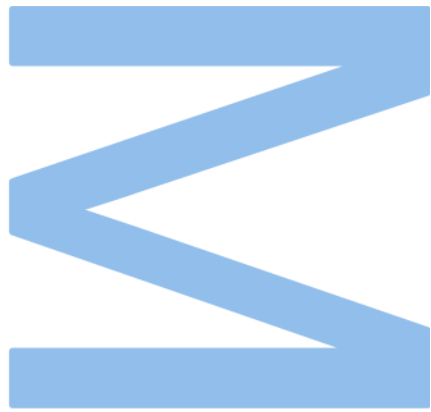
Strengthening Content-Centric Networking with Cryptographic Hardware Solutions



Leonardo Araújo Freitas
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2026

Strengthening Content-Centric Networking with Cryptographic Hardware Solutions



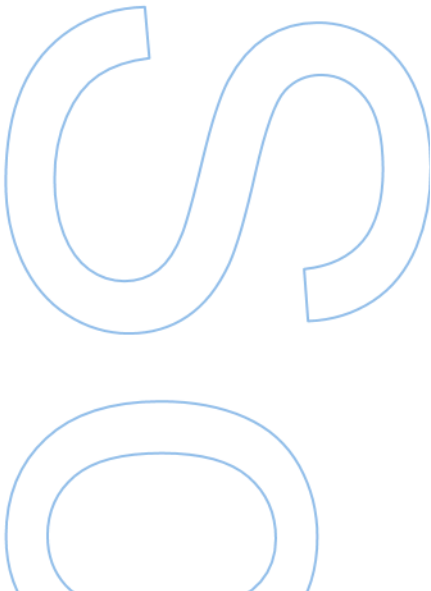
Leonardo Araújo Freitas

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2026

Supervisor
Bernardo Portela, Assistant Professor, Faculty of Sciences, University of Porto



“Finding strength through every trial, turning pain into purpose and every scar into fire.”

Imagine Dragons, inspired by *Believer*

“Long story short, I survived.”

Taylor Swift, *Long Story Short*

“They all say that it gets better.”

Olivia Rodrigo, *teenage dream*

Acknowledgements

This page was more difficult to write than the chapters.

To my Professor Bernardo Portela, thanks for the advice, the patience, and for picking me up every time I showed up confused. Much of this work would not exist if it were not for you.

To my parents. Thank you for never letting me give up, even when I wanted to. Thank you for listening to me whine about things you had no business caring about. Thank you for carrying me through the parts I could not carry myself. You share in everything I accomplish, including this thesis, and I share it with you gladly. To my mother especially, thank you for carrying the heaviest part of it.

To my friends, for the deep conversations and the silly ones, for the support when I needed it and the distraction when I needed that more. Thanks for dragging me away from my computer when I had been working on the same problem for hours, for enduring my explanations of things you never asked about, and for convincing me not to quit every time I told you I wanted to. A special thank you to Matilde Simões, who was there for all of it and somehow still tolerates me.

To all the others who pitched in, in one way or another, or were simply around while I was occupied and absentminded.

And lastly, to myself, for finishing.

Thank you!

Resumo

As redes centradas no conteúdo recuperam dados pelo nome, e não pela localização, pelo que qualquer nó que tenha uma cópia de um objeto pode responder a um pedido por esse nome. O Named Data Networking é a sua instância mais desenvolvida, e torna os encaminhadores stateful, mantendo num conjunto reduzido de tabelas o estado que suporta a cache na rede e a agregação de pedidos. Este desenho reduz a latência e o tráfego, e serve bem os consumidores móveis e a distribuição de conteúdo em larga escala, o que explica grande parte do seu interesse.

Esse mesmo estado expõe o consumidor. Duas das tabelas denunciam o seu comportamento, já que monitorizar a Content Store revela se um pedido foi servido da cache ou falhou, e monitorizar a Pending Interest Table revela os nomes em trânsito. Encriptar o conteúdo não resolve isto, porque o que escapa é o padrão de acessos às tabelas, e não o próprio conteúdo. Um encaminhador que tenha de processar pedidos sobre este estado num host não confiável precisa, então, tanto de esconder o seu estado do host como de esconder o seu padrão de acessos.

Esta dissertação responde a ambos. O encaminhador corre dentro de um enclave Intel SGX, que isola o seu processamento e o seu estado do host, e uma única árvore Path ORAM suporta a Content Store, a Pending Interest Table e o índice de nomes, de modo que cada acesso a uma tabela transfere um caminho completo da raiz à folha e o custo de um pedido deixa de revelar o seu resultado. O estado que sai do enclave é encriptado, autenticado e versionado contra reversão, e um cliente confirma que fala com o encaminhador autêntico através de atestação remota ligada ao canal de transporte.

A contribuição é um protótipo construído sobre o NFD e a biblioteca ndn-cxx, sob o Gramine, que por conveniência de implementação corre o SGX em modo de simulação, acompanhado de uma avaliação do custo desta proteção. Submetendo a Content Store oblívia do encaminhador e um armazenamento comum à mesma carga, uma consulta que um armazenamento comum responde numa fração de microssegundo passa a demorar dezenas de milissegundos e a mover dezenas de megabytes, o débito cai de milhões de operações por segundo para algumas dezenas, e a memória sobe para entre dezassete e duzentas vezes a de um dicionário comum. Nada disto depende da carga. Depende do tamanho da árvore, que é o que impede o custo de revelar seja o que for. Um acerto e uma falha custam os mesmos acessos oblívios, o stash mantém-se limitado à medida que a árvore cresce, e a cache continua a tomar as decisões que uma cache comum tomaria. A proteção mantém-se, a um custo elevado mas limitado, e a dissertação define o que um encaminhador totalmente oblívio ainda exigiria.

Palavras-chave: Redes Centradas em Conteúdo, Named Data Networking, Oblivious RAM, Path ORAM, Ambiente de Execução Confiável, Intel SGX, privacidade de padrões de acesso

Abstract

Content-centric networking retrieves data by name rather than by location, so any node holding a copy of a named object can answer a request for it. Named Data Networking is its most developed instance, and it makes routers stateful, keeping the state that supports in-network caching and request aggregation in a handful of forwarding tables. This design cuts latency and traffic, and it serves mobile and large-scale content distribution well, which is much of its appeal.

That same state exposes the consumer. Two of the tables give consumer behavior away, since monitoring the Content Store reveals whether an Interest was served from cache or missed, and monitoring the Pending Interest Table reveals the names currently in flight. Encrypting the content does not close this, because what leaks is the pattern of access to the tables and not the content itself. A forwarder that has to process requests over this state on an untrusted host therefore needs both its state hidden from the host and its access pattern hidden as well.

This thesis addresses both. The forwarder runs inside an Intel SGX enclave, which isolates its processing and state from the host, and a single Path ORAM tree backs the Content Store, the Pending Interest Table, and the name index, so that every table access moves a full root-to-leaf path and the cost of a request no longer reveals its outcome. State that leaves the enclave is encrypted, authenticated, and versioned against rollback, and a client checks it is talking to the genuine forwarder through remote attestation bound to the transport channel.

The contribution is a prototype built on NFD and the ndn-cxx library under Gramine, which for convenience of implementation runs SGX in simulation mode, together with an evaluation of what the protection costs. Placing the forwarder’s oblivious content store and a plain one under the same load, a lookup that a plain store answers in a fraction of a microsecond takes tens of milliseconds and moves tens of megabytes, throughput drops from millions of operations a second to a few tens, and memory rises to between seventeen and two hundred times a plain dictionary. None of this scales with the load. It scales with the size of the tree, which is what keeps the cost from giving anything away. A hit and a miss cost the same oblivious accesses, the stash stays bounded as the tree grows, and the cache still makes the decisions a plain one would. The protection holds, at a heavy but capped price, and the thesis sets out what a fully oblivious forwarder would still require.

Keywords: Content-Centric Networking, Named Data Networking, Oblivious RAM, Path ORAM, Trusted Execution Environment, Intel SGX, access-pattern privacy

Table of Contents

Acknowledgements ii
Resumo iii
Abstract iv
Table of Contents v
List of Tables viii
List of Figures ix
List of Abbreviations x

1. Introduction 1
   1.1. Objectives 3
   1.2. Report Outline 4
2. Background 5
   2.1. Content-Centric Networking 5
      2.1.1. NDN forwarding: Content Store, PIT, and FIB 7
   2.2. Oblivious RAM 10
      2.2.1. Threat model and definition 10
      2.2.2. Path ORAM 10
      2.2.3. Oblivious data structures 12
   2.3. Trusted Execution Environments 13
      2.3.1. Intel SGX 13
      2.3.2. Intel TDX 14
3. Related Work and Threat Model 16
   3.1. Security in Named Data Networking 16
   3.2. Related Work 17
      3.2.1. Privacy attacks against NDN/CCN forwarders 18
      3.2.2. Oblivious data structures in systems 18
      3.2.3. TEEs for network functions 19
   3.3. Threat Model 20
      3.3.1. System model 20
      3.3.2. Adversary model 21
4. Problem Analysis 23
   4.1. Data and the structures that hold it 23
      4.1.1. Data flow in the network 23
      4.1.2. The forwarding structures 24

4.2. Privacy-sensitive data structures 25
4.2.1. Content Store 26
4.2.2. Pending Interest Table 27
4.2.3. Name resolution 27
4.2.4. Distinguishing tables and outcomes 27
4.2.5. Side channels inside the enclave 28
4.2.6. Observation and tampering of externalized state 28
4.3. From the adversary to the requirements 29
5. Design 32
5.1. Design overview 32
5.2. A single shared oblivious store 33
5.3. Oblivious name resolution 35
5.4. Uniform request handling 37
5.5. Protecting externalized state 38
5.6. Attested entry point 40
5.7. Preserving forwarder behavior 41
6. Implementation 43
6.1. Prototype 43
6.2. The shared oblivious store 47
6.3. The oblivious name index 49
6.4. Integrating the Content Store and the PIT 50
6.5. Externalizing the bucket tree 51
6.6. The attested entry gateway 53
6.7. Correctness and cost of sharing the tree 53
6.8. Limitations 55
7. Evaluation 57
7.1. Experimental setup 57
7.1.1. Environment 57
7.1.2. Method 58
7.2. Results 60
7.2.1. Access cost and tree size 60
7.2.2. Comparison with the baseline 61
7.2.3. Memory footprint 65
7.3. Stash analysis 66
7.4. Discussion 67
8. Conclusion and Future Work 69
8.1. Summary 69

