant key.

Indeed, Pass et al. clearly identify this difference in design, where they declare that considering a local $\mathcal{G}_{\mathsf{att}}$ functionality (the research direction taken in this thesis) implies that a fresh initialization (i.e. $\mathcal{M}.\mathsf{Init}$) must be performed before all protocol executions, if one wants to claim composability. In this regard, note that mechanisms such as the TrustZone key establishment and update described in Section 4.2 allows for the algorithm that instantiates $\mathcal{M}.\mathsf{Init}$ to be run again, and could facilitate the need for repeated hardware initializations in real-world deployments. Their paper also demonstrates how composability considering a global functionality leads to the development of less intuitive protocols for secure computation. The author considers this work to nicely complement the results presented in this thesis, especially as a demonstration of the advantages and limitations of considering different abstractions for the behaviour of IEE-enabling hardware.

Models and definitions

The models presented in this thesis consider $n + 1$ participants $P_1, \ldots, P_n, \mathcal{M}$, where any number of parties can be statically corrupted, except $\mathcal{M}$, which is considered incorruptible (the designated tamper-proof hardware assumption). Note that, unlike Katz [Kat07], $\mathcal{M}$ is viewed as an honest participant, rather than an ideal functionality in some hybrid world; this is because in real-world scenarios the communication with $\mathcal{M}$ is always controlled by the adversary. Furthermore, it is necessary to rely upon the PKI set up assumption that permits protocol participants $P_1, \ldots, P_n$ to authenticate each others' public parameters and also those of the special participant $\mathcal{M}$. With these restrictions, the proposed formalization is essentially a code-based presentation of the UC model in the "dummy adversary" setting, which is an equivalent setting to general UC (see Section 4.4.1 in [Can01] for more details). The security adversary $\mathcal{A}$ combines both the environment and the dummy adversary; it can add inputs and retrieve outputs from honest parties adaptively (after statically defining the number of corrupt parties); and it controls the entire communications between all parties with the Send, Load and Run oracles. The associated simulator in the ideal world is able to control the scheduling of the ideal functionality; it obtains leakage of honest inputs and outputs (leakage here is parametrizable, but only input/output lengths are considered, for simplicity); and it can control the inputs and see the outputs of corrupt parties when interacting with the ideal functionality.

In what follows, it is shown how the proposed view of ideal functionalities as transition functions permits expressing general reactive functionalities studied in other works relying on the UC framework. An interesting direction for future work is to relate the described general approach with the systematic methodology proposed by Garay, Kiayias and Zhou [GKZ08a, GKZ08b] for the specification of ideal functionalities for the Universal Composability framework. In particular, the presented work deviates slightly from the notion of canonical functionalities specified in [GKZ08a, GKZ08b], since these models factor out from

the functionality transition function all aspects related to party corruption and leakage, but the insights gained on how to specify sound functionalities carry over to this setting.

Functionalities

Recasting ideal functionalities expressed in the UC style amounts to describing a transition function that emulates the same behaviour. Three examples are now provided: commitments, oblivious transfer and secure function evaluation.

The commitment [CF01] functionality $\mathcal{F}_{\mathsf{COM}}$ is shown in Figure 8.15. It permits a single commitment operation between two parties, and it operates in three stages: first, it expects to receive some commit value b from some party id_S to some party id_R; from that point on, in stage 1, it allows party id_R to check that some value has been committed without revealing it. Once the open command is received from id_A, the functionality enters stage 2 in which the committed value b can be provided to party id_R. Leakage functions reveal the involved identities and the lengths of inputs and outputs.

The oblivious transfer [CLOS02] functionality $\mathcal{F}_{\mathsf{OT}}$ is shown in Figure 8.16. Again, it permits oblivious transfer from some party id_S to some party id_R, and is structured in three stages. In the first stage it expects to receive a list of messages list of arbitrary size (the length may be leaked by the protocol). The sender specifies the identities of the sending and receiving parties. In the second stage, it accepts an index idx from id_R and it returns the corresponding list element. A single reading is allowed, which puts the functionality in a non responsive stage 2.

Finally, the secure function evaluation functionality $\mathcal{F}_{\mathsf{SFE}}$ for a function f with n inputs and n outputs is given in Figure 8.17. The functionality has two stages. In the first stage it accepts one input from each party. Once these are all collected, it computes f on these values and enters stage 2, where each party can collect its corresponding output. Leakage reveals input and output lengths.

Observe that, contrary to [CF01, CLOS02], the syntax defined in this thesis explicitly distinguishes the functional aspect (F) from the leakage associated with its execution ($\mathsf{Lin}, \mathsf{Lout}$). Many similarities can also be identified among the provided examples, which suggests the construction of these functionalities can be done in a systematic way, such as the explicit management of states in $\mathsf{st}_\mathsf{F}.\mathsf{st}$, and the leakage of length associated with sensitive input/output values. A rigorous method for converting UC-flavoured ideal functionality descriptions into this syntax is suggested as future work.

$\mathbf{G2}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$(\mathsf{prms}, \mathsf{sk}) \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
$\mathsf{hdl} \leftarrow 0$
$\mathsf{fake} \leftarrow [\,]$
$(\mathsf{st}_\mathcal{A}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
For $\mathsf{id} \in [1..k]$:
  $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, \mathsf{st}_{\mathsf{id}}.\mathsf{pub}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
For $\mathsf{id} \in [k+1..n]$:
  $(\mathsf{st}_\mathcal{A}, \mathsf{pub}_{\mathsf{id}}) \leftarrow\$ \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
For $\mathsf{id} \in [1..k]$:
  $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{ListOut} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} \leftarrow 0$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow 0$; $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow 1$; $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n} ; \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
  $L \leftarrow \{(p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_{\mathsf{id}}.\mathsf{id})\}$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_V \leftarrow (P, L)$
  $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{st}_\mathcal{A})$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$:
$(P, \mathsf{seq}, \mathsf{st}_{\mathsf{hdl}}) \leftarrow \mathsf{HdlList}[\mathsf{hdl}]$
If $P = P^*$: // The agreed protocol.
  If $(p, (\mathsf{id}, \epsilon)) \notin L$: Return $\perp$
  If $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{stage} = 0$:
    $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
    $\mathsf{msg}' \leftarrow\$ P^*[\mathsf{st}_{\mathsf{hdl}}, \mathsf{sk}](\mathsf{id}, \mathsf{msg})$
    If $\mathsf{st}_{\mathsf{id}} \neq \epsilon \wedge \mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{key} \notin \mathsf{fake} \wedge$
        $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
      $\mathsf{key}^* \leftarrow\$ \{0,1\}^{1^\lambda}$
      $\mathsf{st.fake} \leftarrow (\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  Else If $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{stage} = 1$:
    $(\mathsf{seq}_{\mathsf{in}}, \mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
    If $(\mathsf{seq}[\mathsf{id}] \neq \mathsf{seq}_{\mathsf{in}})$: Return $\perp$
    If $\mathsf{st}_{\mathsf{id}} \neq \epsilon$: // Honest participant
      $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), \mathsf{in})$
      If $\mathsf{msg}' = (\mathsf{seq}[\mathsf{id}], \mathsf{out}')$:
        $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{InList}[\mathsf{seq}[\mathsf{id}]])$
        $\mathsf{st}_{\mathsf{id}}.\mathsf{OutList}[\mathsf{seq}[\mathsf{id}] + 1] \leftarrow \mathsf{out}$
        $\mathsf{msg}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}]),$
            $(\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$
    Else: // Corrupt participant
      $\mathsf{in}^* \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
      $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_\mathsf{F}](\mathsf{id}, \mathsf{in})$
      $\mathsf{msg}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}))$
    $\mathsf{seq}[\mathsf{id}] \leftarrow \mathsf{seq}[\mathsf{id}] + 2$
Else: // Any other program on $\mathcal{M}$.
  $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
  $\mathsf{msg}' \leftarrow\$ P^*[\mathsf{st}_{\mathsf{hdl}}, \mathsf{st.sk}](\mathsf{id}, \mathsf{msg})$
$\mathsf{HdlList}[\mathsf{hdl}] \leftarrow (P, \mathsf{seq}, \mathsf{st}_{\mathsf{hdl}})$
Return msg'