8.2. Future work 70

Bibliography 72

List of Tables

4.1. Summary of threats by forwarding component. 29

5.1. The table interface before and after: the same three operations and signatures, different work underneath. 42

6.1. Parameters the prototype runs with. 47

7.1. Machine and toolchain the measurements were taken on. 58
7.2. Per-operation cost of the oblivious store against a non-oblivious baseline, at the deployed 16-level tree. The baseline issues one access and moves about one entry. 62

List of Figures

2.1. Host-centered versus content-centered communication. 6
2.2. Processing an *Interest* through the NFD pipeline. 8
2.3. The *Data* return path through the NFD forwarder. 9
2.4. The Path ORAM structure, with the untrusted bucket tree and the client's *position map* and *stash*. 10
2.5. Isolation boundaries in Intel SGX and Intel TDX. 14

3.1. Network setting and trust boundary. 21

4.1. Data flow in a named-data network. 24
4.2. The forwarding structures and how one request threads through them. 25
4.3. The forwarder's structures as the adversary sees them. 26
4.4. Why a cache hit and a miss are distinguishable. 28
4.5. The subnetwork the protected forwarder runs in, and what the adversary sees on each host. 30

5.1. Design overview: enclave, single oblivious store, and the boundary with the untrusted host. 33
5.2. Block layout. A validity flag, the identifier, and the domain tag precede the table payload, all inside the encrypted block. 34
5.3. The oblivious name index. Nodes are blocks in the shared store, each pointing at its children and a data block by identifier. 37
5.4. Protecting an externalized bucket with AES-256-GCM and a version counter bound as additional authenticated data. 39
5.5. Attested entry. The client admits the channel only after checking the enclave's hardware quote and the key bound into it. 41

6.1. The deployed prototype. 45

7.1. Latency of a single oblivious access against tree capacity. 61
7.2. Per-operation latency of the oblivious store against the baseline (left) and the resulting overhead factor (right). 63
7.3. Throughput against cache size, oblivious store against the baseline. 64
7.4. Oblivious accesses per CS operation, with the hit and miss lines coinciding. 65
7.5. Memory footprint of the oblivious store against a plain dictionary. 66
7.6. Stash occupancy against tree size. 67

List of Abbreviations

AES Advanced Encryption Standard. ix, 38, 39, 46, 47, 51, 63

CCN Content-Centric Networking. 1, 3, 4, 6, 7, 18, 19, 69

CS Content Store. ix, 1–5, 7–9, 12–15, 18, 19, 21, 24, 26–28, 30, 32–34, 37, 38, 42–45, 47–50, 54, 65, 69–71

FIB Forwarding Information Base. 1, 7–9, 19, 21, 24, 26, 27, 30, 34, 37, 38, 42, 47, 48, 57, 59, 68, 69

GCM Galois/Counter Mode. ix, 38, 39, 46, 47, 51, 63

IP Internet Protocol. 1, 5, 16

NDN Named Data Networking. 1, 3–7, 16, 18, 20, 33, 42, 43, 48, 50, 51, 53, 55, 56, 58, 65, 69, 71

NFD NDN Forwarding Daemon. ix, 1–5, 7–9, 13, 14, 34, 43, 44, 46–50, 53, 58, 59, 68, 70, 71

ODS Oblivious Data Structure. 4, 12, 13, 19, 35, 43, 49

ORAM Oblivious RAM. 2, 4, 5, 10–13, 18–20, 30, 32, 35–38, 40–43, 56, 57, 62, 66, 70

PIT Pending Interest Table. 1–5, 7–9, 12–15, 17–19, 21, 22, 24, 26–28, 30, 32–34, 37, 38, 42–44, 47–49, 51, 54, 55, 57, 59, 68–71

RA-TLS Remote Attestation Transport Layer Security. 4, 40, 46, 53

RAM Random Access Memory. 19

SGX Software Guard Extensions. ix, 2–5, 13–15, 19, 22, 44, 46, 47, 52, 53, 56, 58, 62, 67–71

TCP/IP Transmission Control Protocol / Internet Protocol. 1, 5, 7, 19

TDX Trust Domain Extensions. ix, 2–5, 13–15, 44

TEE Trusted Execution Environment. 13, 15, 19–21, 29, 30, 44, 56, 57

TLS Transport Layer Security. 40, 42, 53

VM Virtual Machine. 3, 13–15, 44

Chapter 1

Introduction

Today's network is based on the location of content. Each request is sent to the origin server identified by a unique Internet Protocol (IP) address, and each client in the traditional Transmission Control Protocol / Internet Protocol (TCP/IP) network retrieves content from the server storing the content. If the server goes offline, the content becomes unavailable. Content-Centric Networking (CCN) [1] flips this design on its head. It shifts the network from host-based addressing to content, centering the network on a named object rather than on its location. One of the instances of CCN is Named Data Networking (NDN) [2]. The reference implementation of NDN, named NDN Forwarding Daemon (NFD) [3], encapsulates the core CCN architecture in three subsystems, the Content Store (CS), the Pending Interest Table (PIT), and the Forwarding Information Base (FIB), that allow the network to retrieve content by name and maintain forwarding-plane state necessary for retrieval.

In-network caching is performed by the CS, which stores a copy of data objects traversing the forwarder. Upon arrival of a subsequent request for the same named content, NFD satisfies the *Interest* using the cached copy, thus avoiding a round trip to the upstream. This reduces the end-to-end latency and also alleviates the network traffic by avoiding redundant transmission of the same content. The performance of the CS critically depends on the caching decisions that determine what content is to be cached and for how long.

The state of the outgoing *Interests* is kept by the PIT. If there are multiple *Interests* for a named object arriving at NFD prior to that object returning, NFD aggregates them into a single PIT entry with all the downstream faces that requested it. Serving a single upstream request on behalf of all these downstream requests prevents the same content from being forwarded upstream multiple times and conserves bandwidth. The lifetime of the PIT entries determines how long NFD responds to subsequent *Interests* for the same object, and how it prevents forwarding the same content packets more than once.

The last of the three is the FIB, which maps named-content prefixes to a next hop for each. The NFD uses a longest-prefix match on an incoming *Interest* to determine which

upstream host to forward to. Collectively, these three tables provide the NFD with the necessary state to route *Interests*, as well as respond to them from cache.

All this contributes to faster forwarding, and to a more accurate forwarding decision, but the state that enables these features can be exploited against the consumer. By monitoring the PIT and the CS caches, many of the consumers' network requests can be inferred [4, 5]. For example, a cache hit, instead of a cache miss, in the CS binds the consumer to content that passed through recently, and the manner in which the PIT gets populated and depopulated reveals, in a similar fashion, which *Interests* are looking up content under the same name at a time.

The ideal defense would be to trust the hosts a forwarder runs on, but that assumption does not hold on the modern internet, where a forwarder often runs on infrastructure its operator neither owns nor controls. The problem then splits in two. A compromised host must be kept from reading the confidential data the forwarder handles, and it must also be kept from inferring the forwarder's behavior once that data is hidden from it. The first calls for trusted hardware that keeps the forwarding logic out of the host's reach. The second calls for oblivious storage, which hides the pattern of accesses the forwarder makes to its own state.

A hardware-enforced isolated execution environment, also called a secure enclave, answers the first. Technologies such as Intel Software Guard Extensions (SGX) [6], Arm TrustZone [7], and Intel Trust Domain Extensions (TDX) [8] carve out a region of a system whose code and data the rest of the platform cannot touch, and an enclave of this kind keeps out even the operating system and the hypervisor. The forwarding logic of NFD, together with its CS and PIT, can therefore be moved inside an enclave, where the host can no longer read the state it holds.

An enclave on its own does not settle the second concern. Even when the host cannot read the enclave's memory, it can still watch how that memory is accessed, and for the CS and the PIT the sequence of accesses tracks what the consumer requested. Oblivious storage closes that channel. Oblivious RAM (ORAM) [9, 10] turns each access to a table into a sequence of physical accesses whose shape does not depend on which entry was touched, so a cache hit and a cache miss, or one pending name and another, look alike to whoever is watching the host. What ORAM hides is which entry each access touches. It does not, on its own, hide how many accesses a request makes or how long the request takes, so the design also fixes that count for every request. Fine-grained timing, at the

level of individual instructions, stays outside the scope that Chapter 3 sets. Applied to the CS and the PIT, it hides the access patterns that would otherwise give the consumer away.

1.1. Objectives

The aim of this thesis is to identify and analyze the security vulnerabilities inherent to CCN/NDN forwarding architectures, and to investigate how these risks can be reduced through algorithmic and cryptographic-hardware mechanisms. Concretely, the work maps the threats a NDN forwarder faces from the host it runs on. The focus is the privacy of consumer requests, together with the confidentiality, integrity, and freshness of the state the forwarder has to keep in host memory, and the CS and the PIT are its most exposed components. Availability attacks such as *Interest* flooding are real but stay outside this scope, as Chapter 3 explains. From that mapping it designs an oblivious store that integrates Path ORAM into the CS, the PIT, and the name index that resolves lookups into them, so that memory access patterns no longer betray consumer behavior. It isolates that logic with hardware-enforced execution, building the prototype on the Intel SGX model through Gramine, whose process-level enclave fits isolating the forwarder itself rather than the Virtual Machine (VM)-level alternative of Intel TDX. Finally, it measures what the design costs, setting the oblivious forwarder against an equivalent non-oblivious one under the same workload, and confirms that the running prototype still exhibits the guarantees the construction provides.

Put as a single research question, the thesis asks whether a NDN forwarder running on a host it does not trust can keep that host from learning what consumers request, both from the state it holds and from the way it accesses that state, without changing how it forwards, and what that protection costs. The method follows from the question. A threat analysis of the forwarding structures yields a set of requirements, a design is built to meet each of them, a prototype implements the design inside NFD, and an evaluation measures its cost against a non-oblivious baseline.