Oracle $\mathsf{Load}(P)$:
$\mathsf{hdl} \leftarrow \mathsf{hdl} + 1$
For $i \in L$: $\mathsf{seq}[i] \leftarrow 0$
$\mathsf{HdlList} \leftarrow (\mathsf{hdl}, \mathsf{seq}, \epsilon)$
Return hdl

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{msg})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 0$:
  $(i, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_{\mathsf{id}}.\mathsf{prms}, (p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), \mathsf{in}_{\mathsf{last}},$
      $\mathsf{msg}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, i)$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{st.fake} \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  If $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{st}_{\mathsf{KE}}.\delta) = \mathsf{accept}$: Then $\mathsf{stage} \leftarrow 1$
  $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
  Return msg'
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 1$:
  If $\mathsf{msg} = \epsilon$:
    $\mathsf{in} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{InList}[\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}] \leftarrow \mathsf{in}$
    $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \{0\}^{|\mathsf{in}|}))$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} + 2$
    $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
    Return msg'
  Else:
    $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), \mathsf{msg})$
    If $\mathsf{msg}' = (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}, \mathsf{out}')$:
      $\mathsf{st}_{\mathsf{id}}.\mathsf{ListOut} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{OutList}[\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}] : \mathsf{st}_{\mathsf{id}}.\mathsf{ListOut}$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} + 2$
      $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)$
      Return msg'
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$(\mathsf{out}_1, \ldots, \mathsf{out}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListOut}$
Return $\mathsf{out}_1 \,||\, \ldots \,||\, \mathsf{out}_i$

Figure 8.13: Second hop of the proof.

$\mathbf{G3}_{\mathcal{F},\pi,\mathcal{A}}(1^\lambda)$:
$(n, \mathsf{F}, \mathsf{Lin}, \mathsf{Lout}) \leftarrow \mathcal{F}$
$(\mathsf{prms}, \mathsf{sk}) \leftarrow\$ \mathsf{SMInit}(1^\lambda)$
$\mathsf{hdl} \leftarrow 0$
$\mathsf{fake} \leftarrow [\,]$
$\mathsf{forgeAuth} \leftarrow \mathsf{F}$
$\mathsf{authList} \leftarrow [\,]$
$(\mathsf{st}_{\mathcal{A}}, k) \leftarrow\$ \mathcal{A}(\mathsf{prms})$
For $\mathsf{id} \in [1..k]$:
  $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, \mathsf{st}_{\mathsf{id}}.\mathsf{pub}) \leftarrow \mathsf{Setup}_{\mathsf{KE}}(1^\lambda, \mathsf{id})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_k)$
For $\mathsf{id} \in [k+1..n]$:
  $(\mathsf{st}_{\mathcal{A}}, \mathsf{pub}_{\mathsf{id}}) \leftarrow\$ \mathcal{A}(\mathsf{st}_{\mathcal{A}}, \mathsf{id}, \mathsf{Pub})$
$\mathsf{Pub} \leftarrow (\mathsf{pub}_1, ..., \mathsf{pub}_n)$
For $\mathsf{id} \in [1..k]$:
  $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{ListOut} \leftarrow [\,]$; $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} \leftarrow 0$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow 0$; $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow 1$; $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow \epsilon$
  $(\mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}}) \leftarrow \mathsf{Pub}$
  $P \leftarrow \langle\, \langle \mathsf{Rem}^1_{\mathsf{KE}}, \ldots, \mathsf{Rem}^n_{\mathsf{KE}} \rangle_{1,\ldots,n} ; \mathsf{Box}\langle \mathcal{F}, \Lambda \rangle \,\rangle_{\phi_{\mathsf{key}},p,q}$
  $L \leftarrow \{(p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), (q, \mathsf{st}_{\mathsf{id}}.\mathsf{id})\}$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_V \leftarrow (P, L)$
  $P^* \leftarrow \mathsf{LAC.Compile}(\mathsf{prms}, P, L)$
$b \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{st}_{\mathcal{A}})$
If $\mathsf{forgeAuth} = \mathsf{T}$: $b \leftarrow\$ \{0,1\}$

Oracle $\mathsf{Run}(\mathsf{hdl}, l, \mathsf{msg})$:
$(P, \mathsf{seq}, \mathsf{st}_{\mathsf{hdl}}) \leftarrow \mathsf{HdlList}[\mathsf{hdl}]$
If $P = P^*$: // The agreed protocol.
  If $(p, (\mathsf{id}, \epsilon)) \notin L$: Return $\perp$
  If $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{stage} = 0$:
    $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
    $\mathsf{msg}' \leftarrow\$ P^*[\mathsf{st}_{\mathsf{hdl}}, \mathsf{sk}](\mathsf{id}, \mathsf{msg})$
    If $\mathsf{st}_{\mathsf{id}} \neq \epsilon \wedge \mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{key} \notin \mathsf{fake} \wedge$
        $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
      $\mathsf{key}^* \leftarrow\$ \{0,1\}^{1^\lambda}$
      $\mathsf{st.fake} \leftarrow (\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  Else If $\mathsf{st}_{\mathsf{hdl}}[\mathsf{id}].\mathsf{stage} = 1$:
    $(\mathsf{seq}_{\mathsf{in}}, \mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
    If $(\mathsf{seq}[\mathsf{id}] \neq \mathsf{seq}_{\mathsf{in}})$: Return $\perp$
    If $\mathsf{st}_{\mathsf{id}} \neq \epsilon$: // Honest participant
      If $\mathsf{msg}' = (\mathsf{seq}[\mathsf{id}], \mathsf{out}')$:
        $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_{\mathsf{F}}](\mathsf{id}, \mathsf{InList}[\mathsf{seq}[\mathsf{id}]])$
        $\mathsf{st}_{\mathsf{id}}.\mathsf{OutList}[\mathsf{seq}[\mathsf{id}] + 1] \leftarrow \mathsf{out}$
        $\mathsf{msg}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}]),$
            $(\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|}))$
        $\mathsf{authList} \leftarrow (\mathsf{fake}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}]),$
            $(\mathsf{seq}[\mathsf{id}] + 1, \{0\}^{|\mathsf{out}|})) : \mathsf{authList}$
    Else: // Corrupt participant
      $\mathsf{in}^* \leftarrow\$ \Lambda.\mathsf{Dec}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{in}))$
      $\mathsf{out} \leftarrow \mathsf{F}[\mathsf{st}_{\mathsf{F}}](\mathsf{id}, \mathsf{in})$
      $\mathsf{msg}' \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}], (\mathsf{seq}[\mathsf{id}] + 1, \mathsf{out}))$
    $\mathsf{seq}[\mathsf{id}] \leftarrow \mathsf{seq}[\mathsf{id}] + 2$
Else: // Any other program on $\mathcal{M}$.
  $(\mathsf{id}, \mathsf{in}) \leftarrow \mathsf{msg}$
  $\mathsf{msg}' \leftarrow\$ P^*[\mathsf{st}_{\mathsf{hdl}}, \mathsf{st.sk}](\mathsf{id}, \mathsf{msg})$
$\mathsf{HdlList}[\mathsf{hdl}] \leftarrow (P, \mathsf{seq}, \mathsf{st}_{\mathsf{hdl}})$
Return msg'

Oracle $\mathsf{Load}(P)$:
$\mathsf{hdl} \leftarrow \mathsf{hdl} + 1$
For $i \in L$: $\mathsf{seq}[i] \leftarrow 0$
$\mathsf{HdlList} \leftarrow (\mathsf{hdl}, \mathsf{seq}, \epsilon)$
Return hdl

Oracle $\mathsf{SetInput}(\mathsf{in}, \mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$\mathsf{st}_{\mathsf{id}}.\mathsf{InList} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList} + [\mathsf{in}]$

Oracle $\mathsf{Send}(\mathsf{id}, \mathsf{msg})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 0$:
  $(i, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_V) \leftarrow \mathsf{LAC.Verify}(\mathsf{st}_{\mathsf{id}}.\mathsf{prms}, (p, (\mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)), \mathsf{in}_{\mathsf{last}}, \mathsf{msg},$
      $\mathsf{msg}, \mathsf{st}_V)$
  If $i = \perp$: Return $\perp$
  $(o, \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L) \leftarrow\$ \mathsf{Loc}_{\mathsf{KE}}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L, i)$
  $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
  If $\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key} \notin \mathsf{fake} \wedge \mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\delta \in \{\mathsf{derived}, \mathsf{accept}\}$:
    $\mathsf{key}^* \leftarrow\$ \{0,1\}^\lambda$
    $\mathsf{st.fake} \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}, \mathsf{key}^*) : \mathsf{fake}$
  If $(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{st}_{\mathsf{KE}}.\delta) = \mathsf{accept}$: Then $\mathsf{stage} \leftarrow 1$
  $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
  Return msg'
If $\mathsf{st}_{\mathsf{id}}.\mathsf{stage} = 1$:
  If $\mathsf{msg} = \epsilon$:
    $\mathsf{in} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{InList}[0]$
    $(\mathsf{in}_1, \ldots, \mathsf{in}_k) \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{ListIn}_{\mathsf{id}} \leftarrow (\mathsf{in}_1, \ldots, \mathsf{in}_k)$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{InList}[\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}] \leftarrow \mathsf{in}$
    $o \leftarrow\$ \Lambda.\mathsf{Enc}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \{0\}^{|\mathsf{in}|}))$
    $\mathsf{authList} \leftarrow (\mathsf{fake}(\mathsf{st}_{\mathsf{hdl}}.\mathsf{key}[\mathsf{id}]), (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}}, \{0\}^{|\mathsf{in}|})):$
        $\mathsf{authList}$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{in}_{\mathsf{last}} \leftarrow o$
    $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{in}} + 2$
    $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, o)$
    Return msg'
  Else:
    $\mathsf{msg}' \leftarrow \Lambda.\mathsf{Dec}(\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), \mathsf{msg})$
    If $\mathsf{msg}' \neq \perp \wedge (\mathsf{fake}(\mathsf{st}_{\mathsf{id}}.\mathsf{st}_L.\mathsf{key}), \mathsf{msg}) \notin \mathsf{authList}$:
      $\mathsf{forgeAuth} \leftarrow \mathsf{T}$
    If $\mathsf{msg}' = (\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}, \mathsf{out}')$:
      $\mathsf{st}_{\mathsf{id}}.\mathsf{ListOut} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{OutList}[\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}}] : \mathsf{st}_{\mathsf{id}}.\mathsf{ListOut}$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} \leftarrow \mathsf{st}_{\mathsf{id}}.\mathsf{seq}_{\mathsf{out}} + 2$
      $\mathsf{st}_{\mathsf{id}}.\mathsf{out} \leftarrow \mathsf{out}'$
      $\mathsf{msg}' \leftarrow (\mathsf{st}_{\mathsf{id}}.\mathsf{stage}, \mathsf{st}_{\mathsf{id}}.\mathsf{id}, \epsilon)$
      Return msg'
Else: Return $\perp$

Oracle $\mathsf{GetOutput}(\mathsf{id})$:
If $\mathsf{id} \notin [1..k]$ Return $\perp$
$(\mathsf{out}_1, \ldots, \mathsf{out}_k) \leftarrow \mathsf{ListOut}_{\mathsf{id}}$
Return $\mathsf{out}_1 \,||\, \ldots \,||\, \mathsf{out}_i$

Figure 8.14: Third hop of the proof.

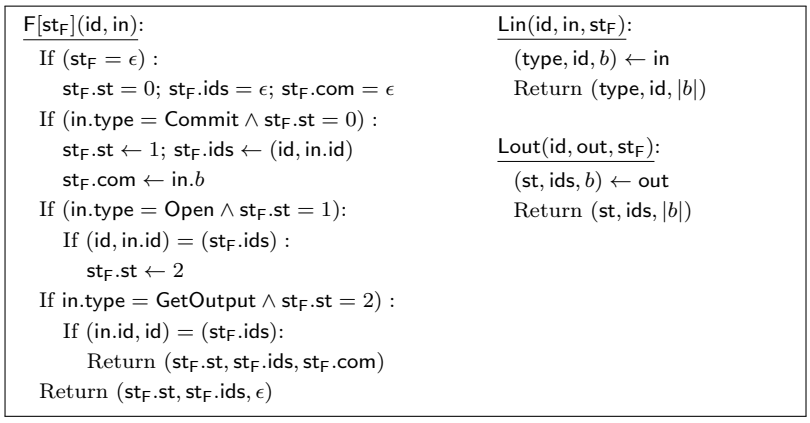


Figure 8.15: Description of functionality $\mathcal{F}_{\mathsf{COM}}$.

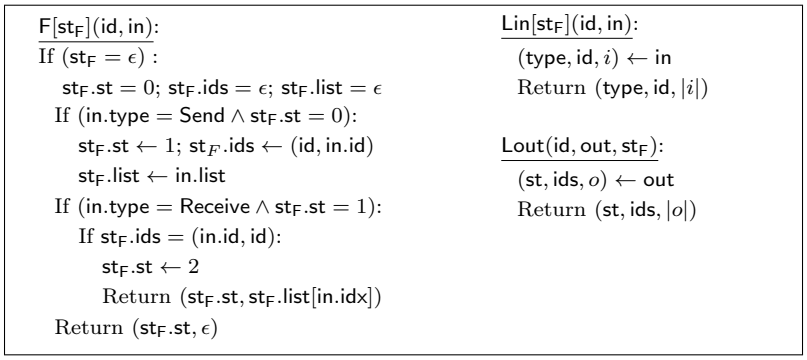


Figure 8.16: Description of functionality $\mathcal{F}_{\mathsf{OT}}$.

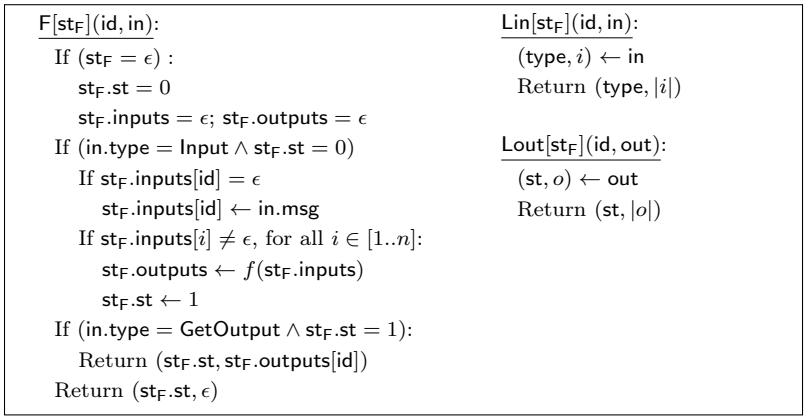


Figure 8.17: Description of functionality $\mathcal{F}_{\mathsf{SFE}}$.