The contributions of this work are the following:

- an analysis of what each forwarding structure leaks to the host, condensed into six requirements, R1 to R6 (Chapter 4);
- a design that places the CS, the PIT, and the name index in a single shared Path ORAM store, with domain tags kept inside the encrypted blocks so that no access reveals the table it serves (Chapter 5);

- an oblivious name index, our own implementation of an Oblivious Data Structure (ODS)-style AVL tree built over that same store (Chapters 5 and 6);
- the protection of the externalized bucket tree with authenticated encryption and per-bucket versions against rollback, and an attested gateway in the Remote Attestation Transport Layer Security (RA-TLS) style (Chapters 5 and 6);
- a prototype that integrates all of this into NFD under Gramine (Chapter 6);
- an evaluation of its cost in latency, bandwidth, throughput, and memory against a non-oblivious store (Chapter 7).

1.2. Report Outline

The rest of this report is organized as follows. Chapter 2 reviews the background the thesis relies on, from the CCN/NDN architecture and its forwarding tables to the principles of ORAM, Path ORAM, and trusted execution on Intel SGX and TDX. Chapter 3 surveys related work on privacy in CCN/NDN, on oblivious storage, and on trusted execution for network functions, and then fixes the adversary we assume and the assets we protect. Chapter 4 recalls what state a forwarder holds, states what an adversary learns from each structure, and derives the security requirements that guide the design. Chapter 5 presents the design, instantiating Path ORAM as a single oblivious store shared by the CS, PIT, and name index inside an SGX enclave. Chapter 6 describes the prototype that integrates these mechanisms into NFD on the ndn-cxx library. Chapter 7 measures what the protection costs against a non-oblivious baseline and confirms that the forwarder still behaves as a named-data cache. Chapter 8 closes with the main insights and outlines future work.

Chapter 2

Background

This chapter sets out the technical background this thesis builds on. We begin by reviewing the transition from host-centric communication to content-centric communication and describe the routing architecture of NDN in sufficient detail to justify the threat model developed later on. Next, we present ORAM as a generic primitive for hiding memory access patterns and introduce Path ORAM as the concrete construction we instantiate over CS and PIT. We close with a review of trusted execution technologies, focusing on Intel SGX and Intel TDX, as they define the design space available to isolate the routing logic of NFD from a compromised host.

2.1. Content-Centric Networking

The internet's main architecture is the TCP/IP stack, in which all communication is tied to a pair of host addresses. A user interested in a particular set of data must first find a server that holds it, establish a connection using the server's IP address, and only then retrieve the content.

This misalignment between *what* the user wants and *where* that content is located has well documented costs, each of which follows from the same root cause, that a name ultimately resolves to a host address:

- Load concentrates on the origin servers. Since the content is reachable only through the address of the server that holds it, every consumer of a popular object is sent to that same server, even when an identical copy has just crossed a nearby link. The network moves the same bytes again because it cannot reuse a copy it already carried.
- Mobility is hard to support. An IP address names a location in the topology, so a host that moves is given a new address and its open connections break. A host reachable through several links, in turn, needs extra machinery to make those addresses appear as one endpoint.

- Content distribution has to be overlaid on the network. Because the network only moves bytes between addresses and is unaware of the content, caching and replication cannot be provided by it and are instead pushed up into content delivery networks and application-level overlays.

CCN reshapes communication so that data is bound to a name rather than to the hosts that store it. A request, called an *Interest*, carries the name of the content the consumer wants, and any node holding a valid copy of that object can answer it.

The two names for this idea come from two stages of the same line of work. CCN is the original architecture, proposed by Jacobson et al. [1] at PARC, which put content naming at the center of the network. NDN [2] is the research project that grew out of it and turned the proposal into a concrete design, with a packet format, a hierarchical naming scheme, and defined forwarding semantics. The distance between the two is one of maturity and specification more than of principle, and NDN is the version with a running reference implementation, so it is the one this thesis targets.

The model pays off most where the same content is requested by many consumers, or where the endpoint is not fixed. Large-scale content distribution is the clearest case, since any router with a cached copy can serve it without reaching the producer. Mobile or intermittently connected consumers gain as well, because a request names what is wanted rather than a location and can be satisfied by whichever nearby node holds the data.

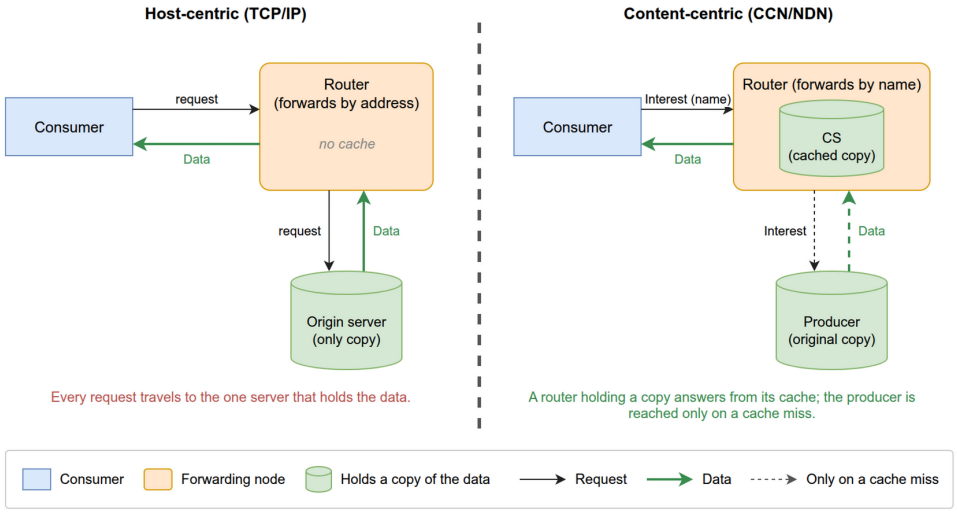


Figure 2.1: Host-centered versus content-centered communication.

Figure 2.1 contrasts the two models. Under TCP/IP the consumer has to reach the one origin server that holds the data, so every request travels to the same host. Under CCN the consumer names the content instead, and any router already holding a copy can answer, with the producer contacted only on a cache miss.

This shift has two consequences that matter for the rest of this thesis. First, because *Data* is self-identifying and signed at the source, any router on the path can safely cache it and serve it again. In-network caching becomes a first-class network function rather than a layer optimization. Second, the network has to remember, for each *Interest*, where the request came from, so it can return the matching *Data*. This makes NDN routers *stateful*, unlike TCP/IP routers.

These ideas are backed by an active research ecosystem rather than a single paper. NDN is maintained as an open research project with a reference forwarder, the NFD [3], and a testbed that links participating institutions across the public internet. Around it, studies have applied content naming to settings such as the Internet of Things, vehicular networks, and the exchange of large scientific datasets. This dissertation does not extend that application work. It takes the forwarder as given and asks what its state reveals to the host it runs on.

2.1.1 NDN forwarding: Content Store, PIT, and FIB

The NFD [3] is the reference implementation of NDN. It realizes the forwarding model through three coordinated tables, the CS, the PIT, and the FIB [11].

Before, in a TCP/IP router, forwarding is close to memoryless. A packet is matched against a routing table and sent on toward a destination address, and the router keeps nothing about the request once the packet has left. Caching a response, noticing that two clients asked for the same object, or serving data from anywhere other than the destination are all things that happen at the endpoints, not in the network. An NDN forwarder now keeps state so that these decisions can be made inside the network and per name, and it splits that state across the three tables:

- the CS, a cache of *Data* the node has recently carried, which lets a matching *Interest* be answered locally;
- the PIT, a record of *Interests* forwarded upstream but not yet answered, which lets duplicate requests for one name be collapsed and the returning *Data* sent back to every face that asked;

- the FIB, a map from name prefixes to the upstream faces that lead toward a producer, which lets an unanswered *Interest* be routed by name.

An incoming *Interest* is checked against these tables in the same order, so it is answered locally when possible, merged with a request already in flight when one exists, and otherwise sent upstream toward the producer.

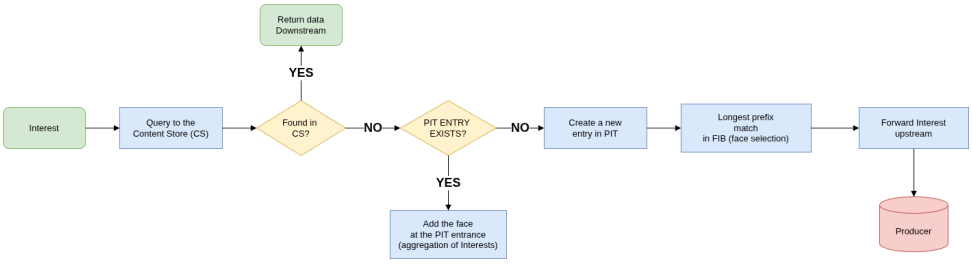


Figure 2.2: Processing an *Interest* through the NFD pipeline.

Figure 2.2 illustrates this pass. An incoming *Interest* is first checked against the CS, and on a hit the cached *Data* is returned downstream at once, so the request never leaves the node. On a miss the *Interest* is recorded in the PIT, where a request already in flight for the same name absorbs it, and otherwise the FIB selects an upstream face and the *Interest* is forwarded on toward the producer.

The rest of this section follows that pass one table at a time. The forwarder first checks the CS, a cache of recently seen *Data* packets indexed by name. It looks for a stored packet whose name matches the *Interest*, either an exact match on the full name or a prefix match when the *Interest* allows it, and the result is one of two outcomes:

- A ***hit***: the cached *Data* is returned downstream right away and the request never leaves this node. This cuts upstream traffic and end-to-end latency.
- A ***miss***: the lookup moves on to the PIT.

The PIT records *Interests* that have been forwarded upstream but not yet answered. Each entry links a content name to the downstream faces that asked for it. A repeated *Interest* for a name already pending does not add a second entry. The forwarder appends the new face to the existing one, so a single upstream retrieval can be multicast back to every interested face when the *Data* returns. Each entry also carries a timer that clears *Interests* that go unanswered.