Chapter 9

Experimental results

In Section 8.2, we proposed an implementation of an MPC protocol relying on trusted hardware technology. On this chapter, we present implementations for the proposed protocol in Intel's SGX, overview their vulnerabilities against side-channel attacks, and provide experimental measurements allowing for conclusions to be drawn regarding protocol feasibility. This is accompanied by similar experimental runs of the same operations on a software-based MPC framework, to be regarded as a baseline for the proposed approach.

9.1 Methodology and micro-benchmarks

Experimentation presents two implementations of the proposed MPC protocol—sgx-mpc-mbed and sgx-mpc-nacl— which differ in the underlying cryptography: sgx-mpc-mbed relies on the mbed TLS (formerly PolarSSL) library and sgx-mpc-nacl which relies on the NaCl. Furthermore, sgx-mpc-mbed uses standard RSA technology for the key exchange stage, and an AES128-CTR and HMAC-SHA256 Encrypt-Then-Mac construction for authenticated encryption, whereas sgx-mpc-nacl uses elliptic-curves both for key exchange (Diffie-Hellman) and digital signatures, and a combination of the Salsa20 and Poly1305 encryption and authentication schemes [BLS12] for authenticated encryption.

Both implementations rely on Intel's Software Development Kit (SDK) for dealing with the SGX low-level operations. These include loading a piece of code into an IEE (the Load abstraction), calling a top-level function within the IEE (the Run abstraction), and constructing an attested message (first getting a MAC'ed message within the IEE, and then using the quoting enclave to convert it into a digital signature). Furthermore, both implementations build on top of this cryptographic underpinning and share the structure in Figure 9.1. They employ the LAC scheme proposed in Section 5.3, and include wrappers that match the abstractions of digital signatures and authenticated encryption. These are then used to construct the secure bootstrapping protocol (AttKE) that enables each party to

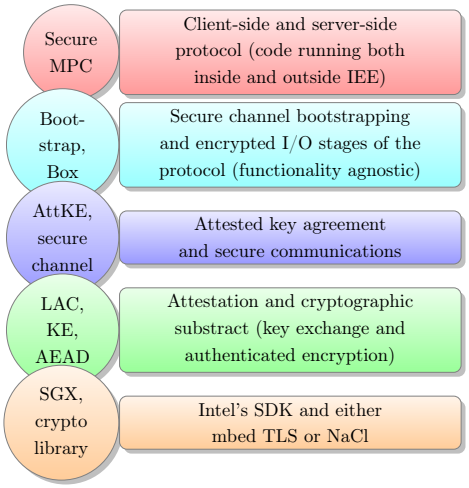


Figure 9.1: Bird's eye view of the implementations.

establish an independent secret key, and a secure channel that uses this key to communicate with the refined Box construction in Figure 8.6 running inside the enclave. Finally, both implementations of this Box are agnostic of the functionality that should be computed by the protocol, and can be linked to arbitrary functionality implementations, provided that these comply with a simple labelled I/O interface. The top-level interface to the proposed protocol includes the code that should be run inside the IEE, the code that runs outside the IEE in the remote machine to perform the book-keeping operations and the client-side code that permits bootstrapping a secure channel and then send inputs/receive outputs from the functionality.

The evaluation was performed on a SGX-enabled platform, equipped with an Intel Core i7-6700 processor (3.4 MHz) and 8 GB DDR4 RAM, running 64 bit Ubuntu 14.04. Performing fine grained performance measurements for SGX is a challenging task due to the lack of methods to directly measure the runtime of subcomponents *inside* an enclave, leading to noisy results. Neither system calls to request time from the OS nor hardware features to directly obtain timing values, like the Time Stamp Counter of the CPU, are available within an enclave [Cor16].[1] This means all timing measurements must be performed outside the enclave, i.e., measuring the time elapsed between entering an enclave until the execution of the enclave ends. For measuring the performance of individual components inside an enclave, a differential approach was employed: the runtime of an enclave was measured with and without the component of interest. Afterwards, the execution times were compared to determine the fraction of the runtime associated with the evaluated component. However, these methods cause the measurements to be noisy, e.g., due to interruption of the enclave execution because of system interrupts. To eliminate the noise, or at least minimize its effect,

[1]The time stamp counter will be available in SGX version 2 [Cor16], however, SGX2 hardware is not publicly available at the time of development.

each measurement was repeated 100 times and only the average value was reported.

Micro-Benchmarks

The first results are protocol agnostic, and show the SGX-specific inherent overhead of creating an enclave, invoking an enclave, generating MAC-based reports and signature generation by the quoting enclave (QE). Table 9.1 lists the runtime of those components with fixed run times. The invocation of an enclave cannot be measured on its own. As such, the time to enter an enclave and immediately returning was measured, specified as *SGX context switch* in Table 9.1. However, this measurement comprises both the time required for entering the enclave, as well as the time required for leaving the enclave.

Component	**Time** (ms)
SGX context switch	0.0054
MAC-based report (`ERPORT`)	0.2887
QE signatur	23.930

Table 9.1: SGX Micro-Benchmarks

The creation time of an SGX enclave is dependent on its size. The initial enclave state must be copied into the enclave memory area; the execution time of this operation is, as would be expected, linear in the size of the enclave. Additionally, the initial state of the enclave is "measured" (i.e., hashed with SHA-256) during enclave creation. Again, this operation takes an amount of time that is linear in the size of the enclave.[2] The creation time of an enclave was measured to be $202.55\mu s + S \cdot 0.72\mu s$ with enclave size S in `KB`.

9.2 Side channels and resilience against timing attacks

Section 4.4 references several works pointing out that IEE-enabled systems such as Intel's SGX do not offer more protection against side-channel attacks than traditional microprocessors. This is a relevant concern, since the IEE trust model adopted throughout this work admits that the code outside IEEs is potentially malicious and that the machine is under the control of an untrusted party. In this respect, there are two aspects to this problem that should be considered separately. The first aspect is related to the production of the IEE-enabled hardware/firmware itself and the protection of the long-term secrets that are used by the attestation security module. If the computations performed by the attestation infrastructure itself are vulnerable to side-channel attacks, then there is nothing that can be

[2]SGX does not require the measurement of the entire initial memory state of an enclave. However, the SDK's default behaviour is to measure the entire enclave (code and data) which is what is required for this MPC protocol.

done at the protocol design/implementation level. This aspect of trust is within the remit of the equipment manufacturers.

An orthogonal issue is the possibility that software running inside an IEE leaks part of its state or short-term secrets via side channels. Here one should distinguish between software observations and hardware/physical observations. In the former, software co-located in the machine observes timing channels based on memory access patterns, control flow, branch prediction, cache-based based attacks [CD16], page-fault side channels [XCP15], etc. Protection against this type of side-channel attacks has been widely studied in the practical crypto community, where a consensus exists that writing so-called *constant-time* software is the most effective countermeasure [Ber05, Lan13]. As mentioned above, constant-time software has the property that the entire sequence of memory addresses (in both data and code memory) accessed by a program can be predicted in advance from public inputs, e.g., the length of messages. When it comes to hardware/physical side-channel attacks such as those relying on temperature measurements, power analysis, or electromagnetic radiation, the effectiveness of software countermeasures is very limited, and improving hardware defences again implies obtaining additional guarantees from the equipment manufacturer.