Once the *Interest* is recorded in the PIT, the forwarder consults the FIB, which maps name prefixes to the upstream faces that can reach a producer for the prefix. A longest-prefix match selects the outgoing face, optionally guided by a strategy module that picks among candidates on metrics such as performance or on policy. Unlike the CS and the PIT, the FIB changes on the slow timescale of the routing plane rather than per request, since it reflects routing state and not consumer behavior.

These tables are not consulted in isolation, and they chain into a single loop. On the way up, a CS miss hands the name to the PIT, and a freshly created PIT entry hands it to the FIB, which selects the upstream face. On the way back, returning *Data* follows the mirror path shown in Figure 2.3. It is matched against the PIT to find the downstream faces waiting for it, is sent to each of them, and is then placed in the CS, so that a later *Interest* for the same name can be answered without leaving the node. The PIT entry is then cleared, while the FIB stays untouched. *Data* that matches no PIT entry is unsolicited and dropped. One upstream retrieval is turned into a local answer for every future request along the same path.

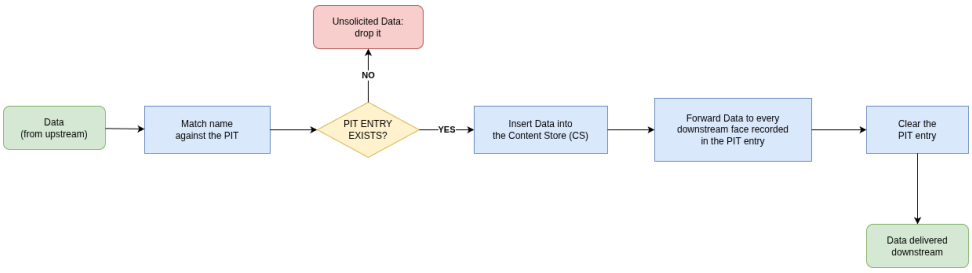


Figure 2.3: The *Data* return path through the NFD forwarder.

For this thesis, the key point is that the CS and the PIT lie on the critical path of every *Interest*, and that how they respond depends on what the client requested.

NDN builds several security functions into the network layer. *Data* is signed at the source, content can be encrypted under name-based access control, and the stateful forwarding plane raises availability concerns of its own. None of these protects the consumer's privacy against the forwarder itself. The access patterns of the CS and the PIT still reveal what is being requested. Chapter 3 reviews this body of work and develops the threat model this thesis defends.

2.2. Oblivious RAM

2.2.1 Threat model and definition

ORAM, introduced by Goldreich and Ostrovsky [9], refers to a cryptographic primitive that hides a client's access pattern from a passive observer of the storage system. The scenario involves a trusted client with a limited amount of local memory and an untrusted server that possesses a significantly larger vector of encrypted blocks. The adversary is allowed to observe every physical interaction that the client sends to the server, the address, the direction (read or write), and the duration of the event. What it must not learn is the logical sequence of reads and writes that the client actually performs.

Formally, an ORAM scheme is secure if, for any two sequences of operations of the same length, the resulting distributions over the physical access patterns are computationally indistinguishable. Encryption hides the content of each block but on its own does not satisfy this property, since the addresses the adversary observes still reveal the logical location the client accessed, and repeated accesses to the same location remain visible as such.

ORAM addresses this by converting each logical access into a random sequence of physical accesses, whose distribution is independent of the logical location.

2.2.2 Path ORAM

Path ORAM, proposed by Stefanov et al. [10], is the approach we adopt in this dissertation. Tree-based ORAM is currently the most widely used family, since it combines a simple invariant with bandwidth costs that practical systems can afford.

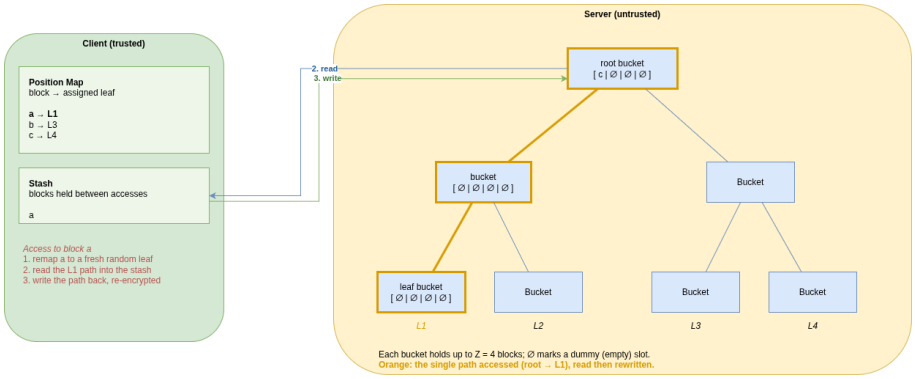


Figure 2.4: The Path ORAM structure, with the untrusted bucket tree and the client's *position map* and *stash*.

Figure 2.4 shows the layout. A *logical block* is one fixed-size unit of the client's data, the granularity at which Path ORAM reads and writes. The encrypted storage is organized

by the server as a binary tree composed of *buckets*, where each *bucket* contains a small, fixed number Z of blocks. The client has two auxiliary structures, a *position map* that, for each logical block, records the leaf to which the block is currently associated, and a small *stash* that temporarily stores blocks in transit. The central invariant of the Path ORAM is that each block is located somewhere along the path from the root to the leaf, as indicated by its entry in the *stash* or the *position map*.

The two client structures scale differently. The *position map* holds one entry per logical block, so it grows linearly with N. When it is too large to keep on the client, Path ORAM stores it inside a smaller ORAM of the same kind, and the construction repeats until the top-level map is small enough to hold directly [10]. The *stash*, by contrast, holds only the few blocks that did not fit back onto the path after an access, and it stays small with high probability.

The access to the logical block a occurs in three steps:

1. The client checks the current leaf ℓ linked to a in the *position map* and immediately remaps a to a leaf ℓ' sampled uniformly and randomly.
2. The client reads all *buckets* along the path from the root to ℓ, decrypts each block, and transfers them all to the *stash*. Since the invariant is preserved, the desired block a remains secure in the *stash*, where the client can read or update it.
3. The client rewrites the path from the root to ℓ, fitting as many blocks from the *stash* as possible into each *bucket*, taking into account the new *position map*. A block can only be inserted into a *bucket* if it lies on the path from the root to the leaf currently linked to that block.

Whether the logical operation is a read or a write, these steps are the same. The block is remapped to a fresh leaf in the *position map*, pulled through the *stash*, and the whole path is read back and rewritten. A read and a write are therefore indistinguishable to the server.

Two things follow from this. Every logical access produces the same physical pattern, a full root-to-leaf read and then a full root-to-leaf write along a uniformly random path, so the server cannot tell which block was touched or whether the access was a read or a write. And a single logical access is no longer a single memory access. It is a traversal of one root-to-leaf path, so its cost grows only with the height of the tree, proportional to $O(\log N)$ blocks for a logical vector of size N. Those constant factors are small enough

that Path ORAM has served as the foundation for processor architectures and file systems based on ORAM [12, 13].

This means that for the CS and the PIT, every query triggered by an *Interest* accesses a path whose structure is independent of the name being looked up.

2.2.3 Oblivious data structures

ORAM hides accesses to a flat array of blocks. Part of a forwarder's state is not a flat array but a pointer-based structure, since the name index is a search tree, or map, that is walked one node at a time. Such a structure can be placed on top of ORAM, but doing so naively is wasteful. Each step of a traversal is a logical access to a block whose location is unknown until the parent has been read, so every step needs its own *position map* lookup, and that map carries the recursive cost noted above.

Wang et al. [14] observe that the access pattern of many data structures is partly predictable. Once the program reads a tree node, the next node it can read is one of that node's children. An ODS exploits this by storing, inside each node, the ORAM position of its children. A traversal then follows positions it already holds, one ORAM access per node along the path, and no separate *position map* is consulted per step. For a structure with N entries this removes the recursive position-map overhead a plain ORAM mapping would pay, and keeps a lookup proportional to the depth of the structure. The contrast with plain Path ORAM is then simple. Path ORAM makes a single access to a flat array oblivious, whereas an ODS makes a whole traversal of a pointer structure oblivious, at a cost of one Path ORAM access per node visited instead of a recursive *position map* lookup at each step.

Two broad routes exist for making a pointer-based structure oblivious. One keeps a general ORAM underneath and treats every node as an ordinary block, which is simple but pays the recursive *position map* cost at each step of the walk. The other specializes the structure, so that each node stores the ORAM positions of its children and the traversal spends a single ORAM access per node. The oblivious maps and search trees that have appeared in the literature [14–16] are variations on this second route, and Chapter 3 places them among related work. This thesis takes that route.

The concrete instance is the oblivious name index used in the following chapters. The index is a search tree whose traversal is made oblivious in this way, while the CS and PIT entries it resolves to live in the same underlying Path ORAM tree. Because the forwarder's

name index is exactly such a node-by-node tree walk, an ODS is the fitting choice, it delivers obliviousness at a cost proportional to the tree's depth, instead of the recursive overhead a flat Path ORAM mapping would impose.

2.3. Trusted Execution Environments

Although ORAM conceals access patterns from an adversary observing the storage, it assumes the client is trusted, since the client performs the *position map* lookup and reads and rewrites the path. The accesses at stake are the ones from Section 2.1.1, where every *Interest* drives a CS and a PIT lookup, and Path ORAM hides those only while the client issuing them cannot be observed. Here that client is the NFD forwarder, which runs on a host operating system we do not trust. By isolating an execution region whose code and data are protected from the rest of the platform, including the hypervisor and the operating system, a Trusted Execution Environment (TEE) keeps the client's execution on hardware the untrusted host cannot observe.

Comparable technologies exist from several vendors, such as Arm TrustZone [7] on mobile and embedded platforms and AMD's Secure Encrypted Virtualization [17] for encrypting whole virtual machines. This section concentrates on Intel SGX and Intel TDX, since they capture the two isolation granularities this design has to weigh, a single process and a whole VM, and since the prototype builds on the process-level one.

2.3.1 Intel SGX

The first TEE on commercially accessible x86 systems was the Intel SGX [6]. It presents the idea of an enclave, a virtual and isolated memory region at the process level whose pages are encrypted and whose integrity the processor protects. No code outside the enclave may read those contents, and that includes the operating system kernel. A basic attestation technique enables a remote verifier to confirm that a particular enclave is running a piece of code as intended on authentic Intel hardware. Entry into and departure from the enclave entail a restricted set of processor instructions that establish a controlled call interface.