For clarity, recall that the two implementations differ in the way they deal with timing channels significantly: while sgx-mpc-nacl enforces a strict constant-time policy that is consistent with the IEE trust model, sgx-mpc-mbed relies on a standard TLS implementation that was not designed to deal with attacks by an adversary co-located in the same machine.

Countermeasures against timing attacks

To assess the practical impact of assuming a stronger attack model in which the remote party may launch timing attacks against implementations running inside enclaves, the two implementations of were compared. More specifically, the focus was to evaluate the performance of the individual steps in the protocol, namely the key exchange between IEE with an input party, and the Box component which decrypts all inputs and encrypts all outputs in stage two of the protocol.[3]

Table 9.2 lists the measurements for the individual components of the two proposed implementations, sgx-mpc-nacl and sgx-mpc-mbed. The reported time comprises the key exchange with *one* input party, and so the overhead will accumulate linearly with an increasing number of input parties. Interestingly, the Box components of the sgx-mpc-nacl implementation is faster than in the sgx-mpc-mbed implementation, which shows that highly optimized constant-time software (deploying cryptographic primitives designed specifically for this purpose) can be faster than their non-constant-time counterparts, as argued for

[3]Note that, when such an attacker is considered, not only the cryptographic components must be implemented following the constant-time coding policies, but also the code that implements the functionality itself (!), and so an additional penalty may be payed in addition to the overhead reported here.

example in [BLS12]. In the key exchange stage, the better performance of RSA public-key operations (encryption and signature verification) gives sgx-mpc-mbed and advantage.

Component	**sgx-mpc-mbed** (ms)	**sgx-mpc-nacl** (ms)
Key exchange (Stage 1)	35.17	127.6
Box (Stage 2)	0.036	0.012

Table 9.2: LAC Components Benchmarks

Throughout the remainder of the Chapter, protocol comparison with other solutions will report to the evaluation results for constant-time implementation sgx-mpc-nacl. The overhead of sgx-mpc-nacl over sgx-mpc-mbed occurs mostly in the key exchange phase, this means that from the execution times of sgx-mpc-mbed can reasonably be inferred from the sgx-mpc-nacl results.

9.3 Comparison with state-of-the-art MPC protocols

The proposed implementation is compared with measurements performed using the ABY framework, described in Section 3.2. By choosing ABY for comparison, one was able to evaluate it on the same platform used for assessing the protocol, therefore avoiding differences due to performance disparities of heterogeneous evaluation platforms. Although it is specific to the two-party secure computation setting, ABY is representative of state-of-the-art MPC implementations and it is expected for results for other frameworks such as Sharemind and SPDZ, described in the same Section, to lead to similar conclusions. Indeed, the crux of the aforementioned performance gains resides in the fact that the proposed solution does not require encoding the computation in circuit form, which happens in one form or another for all of the mentioned protocols.[4]

Evaluation was compared with four different secure two-party computation use cases: minimum, Hamming distance, private set intersection, and AES. Similarly to the proposed approach, the ABY protocol also has two phases: a *preparation phase* and an *online phase*. The preparation phase comprises the key exchange between the input parties by means of oblivious transfer, and the generation of the garbled circuit representing the desired function. In the online phase the GC gets evaluated and the result are send back to the output party. In this thesis' MPC protocol, the preparation phase is used to establish a secure channel between the IEE and the input parties. The online phase of the protocol comprises the decryption of inputs in the Box component, the evaluation of the payload function, and the encryption of the results, again by the Box component.

[4]It should also be noted that ABY assumes a passive adversary, which is weaker than the one considered in this thesis; and nevertheless, performance gains are significant.

Phase	**ABY**	**sgx-mpc-nacl**
Preparation (ms)	196.3	127.7
Online (ms)	0.404	0.024
Total (ms)	196.7	127.7

Table 9.3: Minimum of two inputs

Determination of minimum

Table 9.3 shows the performance of the two stages of ABY and sgx-mpc-nacl for determining the minimum of two inputs, 32 *bits* each. For both phases the runtime of sgx-mpc-nacl is shorter than ABY's runtime; however the performance gains of this approach are not too noticeable for such low-demanding computations, since the preparation stage takes the large majority of the overall runtime (99.79% for ABY and 99.98% for ours).

AES

The results for comparing the secure multi-party computation of AES with ABY and with sgx-mpc-nacl are presented in Table 9.4. AES has become a standard for evaluating MPC protocols, hence, evaluation results for AES are available in a number of related works.

Phase	**ABY**	**sgx-mpc-nacl**
Preparation (ms)	197.9	115.84
Online (ms)	3.249	0.661
Total (ms)	201.1	116.50

Table 9.4: AES – 128 *bit* key and 128 *bit* block size

The most noticeable difference is in the setup, which in ABY implies a larger preparation stage for the AES, while the preparation stage for the MPC protocol proposed is always fixed to the establishment of secure channels. While this presents some performance gains, the following examples will explore increasing input sizes for computation, and will allow for a more clear display of performance gain with respect to ABY.

Hamming Distance

Next, ABY is compared with sgx-mpc-nacl for computing the Hamming Distance of two inputs. The performance was evaluated for different input sizes to demonstrate the scaling behaviour for the different solutions, as detailed in Table 9.5.

Again, the preparation phase as well as the online phase of sgx-mpc-nacl is faster, compared to ABY. However, the more noticeable difference relies on how much faster performance in

	Phase	Preparation (ms)		Online (ms)		Total (ms)	
	Protocol	**ABY**	**sgx-mpc-nacl**	**ABY**	**sgx-mpc-nacl**	**ABY**	**sgx-mpc-nacl**
Input size (bits)	*160*	196.3	115.7	0.752	0.050	197.1	117.75
	1600	196.7	115.7	1.819	0.302	198.5	116.00
	16000	201.6	115.7	13.14	2.798	214.7	118.50
	160000	226.2	115.2	144.4	27.77	370.6	142.97

Table 9.5: Hamming distance with different input sizes

ABY degrades with increasing input sizes, when compared to the proposed protocol. This is the expected outcome, since the actual computation is performed in the clear, within an IEE.

Private set intersection

Table 9.6 lists the comparative results for private set intersection. Consistently to the previous use cases, the runtime of ABY's preparation phase is increasing with the size of the input data, while sgx-mpc-nacl's preparation phase remains constant (modulo some measurement inaccuracies). Additionally, the growth in runtime of ABY's online phase is much stronger compared to sgx-mpc-nacl, for the same reasons listed previously.

	Phase	Preparation		Online		Total	
	Protocol	**ABY**	**sgx-mpc-nacl**	**ABY**	**sgx-mpc-nacl**	**ABY**	**sgx-mpc-nacl**
Set size	*100*	224.8	115.8	1.084	0.043	225.9	115.84
	1000	368.1	115.8	2.168	0.199	370.3	116.00
	10,000	1442.2	115.8	12.88	1.758	1455.1	117.56
	100,000	10,699	115.7	109.5	17.39	10,808	133.09
	1,000,000	84,097	115.7	1616.0	173.1	85,713	288.80

Table 9.6: Private set intersection with different set sizes

In comparison to ABY, the preparation phase and online phase are shorter with sgx-mpc-nacl, and consequently the overall runtime is faster as well. In general, sgx-mpc-nacl is faster for all the testing computations performed. However, the gains are considerably more noticeable when input size and computation are increased. This has the highest significance on evaluation of the private set intersection with the largest input size (1 mill.), where the proposed implementation is roughly 300 times faster.