Two limitations of SGX matter for router-like workloads. The *enclave page cache* is small, historically 128 or 256 megabytes of usable memory on client platforms, and system calls require expensive transitions out of the enclave. Beyond that, several microarchitectural side channels, including page-fault sequences, cache timing, and branch-predictor state, have been shown to leak information about execution inside the enclave despite memory encryption [18, 19].

These restrictions are limiting for a high-throughput router that exchanges packets with the network on each operation and keeps gigabytes of CS and PIT state.

These limitations also shape how an application is brought into an enclave in the first place. Porting a large code base such as NFD to the narrow enclave call interface by hand is laborious, and a common alternative is to run the unmodified application on top of a *library operating system* that itself lives inside the enclave and services the application's system calls from within the protected region.

Gramine [20] is one such library OS for SGX, and it is the execution model on which the prototype in this thesis is built. It lets the existing NFD forwarder run inside an enclave without source-level porting. Gramine does not, however, remove the enclave-page-cache budget or the microarchitectural side channels described above, and it only addresses the engineering cost of entering the enclave. That is one reason an oblivious access mechanism is still required for the forwarding state the enclave operates on.

2.3.2 Intel TDX

Intel TDX extends hardware isolation from a process-level enclave to a whole virtual machine [8]. A *trust domain* is a hardware-protected VM whose memory is encrypted with a domain-specific key and whose execution state is protected from the hypervisor by a small Intel-signed module that mediates the VM's inputs and outputs. From the guest's point of view, the *trust domain* runs a conventional operating system and can host arbitrary user-space code, including unmodified network stacks and routers.

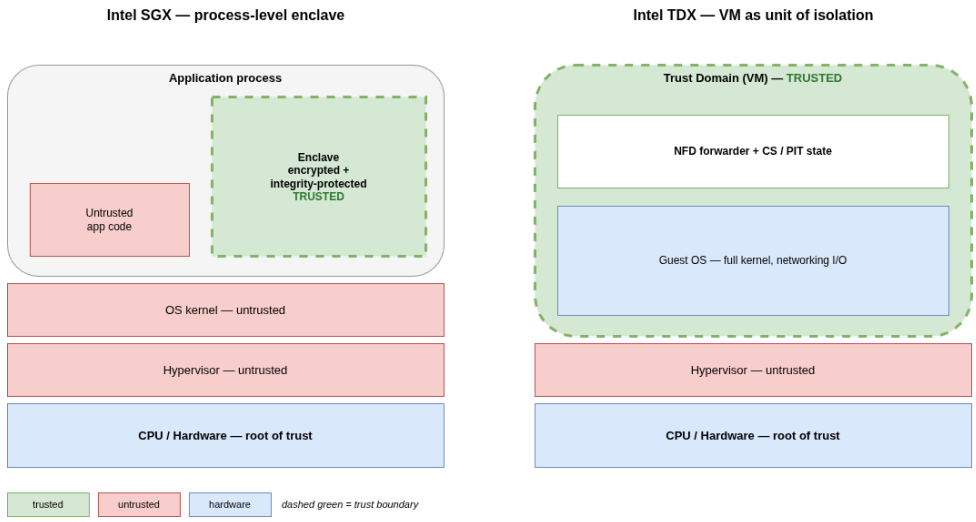


Figure 2.5: Isolation boundaries in Intel SGX and Intel TDX.

Figure 2.5 contrasts the two isolation boundaries. Under SGX the process-level enclave alone sits inside the trust boundary, and the host kernel stays outside it. Under the VM-level trust domain of TDX a full guest operating system runs within the protected boundary, and only the hypervisor is left untrusted.

Of these two, this work builds on SGX, and it is worth saying why TDX is set aside. A trust domain protects a whole VM, its guest operating system included, which is more than the design needs. What has to be isolated is the forwarder, not the system around it, and the process-level enclave of SGX draws the trust boundary tightly around the CS, PIT, and name index while leaving the rest of the platform outside it. TDX is also the newer of the two, and it is found mainly on recent server parts, typically hosted infrastructure an operator does not own or control, whereas the process-level model is the better studied and the closer fit for isolating a single network function. TDX therefore stays in the design space as the VM-level alternative rather than the mechanism this work adopts. For convenience of implementation, the prototype runs SGX in simulation mode through Gramine, which supplies the “run an unmodified forwarder inside a TEE” property the design relies on, and the implementation and evaluation chapters return to this point.

Chapter 3

Related Work and Threat Model

This chapter places the thesis among existing work and fixes the threat model it defends. Section 3.1 reviews what NDN secures by design and the one property it leaves open, the consumer's privacy against the forwarder. Section 3.2 turns to the work closest to ours and uses it to make the problem precise, both the privacy attacks that expose it and the two families of defense we build on. Section 3.3 closes by fixing the system we assume, the adversary against it, and the assets we protect.

3.1. Security in Named Data Networking

NDN was a system design exercise, not a security design exercise. Its goal was to retrieve content by name, and no security features were designed into the system. Securing NDN post hoc is not merely the addition of a security feature. It involves making the entire forwarding plane secure. Approaches to secure the IP layer thus require intrusive re-designs, as was the case with IPsec [21]. One challenge unique to content-centric forwarding is that routing is dependent on what content is being requested. A forwarder matches an incoming name against its cache and its routing table, requiring it to read the name. In IP the destination address also travels in the clear, so the two endpoints of a conversation are never hidden from the network. The difference is in what the clear-text field says. An IP address names a host, and what is being asked of that host can be encrypted end to end. An NDN name is the description of the content itself, and the forwarder needs it in the clear to route the request, so the field that must stay readable is exactly the one that reveals what the consumer wants. Hence, the confidentiality of requested content cannot be achieved by simply encrypting the requests.

Nevertheless, there is a set of work that pushes several security functionalities to the network layer. We briefly discuss these to separate what NDN is designed to secure and what it is designed to leave open, considering that this thesis aims at something that the above mechanisms do not address.

All *Data* packets are named and signed by the producer, allowing any node to verify a packet without relying on the node from which it received the packet [2]. Content integrity and provenance are native properties of the network layer and are not added as an afterthought. The operational aspect remains, namely determining which key is authorized to sign which name and automating this process. Trust schemas provide a mechanism to describe these key-to-name relationships as a policy that can be validated by a forwarder or application in an automated fashion [22].

But content is not obfuscated by signing, so confidentiality is addressed independently. Access control based on names can encrypt *Data* at the time of production and distribute decryption keys using naming conventions, so that only authorized consumers can decrypt a cached copy of the object, which may be stored in every node [23].

The stateful forwarding that enables caching and aggregation also introduces a denial-of-service attack surface. Interest flooding places a large number of *Interests* in the PIT for names that will never be satisfied. A line of work characterizes and mitigates these attacks [24, 25]. We merely note this surface for the sake of completeness. It is outside the scope of this work.

These mechanisms protect the content and the routing of requests, but do not protect consumers' privacy from the forwarder. This is the access-pattern problem. Because the forwarder operates on names and maintains state based on names, its actions reveal what is being requested, even if the *Data* is signed and encrypted. For example, whether a name is cached, whether a particular *Interest* is pending, and which entry a lookup accesses all reveal what a consumer is requesting, and all of these can be inferred from the behavior of the forwarder [4, 5]. This thesis focuses on addressing this leakage. It is largely independent of the previous work on authenticity, confidentiality, and availability, and we revisit this topic in Section 3.2.

3.2. **Related Work**

This section examines the work closest to ours. We start from the attacks that reveal access-pattern leakage in content-centric forwarders, which fixes what our defense needs to hide. We then discuss the two families of defense that this thesis builds on. Oblivious storage hides the identity of the accessed entry, while trusted execution separates the forwarder from the host machine it runs on. Neither is sufficient on its own. The former does not prevent the host from reading the forwarder's memory, while an enclave does

not hide the access pattern within it. The two are complementary. Below, we discuss each family of techniques, first describing the generic approach, then how it relates to what we build.

3.2.1 Privacy attacks against NDN/CCN forwarders

Early work by Acs et al. [4] and Mohaisen et al. [5] began showing that a CCN router is a side channel of interest. The adversary can learn whether each response is a cache *hit* or cache *miss* by issuing probing *Interests* and measuring the response times, thereby recovering the access history of previous clients.

Subsequent work extended these attacks to the PIT, where pending-entry timers reveal competing *Interests* for the same name [25]. These attacks fix what a defense must hide, not the cached bytes, which signing and access control already cover, but the fact of a hit or a miss and the identity of the entry a request touches.

The countermeasures those papers propose share a shape. They perturb the signal an observer measures rather than remove the channel that carries it, either by degrading the timing difference between a hit and a miss or by limiting what the cache is allowed to serve. Concretely, they are *statistical*, such as cacheability signals under the producer's control, cache partitioning, and random delays. They sacrifice the latency and bandwidth benefit that the cache provides, and only weaken the signal without actually eliminating the channel. Our approach is different. Instead of modifying what the CS and PIT would reveal through their actions, we make their access patterns indistinguishable in the first place.

3.2.2 Oblivious data structures in systems

The first of the two families is oblivious storage, which hides the identity of the accessed entry. Many existing oblivious systems, such as ObliviStore [26], Phantom [12], and ORAM-based file systems [27], use Path ORAM as their storage abstraction. These systems show that the $O(\log N)$ overhead of Path ORAM is practically efficient for moderately sized states, especially when the client is co-located with the store, for example on the same machine. The architecture is a trusted client storing a small amount of state, and an untrusted server storing encrypted data. None of them was built for a named-data forwarder. They were designed and evaluated in other settings, oblivious cloud storage, secure processors, and encrypted file systems, and are orthogonal to NDN. What carries over to our problem is the storage abstraction, not the setting.

More relevant to our work, a recent series of works considered ODSs *within* TEEs, the host's Random Access Memory (RAM) outside the protected area being the untrusted server, and the enclave being the ORAM client [15, 16]. We adopt this setting. The enclave is memory-constrained, so the majority of the data structure must reside outside the enclave, while the client code resides inside.