Chapter 10

Conclusion

Emerging technological developments allow for programmers to design software to be executed within protected execution environments, with strict enforcement of resource isolation – IEEs. A major advantage of employing IEE-enabling hardware in software design comes from the fact that these systems can produce cryptographic signatures of code running within these environments, and of outputs produced by them. Having feasible hardware-based runtime guarantees allows for the design of security-critical applications that would otherwise require computationally heavy cryptographic mechanisms such as input commitment and zero-knowledge proofs. This has led to a flurry of publications making use of these hardware assumptions to enable highly efficient protocols for applications such as secure map-reduce operations [SCF+15b], Functional encryption [FVBG], private genome analysis [SAM+17], block-chain systems [MHWK16], two-party computation [GMF+16] or secure database management [FBB+17].

However, performing a provable security analysis of systems employing this novel approach of IEEs is non-trivial. This is a direct consequence of the paradigm imposed by these hardware assumptions, namely the lack of long-term uniquely identifying cryptographic material, and the requirement of loading code as a monolithic program where state is necessarily shared. Towards this goal, Attested Computation and Labelled Attested Computation are cryptographic primitives proposed to abstract the behaviour of IEE-enabling hardware, and for which one can rigorously specify the obtained security properties.

In this context, a highly efficient key-exchange protocol is proposed, considering an asymmetric setting where a participant wants to establish a secure channel with a (potentially malicious) remote machine equipped with an IEE. Combining such a key-exchange protocol with the previously specified guarantees of AC/LAC also allows for proving utility theorems where key secrecy withstands composition of key-exchange protocols with arbitrary code.

The application of these theorems enables the design of protocols for Secure Outsourced Computation, Secure Function Evaluation and Secure Multiparty Computation. A significant

benefit of developing security-critical applications taking such an approach is that all results are accompanied by specific security properties and validating cryptographic proofs. By using isolated execution environments and mechanisms for output attestation in the process of secure channel establishment, we can converge to algorithm designs that are theoretically much more efficient than their software-based counterparts. This is an intuitive result, since most of the cryptographic effort is made on the key-exchange stage (preparation) and the processing stage (online) can be done mostly in the clear within IEEs. Experimental results demonstrate the magnitude of these gains, especially when considering scalability, where the efficiency of the online phase is most relevant.

Future work

The contributions proposed in this thesis and the obtained results suggest additional research directions. The most promising are now listed.

- Exploring other real-world applications using the developed results. For instance, this approach can be used as a mechanism for secure data processing to be included on the (originally unrelated) work of [MPP+17], where a framework for employing cryptographic mechanisms for NoSQL execution is presented. This would enable for a more extensive evaluation of IEEs under the secure database processing use case, where NoSQL benchmarks can be employed for comparing IEE-based techniques and standard privacy-preserving cryptographic approaches.

- Extending theoretical MPC results to consider adaptively corrupting adversaries. The security model proposed in Chapter 8 considers adversaries behaving arbitrarily, with static corruptions (the participants corrupted by the adversary must be fixed a priori). Intuitively, the presented MPC approach should also withstand adversaries that trigger corruptions while the protocol is ongoing (adaptive corruptions), however a full rigorous proof is yet to be formalised.

- Develop side-channel countermeasures for software running within IEEs. The presented implementation of sgx-mpc-nacl makes use of the NaCl cryptographic library with constant-time algorithms, which is often considered to be the most efficient countermeasure to these attacks, however it must also be the case for the actual function running securely to run in constant-time. This motivates orthogonal work on the design of an additional constant-time verification stage before allowing for any function to be executed within the IEE.

- Implement and produce experimental results for ARM TrustZone. Despite the significant number of publications for TrustZone-enabled applications, no actual implementation of this specific flavour of key-exchange and secure computation exists in TrustZone.

One expects the results and gains to be comparable, however it would be interesting to produce experimental values for efficiency comparison, as it might provide insights regarding under which conditions each hardware can overperform its alternative.

- Explore alternative trusted hardware frameworks for protocols relying on IEEs. Recent contributions, such as Sanctum [CLD] and TrustLite [KSSV14], propose interesting alternatives to the SGX/TrustZone. In particular, it would be interesting to validate and refine our abstraction to capture the machine behaviour delimited by these approaches.

- Employ computer-aided cryptographic proof assistants to verify the presented security proofs. To further validate the theoretical results proposed, the usage of automated tools for cryptographic proof verification such as EasyCrypt [1] or ProVerif [2] would allow for a higher level or rigour in the mathematical verification of results, and thus reduce the possibility of human error.

- Develop protocols that rely on LAC to achieve the required security properties. One of the main advantages of our approach is its modularity, as the establishment of multiple secure channels may be overkill some use cases. The implementation of a protocol using LAC for applications such as a public bulletin board, where integrity and verifiability are necessary but confidentiality is not, might provide even better experimental results for execution time and scalability.

[1] `https://www.easycrypt.info/trac/`
[2] `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`

References

[AAD+16] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[ABL+04] Mikhail Atallah, Marina Bykova, Jiangtao Li, Keith Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2004.

[AF04] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 805–817. ACM, 2016.

[AGJS13] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[AL17] Na-Young Ahn and Dong Hoon Lee. Countermeasure against side-channel attack in shared memory of trustzone. *arXiv preprint arXiv:1705.08279*, 2017.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 535–548. ACM, 2013.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. *EUROCRYPT (1)*, 9056:673–701, 2015.

[AMPR14] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. Non-interactive secure computation based on cut-and-choose. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 387–404. Springer, 2014.

[And] Ross Anderson. Security engineering: A guide to building dependable distributed systems. 2001.

[ARM09] ARM ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[BBB+17] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from sgx. In *Financial Cryptography*, volume 10322. Springer, 2017.

[BCC04] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145. ACM, 2004.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography*, pages 315–333. Springer, 2013.

[BCK+14] Fabrice Benhamouda, Jan Camenisch, Stephan Krenn, Vadim Lyubashevsky, and Gregory Neven. Better zero-knowledge proofs for lattice encryption and their application to group signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 551–572. Springer, 2014.

[BCL08] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[BCLK17] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. *arXiv preprint arXiv:1701.00981*, 2017.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM, 2008.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In *Annual International Cryptology Conference*, pages 97–109. Springer, 1995.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[BGN+16] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. *arXiv preprint arXiv:1612.03744*, 2016.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[BJL12] Dan Bogdanov, Roman Jagomägis, and Sven Laur. A universal toolkit for cryptographically secure privacy-preserving data mining. In *Pacific-Asia Workshop on Intelligence and Security Informatics*, pages 112–126. Springer, 2012.

[BKLP15] Fabrice Benhamouda, Stephan Krenn, Vadim Lyubashevsky, and Krzysztof Pietrzak. Efficient zero-knowledge proofs for commitments from learning with errors over rings. In *European Symposium on Research in Computer Security*, pages 305–325. Springer, 2015.

[BKLS16a] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Implementation and evaluation of an algorithm for cryptographically private principal component

analysis on genomic data. In *Proceedings of the 3rd International Workshop on Genome Privacy and Security (GenoPri'16)*, GenoPri '16, pages 1–8, 2016.

[BKLS16b] Dan Bogdanov, Liina Kamm, Swen Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. *IEEE Transactions on Dependable and Secure Computing*, 2016.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 1994.

[BL11] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *International Journal of Information Privacy, Security and Integrity 2*, 1(1):3–33, 2011.

[Bla79] George Robert Blakley. Safeguarding cryptographic keys. *Proc. of the National Computer Conference1979*, 48:313–317, 1979.

[BLLP14] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 184–198. IEEE, 2014.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing*, 13(4):850–864, 1984.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, 2017.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[BMS+15] Ferdinand Brasser, Hiva Mahmoodi, Ahmad-Reza Sadeghi, Martin Haerterich, Ágnes Kiss, Michael Stausholm, Cem Kazan, Sander Siim, Manuel Barbosa,

Bernardo Portela, Meilof Veeningen, Niels de Vreede, Antonio Zilli, and Stelvio Cimato. PRACTICE Deliverable D12.2: adversary, trust, communication and system models, 2015. Available from `http://www.practice-project.eu`.

[BNB+15] Niklas Buescher, Peter Nordholt, Dan Bogdanov, Roman Jagomägis, Jaak Randmets, José Bacelar Almeida, Bernardo Portela, and Hugo Pacheco. PRACTICE Deliverable D12.3: formal verification requirements, 2015. Available from `http://www.practice-project.eu`.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, 2013.

[BoSV16] Dan Bogdanov, Marko J oemets, Sander Siim, and Meril Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Technical Report T-4-24, Cybernetica AS, `http://research.cyber.ee/`., 2016.

[BPS+16] Manuel Barbosa, Bernardo Portela, Berry Schoenmakers, Niels de Vreede, Guilaume Scerri, and Bogdan Warinschi. PRACTICE Deliverable D12.3: efficient verifiability and precise specification of secure computation functionalities, 2016. Available from `http://www.practice-project.eu`.

[BPSW16] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *EuroS&P*, pages 245–260. IEEE, 2016.

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. *Ucsd Cse*, 207:207, 2005.

[Bra93] Stefan Brands. Untraceable off-line cash in wallet with observers. In *Annual International Cryptology Conference*, pages 302–318. Springer, 1993.

[Bra13] Luís TAN Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 441–463. Springer, 2013.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[BSCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 287–304. IEEE, 2015.

[BSCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *International Cryptology Conference*, pages 276–294. Springer, 2014.

[BSCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796, 2014.

[BSWW13] Christina Brzuska, Nigel P. Smart, Bogdan Warinschi, and Gaven J. Watson. An analysis of the EMV channel establishment protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 373–386. ACM, 2013.

[BTS+17a] Dan Bogdanov, Karl Tarbe, Ville Sokk, João Paulo, Francisco Maia, Tiago Oliveira, Rogério Pontes, and Bernardo Portela. SAFECLOUD Deliverable D3.5: secret-sharing and order-preserving encryption based private computation, 2017. Available from `http://www.safecloud-project.eu/`.

[BTS+17b] Dan Bogdanov, Karl Tarbe, Ville Sokk, João Paulo, Francisco Maia, and Bernardo Portela. SAFECLOUD Deliverable D3.4: non-elastic restricted secure sql engine, 2017. Available from `http://www.safecloud-project.eu/`.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[CAGAA+13] R An C Anetti, Ju An A G Ar Ay, et al. Advances in cryptology-crypto 2013, 2013.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[CC+08] Jan Camenisch, Rafik Chaabouni, et al. Efficient protocols for set membership and range proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 234–252. Springer, 2008.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *CRYPTO (2)*, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[CD] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained, 2016.

[CDN05] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation, an introduction. *Contemporary cryptology*, pages 41–87, 2005.

[CDVdG87] David Chaum, Ivan B Damgård, and Jeroen Van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 87–119. Springer, 1987.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Transactions on Emerging Telecommunications Technologies*, 8(5):481–490, 1997.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. *Advances in Cryptology—EUROCRYPT 2001*, pages 453–474, 2001.

[CLD] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. Technical report, Cryptology ePrint Archive, Report 2015/564, 201 5. http://eprint. iacr. org.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 494–503. ACM, 2002.

[Cor16] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`, 2016.

[CP92] David Chaum and Torben P Pedersen. Wallet databases with observers. In *Crypto*, volume 92, pages 89–105. Springer, 1992.

[CP93] Ronald JF Cramer and Torben P Pedersen. Improved privacy in wallets with observers. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 329–343. Springer, 1993.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*, pages 119–136. Springer, 2001.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[DM15] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[DNNR16] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. Gate-scrambling revisited-or: The tinytable protocol for 2-party secure computation. *IACR Cryptology ePrint Archive*, 2016:695, 2016.

[DPR+13] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[FBB+17] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with sgx. *arXiv preprint arXiv:1703.04583*, 2017.

[FJN14] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. *Faster Maliciously Secure Two-Party Computation Using the GPU*, pages 358–379. Springer International Publishing, Cham, 2014.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 225–255. Springer, 2017.

[FMSW11] Atanas Filyanov, Jonathan M McCuney, Ahmad-Reza Sadeghiz, and Marcel Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 1–12. IEEE, 2011.

[FNRT14] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[FVBG] Ben A Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using intel sgx.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.

[GGH+16] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.

[GH11] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–148. Springer, 2011.

[GK96] Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for np. *Journal of Cryptology*, 9(3):167–189, 1996.

[GKZ08a] Juan A. Garay, Aggelos Kiayias, and Hong-Sheng Zhou. A framework for the sound specification of cryptographic tasks. *IACR Cryptology ePrint Archive*, 2008:132, 2008.

[GKZ08b] Juan A. Garay, Aggelos Kiayias, and Hong-Sheng Zhou. Sound and fine-grain specification of ideal functionalities. In *Theoretical Foundations of Practical Information Security*, 2008.

[GLX+17] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. *arXiv preprint arXiv:1704.05600*, 2017.

[GMF+16] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor. Using intel software guard extensions for efficient two-party secure function evaluation. In *Proceedings of the 2016 FC Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2016.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 174–187. IEEE, 1986.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[Gra09] David Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[GT07] He Ge and Stephen R. Tate. A direct anonymous attestation scheme for embedded devices. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, volume 4450 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2007.

[GVJ14] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 132–150. Springer, 2011.

[HLP+13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, page 11. ACM, 2013.

[hSas11] Chih hao Shen and a. shelat. Two-output secure computation with malicious adversaries. In *Eurocrypt 2011*, 2011.

[HSS+10] Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, Immo Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*, pages 145–161. Springer, 2003.

[Jag10] Roman Jagomägis. Secrec: a privacy-aware programming language with applications in data mining. *Master's thesis, University of Tartu*, 2010.

[JTM+] Yaoqi Jia, Shruti Tople, Tarik Moataz, Deli Gong, Prateek Saxena, and Zhenkai Liang. Robust synchronous p2p primitives using sgx enclaves.

[Kat07] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2007.

[KJ01] Paul C Kocher and Joshua M Jaffe. Secure modular exponentiation with leak minimization for smartcards and other cryptosystems, October 2 2001. US Patent 6,298,442.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for xor gates that beats free-xor. In *International Cryptology Conference*, pages 440–457. Springer, 2014.

[KOR+17] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of aes and des using lookup tables. Cryptology ePrint Archive, Report 2017/378, 2017. `http://eprint.iacr.org/2017/378`.

[KRW17] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and communication-efficient, constant-round, secure two-party computation. *IACR Cryptology ePrint Archive*, 2017:30, 2017.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. *Automata, Languages and Programming*, pages 486–498, 2008.

[KSH+15] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *International Conference on Cryptology and Network Security*, pages 1–20. Springer, 2009.