We adopt Path ORAM for the CS and PIT in a similar fashion. The ORAM client logic along with the *position map* and *stash* run inside the protected region, while the bucket tree is authenticated-encrypted and may reside in the untrusted host memory, so that the limited enclave memory need not store the entire forwarding state. There are two differences in our instantiation. First, we want to hide not only from someone observing the encrypted store outside the enclave but also from microarchitectural side channels that are still visible inside the enclave. Second, the CS and PIT share the same ORAM tree as the name index, so an adversary monitoring the access pattern cannot tell which of these tables is being touched, much less which name.

3.2.3 TEEs for network functions

The second family is trusted execution, which isolates the forwarder from the host it is executed on. In an untrusted network operator scenario, such as multi-tenant clouds and managed services, a contemporaneous set of works envisions and implements network functions inside TEEs. ShieldBox [28] implements a Click-based *middlebox* inside an SGX enclave, hiding packet processing from the host, and evaluates the throughput overhead caused by enclave transitions. SafeBricks [29] protects a network function from the cloud provider that executes it, and partitions the network function so that only the minimal packet processing core is inside the enclave and enforces least-privilege among its modules. Slick [30] constructs a secure *middlebox* along similar lines, evaluating the packet manipulation latency introduced by the enclave. All these systems are targeted towards TCP/IP *middleboxes* rather than CCN routers, and none of them considers leakage via access pattern that occurs inside the protected region, but they show that routing-type workloads can be executed inside a TEE. The closest work in the named-data setting is SGX-ICN [31], which runs a routing enclave on every router, built on CCNx under the Occlum library operating system, next to enclaves that enforce access control and distribute keys. Its router keeps the name-tree indexes of the CS, the PIT, and the FIB inside the enclave and the encrypted contents in untrusted host memory, protected by authenticated encryption and, for large objects, a Merkle tree. It shares our starting point,

an enclave on the forwarding node with the bulk of the state kept outside it, but it leaves the access pattern open. Nothing in it hides which encrypted entry a lookup fetches from host memory, the in-enclave index remains exposed to the side channels the authors acknowledge, and they name protecting access privacy to untrusted memory as future work. That leakage is the one this thesis sets out to close. This is where our two families meet. We inherit the isolation that these systems provide, and augment it with a layer of ORAM inside the TEE around the NDN forwarder's state to close the access pattern that isolation by itself leaves exposed.

That combination has significance only against a specific adversary. The attacks above describe what needs to be hidden, and the defenses describe how, but neither characterizes the adversary against whom the forwarder must protect or what the forwarder needs to protect.

3.3. Threat Model

The forwarder operates on an untrusted host, in between the consumers that send out *Interests* and the producers that generate *Data*. Our objective is to add more confidentiality to what the network already provides to consumers and producers. In particular, we want to hide from the host the *Data* that a consumer retrieves, as well as the consumers' retrieval patterns, that is, how consumers are retrieving which content. We want to achieve these confidentiality requirements with as few modifications as possible, so we do not consider the entire network to be honest, nor do we aim to protect it entirely. Only the hosts running our forwarder are protected, and the rest of the network remains as is. In the rest of this section, we describe the assumed system, the adversary, and the assets we protect.

3.3.1 System model

As illustrated in Figure 3.1, the network has a set of consumers requesting content, a set of producers publishing content, and a set of forwarding hosts that forward requests between consumers and producers. Communication between consumers and producers through forwarders is performed over a synchronous network using *Interest* and *Data* packets. Each link may or may not be secure, as we are not relying on confidentiality of communication on the wire. The leakage we address happens at the forwarder rather than on the wire.

To keep our design modular and as non-intrusive as possible, not all forwarding hosts are protected. A host is said to be protected if it runs our forwarder in a TEE, while other hosts

run a normal forwarder and therefore reside outside our trust boundary. We choose to protect forwarding hosts one at a time as opposed to protecting the entire network. On a protected host, the boundary is defined by the enclave. The forwarding logic, together with the runtime state described in Chapter 2, namely the CS, the PIT, the FIB, and a name index that maps a received name to the matching entry, runs inside the enclave. The bucket tree that supports these tables resides in ordinary memory of the host outside the enclave, as it is too large to fit in enclave memory, and is stored there in an encrypted form. We protect the forwarding logic and the above-mentioned state. The adversary is assumed to have control over everything outside the enclave on a protected host.

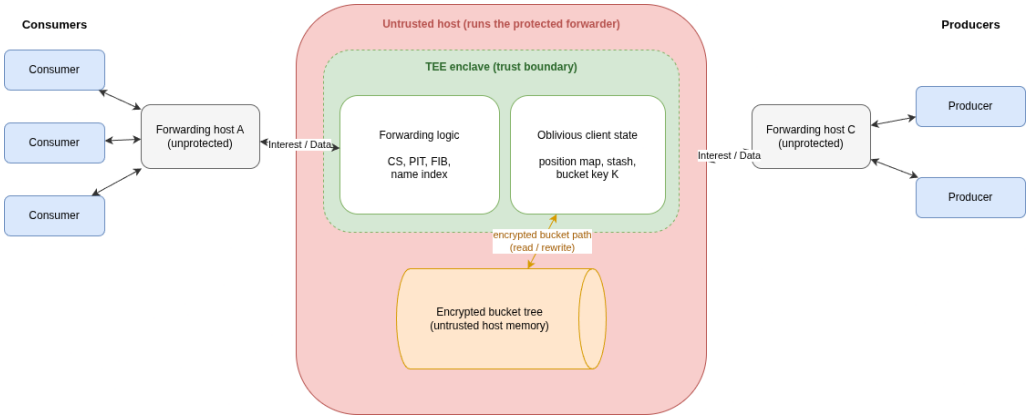


Figure 3.1: Network setting and trust boundary.

3.3.2 Adversary model

On a protected host we assume a host-level adversary. The operating system, the hypervisor, and anyone with physical access to the machine are untrusted and may act maliciously, which is the standard assumption for code that runs on a platform someone else owns. We state the model as a set of assumptions:

- Protected hosts are equipped with TEE hardware, and the forwarder relies on it for isolation.
- The isolated environment does not have unlimited memory, so most of the forwarding state has to be kept outside it.
- The adversary cannot read the memory inside the TEE enclave.
- The adversary can read all memory outside the enclave, and can modify it and replay stale versions.

- The adversary can observe the pattern of memory accesses the enclave makes, directly for the external memory it reads, and through microarchitectural side channels such as cache timing and page faults for the memory inside the enclave [18, 19]. The number of accesses a request triggers and the time it takes are observable in the same way.

We assume the adversary is incapable of subverting cryptographic primitives, or the hardware-enforced isolation of the enclave. The information that an adversary learns is thus from outside the enclave, or based on the inferences the adversary can make from the patterns and timings of the enclave's behavior.

There are other threats that are real but lie outside this model. We do not consider the presence of a network adversary that observes traffic over the wire and deduces user activities based on packet timing and volume. Likewise, we do not consider attacks aimed at causing denial of service, such as *Interest* flooding against the PIT [25]. We also do not attempt to conceal routing decisions since prefixes are not consumer-specific, unlike the requested names. Physical side channels such as power analysis are also out of scope.

Fine-grained timing attacks are likewise outside the scope of this work. We are aware that such attacks are prevalent on hardware like SGX, where controlled-channel and cache-timing attacks recover information about an enclave's execution [32, 33]. Our design removes the input-dependence of the access pattern and pads each request to a fixed number of oblivious accesses, but it does not claim to execute in constant time at the instruction level. Extending it to resist timing attacks would mean carrying a constant-time discipline [34] through both the SGX code and the oblivious data structure it drives, which would be an interesting piece of work in its own right.

Chapter 4

Problem Analysis

Ensuring that a forwarder's structures are properly protected is not a simple matter, because the system beneath them is not simple. A single *Interest* is held across several structures, and resolving it touches a varied set of them, which leaves room for subtlety both in understanding these mechanisms and in protecting them. The aim of this chapter is to make concrete what is explicitly being protected, so that we can bound with some rigor how far that protection reaches. We start from the plain picture, recalling what data a named-data forwarder handles and which structures hold it, with no reference yet to security. We then take those structures one at a time and ask what an adversary learns by watching each. From the adversary's capabilities we derive the security requirements that the design in Chapter 5 is built to meet, and we close by weighing what those guarantees will cost.

4.1. Data and the structures that hold it

Before asking what an adversary can learn, it helps to have the plain picture of what the forwarder handles and how it handles it, with security set aside for the moment. Chapter 2 introduced these mechanics, and here we recall just enough of them to fix what is at stake.

4.1.1 Data flow in the network

In a named-data network, content is addressed by name rather than by the host that stores it. A consumer that wants a piece of content issues an *Interest* carrying its name, and the network replies with the matching *Data*. The *Interest* travels hop by hop from the consumer toward whoever can satisfy it, a producer or any router along the way that already holds a copy, and the *Data* retraces the reverse path back to the consumer. If no router on the path has the content, the *Interest* reaches the producer, which owns the name and signs and returns the *Data*. When several consumers ask for the same name at close to the same time, a router that already has a request in flight for it does not forward a second one. It records the new consumer and later releases the returning *Data* to all of them. Because content is named rather than tied to a location, any router on the path may also

keep a copy of the *Data* it forwards and answer a later request for the same name itself. Figure 4.1 shows one such exchange.

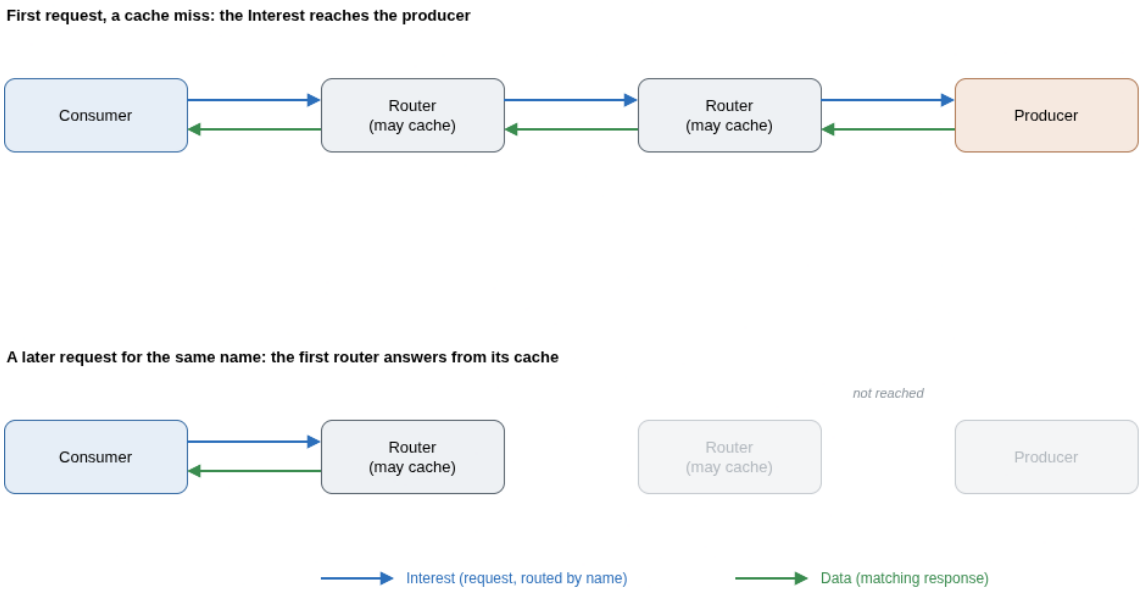


Figure 4.1: Data flow in a named-data network.

4.1.2 The forwarding structures

A router carries out this exchange with three tables and a name index in front of them. The CS caches *Data* the router has recently forwarded, so that a repeated request can be answered locally. The PIT records *Interests* that have been forwarded and are still waiting for their *Data*, and it collapses several requests for the same name into one pending entry. The FIB holds the routes toward producers and says where an unsatisfied *Interest* should go next. The name index sits ahead of all three and maps an incoming name to the entries that concern it.

These structures are not used in isolation. A single *Interest* threads through them in order, and Figure 4.2 traces that path. On the way in, the index resolves the name and the CS is checked for a cached copy. On a hit the *Data* is returned to the consumer and the exchange ends there. On a miss the PIT records the wait, the FIB chooses the next hop, and the *Interest* is forwarded upstream. On the way back, the returning *Data* is stored in the CS and the pending PIT entry releases it to every consumer that was waiting.

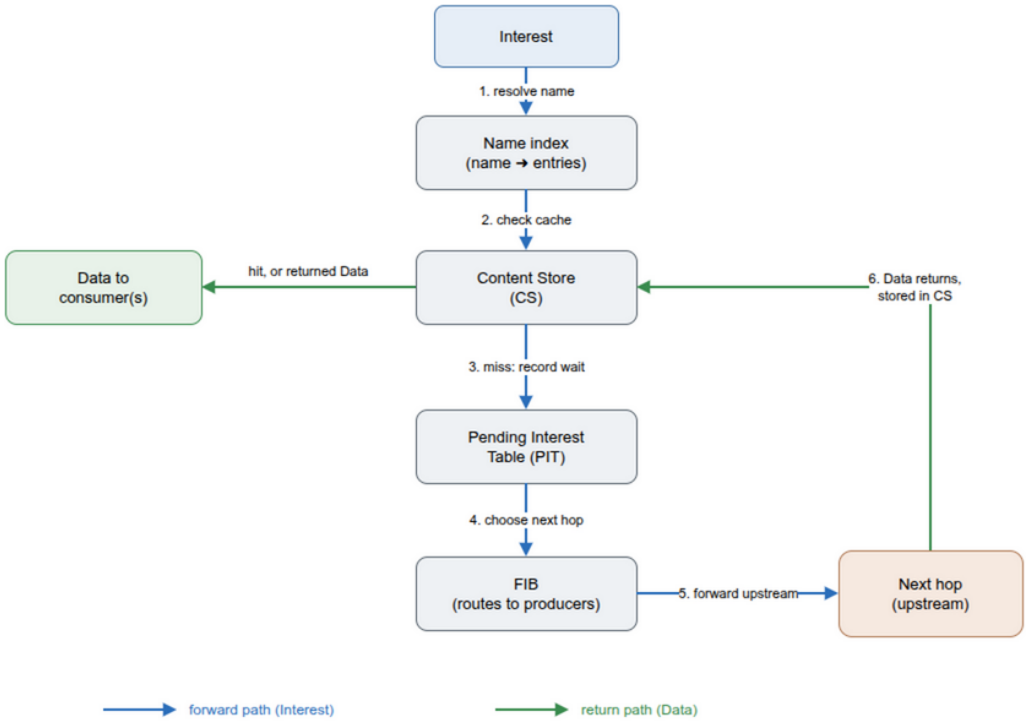


Figure 4.2: The forwarding structures and how one request threads through them.

With the data and the structures that carry it in view, the rest of the chapter turns to security. Each of the steps just described touches a structure at a location that depends on the requested name, so the access pattern that makes forwarding work is also what can betray the request. What we want to protect is exactly this, which names a consumer asks for and whether those requests were served from a cache. The access pattern does not spell out the content of the *Data*, but it does reveal which entries a request touches, and that alone is damaging. An adversary can line those accesses up against informed guesses about popular content and confirm what a consumer is asking for, or with more effort pin the request down directly. The next section takes the structures one at a time and states what is sensitive about each.

4.2. Privacy-sensitive data structures

Each step of the exchange in Section 4.1 reads or writes a structure at a location that depends on the requested name, and that is where the sensitivity lies. The subsections below take the structures one at a time and state, for each, what is sensitive about it and why its access leaks.

Figure 4.3 puts the adversary into that picture. Placing the forwarder in an enclave encrypts its memory, so the contents of the structures are hidden, but the order in which their locations are touched is not. The host watches that order and reads meaning from it. The

slot the index resolves is fixed by the requested name. The branch the CS takes differs for a hit and a miss. The PIT exposes pending and aggregated requests for a name, and the FIB is consulted only when a request misses, so its use marks one. Each subsection below takes one of these structures and states what its access pattern gives away.

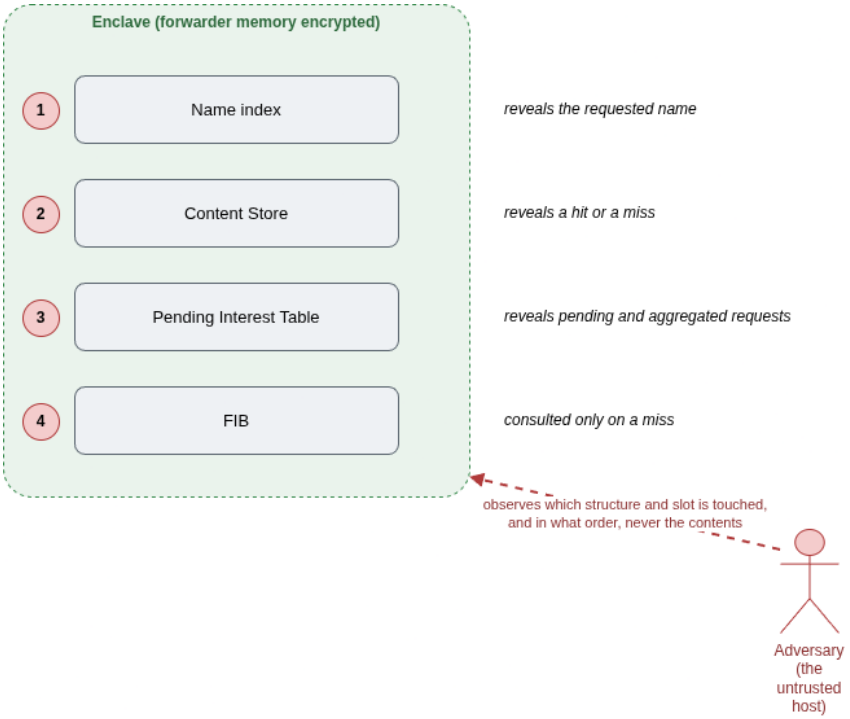


Figure 4.3: The forwarder's structures as the adversary sees them.

4.2.1 Content Store

The CS processes a request by either retrieving it from the cache or forwarding it upstream. This difference is observable, and hits and misses take different paths through the code and memory. An adversary who can distinguish between hits and misses can learn whether a given name was recently requested, which is enough to track a consumer's interest and confirm guesses about past requests. To see this concretely, suppose the attacker wants to know whether a victim has watched a particular video. It issues an *Interest* for that video's name and times the reply. A fast answer means the CS already held the content, so it was fetched through this router recently, while a slow answer means it was not. The attacker confirms the guess without ever reading the cached bytes. This is a cache-privacy problem already studied in named-data and information-centric networks [4, 5]. The leak does not need the cached bytes, because the access pattern and timing of the lookup are enough.

4.2.2 Pending Interest Table

The PIT records *Interests* that are waiting for *Data*. When several consumers request the same name, the forwarder aggregates them into one entry. Thus, the presence, growth, and clearing of a PIT entry reveal relationships between requests and the frequency with which a name is requested. For instance, an entry may already exist for a name and be touched again before its *Data* arrives. An observer then learns that a second consumer asked for the same name while the first request was still pending, and the two requests are tied together without either being read. Monitoring which PIT entry is accessed, and when, reveals the same kind of consumer behavior as the CS, but from the request side rather than the cache side.

4.2.3 Name resolution

Simply protecting the contents of the CS and PIT is not enough. Before either table is read or written, the forwarder uses an index to resolve the incoming name to an entry. A normal index takes a different path for each name, so the access pattern depends on the name being looked up. An adversary who observes this process can recover information about the name, even if the entry itself is hidden. Therefore, the index must also be oblivious, otherwise the name leaks at the lookup even when the entries themselves are protected.

4.2.4 Distinguishing tables and outcomes

The three tables are not used independently. The forwarder links them while processing a single *Interest*, so even if each table is unaware of the others, the path a request takes across them can still leak information. Requests answered from the CS and those that miss are handled differently, because a miss also drives a name-index and FIB lookup to choose a next hop, so a hit and a miss need not produce the same access pattern. Figure 4.4 contrasts the two. A hit touches only the index and the CS and then stops. A miss goes on to record the request in the PIT and consult the FIB before forwarding. The two outcomes therefore differ both in which structures they touch and in how many. An observer who classifies or counts those accesses tells a hit from a miss without reading any content.