[KSS10] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. From dust to dawn: Practically efficient two-party secure function evaluation protocols and their modular design. *IACR Cryptology ePrint Archive*, 2010:79, 2010.

[KSS13a] Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 549–560. ACM, 2013.

[KSS13b] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *International Conference on Applied Cryptography and Network Security*, pages 566–584. Springer, 2014.

[KSSV14] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, page 10. ACM, 2014.

[KW15] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[KY00] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2000.

[KY03] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2003.

[Lan13] Adam Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, February 2013. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, Accessed October 25th, 2015.

[Lar14] Enrique Larraia. Extending oblivious transfer efficiently. In *International Conference on Cryptology and Information Security in Latin America*, pages 368–386. Springer, 2014.

[LEPS16] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. *arXiv preprint arXiv:1612.07766*, 2016.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, pages 549–564, 2016.

[Lin16] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology*, 29(2):456–490, 2016.

[LLXX05] Bao Li, Hongda Li, Guangwu Xu, and Haixia Xu. Efficient reduction of 1 out of n oblivious transfers in random oracle model. *IACR Cryptology ePrint Archive*, 2005:279, 2005.

[LOS+10] Allison B Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Eurocrypt*, volume 6110, pages 62–91. Springer, 2010.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 52–78. Springer, 2007.

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *International Conference on Security and Cryptography for Networks*, pages 2–20. Springer, 2008.

[LSG+16] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[LTW13] Sven Laur, Riivo Talviste, and Jan Willemson. From oblivious aes to efficient and secure database join in the multiparty setting. In *International Conference on Applied Cryptography and Network Security*, pages 84–101. Springer, 2013.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography*, volume 3958, pages 458–473. Springer, 2006.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[MHWK16] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of luck: an efficient blockchain consensus protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, page 2. ACM, 2016.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986*, 2017.

[MKD+] Sinisa Matetic, Kari Kostiainen, Aritra Dhar, David Sommer, Mansoor Ahmed, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution.

[MPP+17] Ricardo Macedo, João Paulo, Rogério Pontes, Bernardo Portela, Tiago Oliveira, Miguel Matos, and Rui Oliveira. A practical framework for privacy-preserving nosql databases. In *Symposium on Reliable Distributed Systems*, 2017.

[MZ] R. Morelos-Zaragoza. Encoder/decoder for binary bch codes in c (version 3.1). `http://www.rajivchakravorty.com/source-code/uncertainty/multimedia-sim/html/bch_8c-source.html`.

[Nao91] Moni Naor. Bit commitment using pseudorandomness. *Journal of cryptology*, 4(2):151–158, 1991.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, volume 7417, pages 681–700. Springer, 2012.

[NP00] Moni Naor and Benny Pinkas. Distributed oblivious transfer. In *ASIACRYPT*, volume 1976, pages 205–219. Springer, 2000.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[OCN+16] O Ohrimenko, CF Manuel Costa, S Nowozin, A Mehta, F Schuster, and K Vaswani. Sgx-enabled oblivious machine learning. In *Proceedings of the 25th USENIX Security Symposium (Security), Austin, TX*, 2016.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[PBT+16] Bernado Portela, Dan Bogdanov, Karl Tarbe, Reimo Rebane, Francisco Maia, João Paulo, and Rogério Pontes. SAFECLOUD Deliverable D3.1: architectural and api proposal for the secure processing stack, 2016. Available from `http://www.safecloud-project.eu/`.

[PG16] Nicolae Paladi and Christian Gehrmann. Trusdn: Bootstrapping trust in cloud network infrastructure. In *12th EAI International Conference on Security and Privacy in Communication Networks*, 2016.

[PPFF17] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure content-based routing using intel software guard extensions. *CoRR*, abs/1701.04612, 2017.

[PPM+16] Bernardo Portela, João Paulo, Francisco Maia, Rogério Pontes, Tiago Oliveira, Reimo Rebane, and Karl Tarbe. SAFECLOUD Deliverable D3.2: privacy-preserving storage and computation techniques, 2016. Available from `http://www.safecloud-project.eu/`.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 250–267. Springer, 2009.

[PST16] Rafael Pass, Elaine Shi, and Florian Tramer. Formal abstractions for attested execution secure processors. Cryptology ePrint Archive, Report 2016/1027, 2016. `http://eprint.iacr.org/2016/1027`.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Crypto*, volume 5157, pages 554–571. Springer, 2008.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 245–254. ACM, 2000.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security Symposium*, pages 297–314, 2016.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[S+13] Chih-hao Shen et al. Fast two-party secure computation with minimal assumptions. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 523–534. ACM, 2013.

[SAM+17] Md Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Shuang Wang, and Xiaoqian Jiang. Safety: Secure gwas in federated environment through a hybrid solution with intel sgx and homomorphic encryption. *arXiv preprint arXiv:1703.02577*, 2017.

[SCF+15a] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[SCF+15b] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[Sch08] Thomas Schneider. Practical secure function evaluation. In *Informatiktage*, pages 37–40, 2008.

[SCNS15] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *arXiv preprint arXiv:1506.04832*, 2015.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Sol] Integrated Silicon Solution. `http://www.alldatasheet.com/datasheet-pdf/pdf/505020/ISSI/IS61LV6416-10TL.html`.

[SRC07] Ben Smyth, Mark Ryan, and Liqun Chen. Direct anonymous attestation (DAA): ensuring privacy with corrupt administrators. In *Security and Privacy in Ad-hoc and Sensor Networks, 4th European Workshop, ESAS 2007, Cambridge, UK, July 2-3, 2007, Proceedings*, volume 4572 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2007.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 475–484. ACM, 2014.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[T+16] Riivo Talviste et al. *Applying secure multi-party computation in practice*. PhD thesis, 2016.

[TLP+16] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N Asokan. The circle game: Scalable private membership test using trusted hardware. *arXiv preprint arXiv:1606.01655*, 2016.

[WB15] Michael Walfish and Andrew J Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

[WMK17] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. Faster secure two-party computation in the single-execution setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–424. Springer, 2017.

[WRK] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. Technical report, Cryptology ePrint Archive, Report 2017/189, 2017. http://eprint. iacr. org/2017/189.

[WW17] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. *arXiv preprint arXiv:1701.01061*, 2017.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE Computer Society, 2015.

[Yao82] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[Yee94] Bennet Yee. *Using secure coprocessors*. PhD thesis, Citeseer, 1994.

[YYQ+15] Bo Yang, Kang Yang, Yu Qin, Zhenfeng Zhang, and Dengguo Feng. Daa-tz: an efficient daa scheme for mobile devices using arm trustzone. In *International Conference on Trust and Trustworthy Computing*, pages 209–227. Springer, 2015.

[YYZ+16] Bo Yang, Kang Yang, Zhenfeng Zhang, Yu Qin, and Dengguo Feng. Aep-m: Practical anonymous e-payment for mobile devices using arm trustzone and divisible e-cash. In *International Conference on Information Security*, pages 130–146. Springer, 2016.

[ZCC+16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 270–282. ACM, 2016.

[ZEE+] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert van Renesse. Rem: Resource-efficient mining for blockchains.

[ZGNM12] Zongwei Zhou, Virgil D Gligor, James Newsome, and Jonathan M McCune. Building verifiable trusted path on commodity x86 computers. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 616–630. IEEE, 2012.

[ZSS+16] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. Technical report, Cryptology ePrint Archive, Report 2016/980, 2016. http://eprint. iacr. org/2016/980, 2016.

[ZZH+14] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. Providing root of trust for arm trustzone using on-chip sram. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pages 25–36. ACM, 2014.