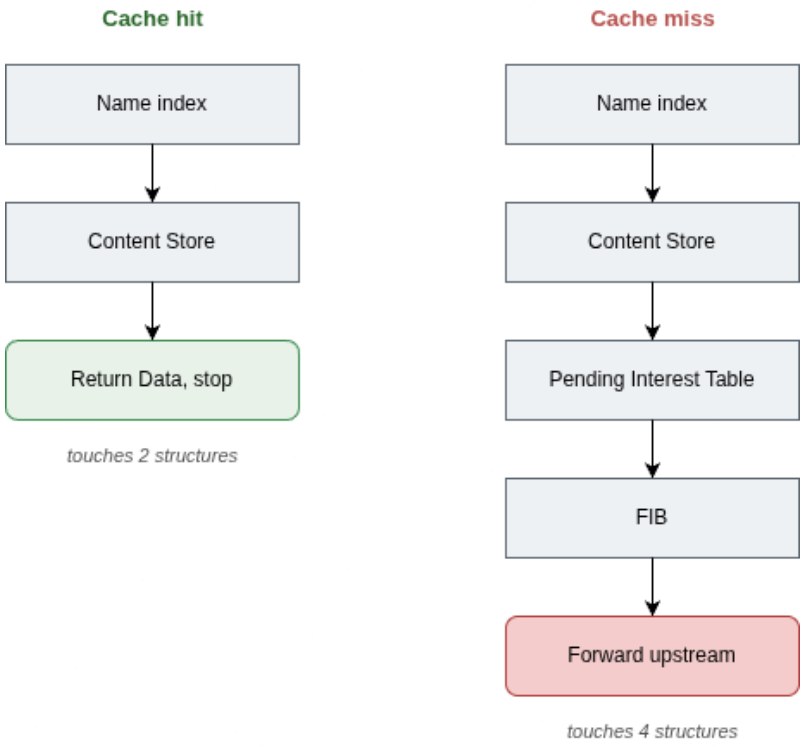


Figure 4.4: Why a cache hit and a miss are distinguishable.

If the adversary can distinguish a CS access from a PIT access, the table identity alone reveals what the request did. Hiding accesses within each table is therefore not enough, and the forwarder must also hide which table an access belongs to and whether a request hit or missed.

4.2.5 Side channels inside the enclave

The leaks described so far assume the host can watch the forwarder's memory accesses. One might expect a hardware enclave to take that assumption away, since it encrypts the forwarder's memory, but it does not hide the access pattern. Cache and page-level side channels still let the host determine which enclave addresses are accessed [18, 19]. An enclave answers "Can the host read the data?" but not "Can the host see how the data is used?". The access-pattern threats above therefore persist inside the enclave, which is why hardware isolation and oblivious access complement each other rather than being alternatives.

4.2.6 Observation and tampering of externalized state

The forwarding state is large, while the enclave's protected memory is limited. Most of the state, the bucket tree that backs the tables, is kept in untrusted host memory. The

host can read these buckets, modify them, and replay stale versions. The externalized state therefore needs confidentiality, integrity, and freshness, not just oblivious access. Bandwidth is also observable, since the number of bytes that cross the boundary per request can itself leak, if it depends on the request.

Table 4.1 collects these leaks, the component each affects, and the channel that carries it.

Target	What leaks	Channel
Content Store	Hit vs miss, requested name	Access pattern, timing
Pending Interest Table	Pending requests, aggregation	Access pattern, timing
Name index	Queried name	Access pattern
Across tables	Which table, hit vs miss	Access pattern
Externalized buckets	Contents, freshness, volume	Host memory, bandwidth

Table 4.1: Summary of threats by forwarding component.

4.3. From the adversary to the requirements

A forwarder may well run on a machine its operator controls. This work assumes it does not, which is the harder case and the one worth defending. The host belongs to someone else, and we treat that host as the adversary.

The host is untrusted to begin with, so its operating system, its hypervisor, and anyone with physical access may all act against the forwarder. Second, the TEE that shelters the forwarder runs on that same untrusted host. Hardware isolation protects the forwarder from the machine it sits on, yet it never turns that machine into a trusted one, which is what makes the problem awkward. Third, what the host can see divides in two. Memory outside the enclave it reads outright, alters, or replaces with a copy it kept from earlier. Memory inside the enclave it cannot read, and even so it still watches which addresses the enclave touches, in what order, and how long each request takes, through the cache and page-fault channels of Section 4.2.5.

Figure 4.5 places that adversary in the network. Traffic runs from a consumer through a chain of forwarding hosts to a producer, and most of those hosts are ordinary servers about which we assume nothing. The dashed circle marks the part that concerns us, a subnetwork whose hosts may be malicious, and our forwarder runs on one of them. On the other host in that circle the adversary has full control and nothing stands in its way. On

ours the TEE puts the forwarder's memory out of reach, and even then the same adversary reads everything on that machine outside the enclave and still watches the pattern of the accesses the enclave makes. The circle only marks where such an attacker might sit. This thesis defends the protected host, and what the adversary sees on the other host or on the wire is the network-level exposure Section 3.3.2 leaves out.

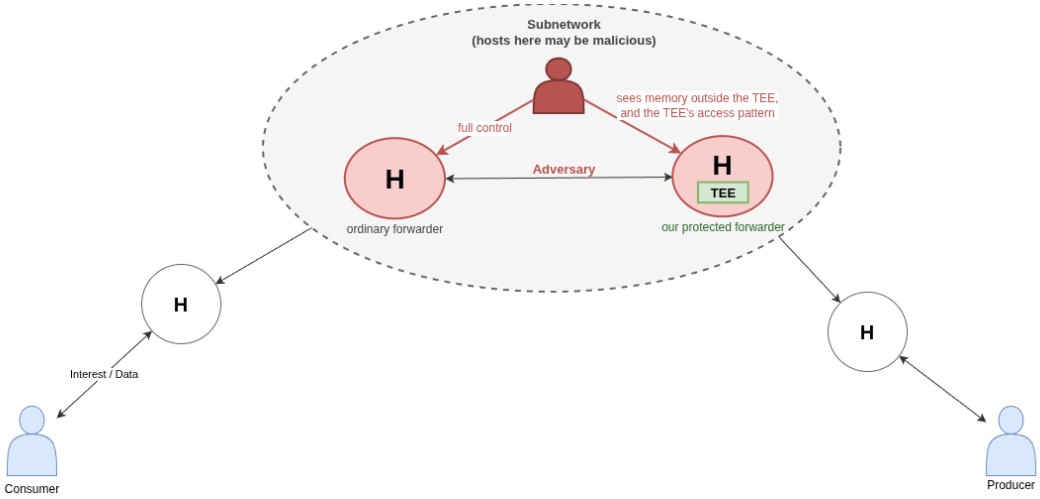


Figure 4.5: The subnetwork the protected forwarder runs in, and what the adversary sees on each host.

An adversary of this kind never needs to break the encryption. It sits underneath the forwarder and reads its behavior instead, which is what Section 4.2 catalogs. A hit told apart from a miss, a name recovered from the shape of an index walk, one table told apart from another, none of it requires reading a single cached byte.

Take the first capability, which is the widest of the three. Because the host reads the pattern and the timing of every access the enclave makes, no part of the forwarder's behavior can be allowed to depend on the request it is serving. Four requirements come out of that single fact. Reads and writes to the CS and the PIT must not reveal which entry they touch, which is **R1 (oblivious table access)**, the job of ORAM, and the answer to the cache and pending-request leaks of Sections 4.2.1 and 4.2.2. The index that turns a name into an entry gives the name away in the same manner, so it too must be oblivious, and that is **R2 (oblivious name resolution)** against Section 4.2.3. These two do not, on their own, give the third. R1 and R2 hide which entry and which name an access touches, but neither hides which *table* the access falls in, nor whether the request stopped at the CS on a hit or ran on through the PIT and FIB on a miss. Making each access oblivious leaves one table distinguishable from another, and a hit from a miss, whenever the tables occupy separate memory or a miss simply does more work. That is what **R3**

(indistinguishable tables and outcomes) requires, and it closes the cross-table leak of Section 4.2.4. R3 hides which tables a request touches. The matching cost has to be hidden too, because a request is not a single access but a run of them, and a run has a length and a duration. If a miss issues more accesses than a hit, or a longer name walks a longer index path, an observer that only counts the accesses or times the request reads the outcome or the name from the cost alone, even when every individual access is oblivious. **R4 (input-independent cost)** requires that this cost, the number of accesses and the time they take, not depend on the request, which in practice means padding every request to a fixed budget of accesses. All four hold inside the enclave as firmly as outside it, because the side channels of Section 4.2.5 hand the host the same view of the memory it cannot read.

The second and third capabilities both bear on the environment the forwarder runs in, and together they ask for a single requirement. The host reaches the state the enclave is too small to keep, the bucket tree behind the tables, which sits in host memory where it can be read, altered, or replaced with a stale copy. The host also controls what code runs on the machine, so a client that opens a connection cannot tell, on its own, our forwarder from a substitute the host has stood up in its place. The forwarding environment must therefore protect both the privacy of the data it holds and the integrity of the process that runs it. Its externalized state stays confidential, tamper-evident, and fresh against rollback, and before its first *Interest* a client can verify that the far end is the genuine code inside a real enclave. That is **R5 (isolated forwarding environment)**, which answers Section 4.2.6 and the code-substitution threat alongside it.

One last requirement answers no leak at all. Whatever the countermeasures do, they must leave the forwarder computing what it computed before and force no change on the protocol or on client software, which is **R6 (behavior preserved)**. Every row of Table 4.1 is claimed by one of these requirements, and the design of Chapter 5 is built to meet them all.

Section 3.3 already fixed what falls outside this work, and the requirements above stay within that scope. What remains is their cost. Making every access oblivious, padding each request to a fixed budget, and encrypting the externalized state all add overhead over a plain forwarder. How much latency and bandwidth the guarantees cost is what Chapter 7 measures.

Chapter 5

Design

Thus far, Chapter 3 fixed the adversary, and Chapter 4 turned the leaks of the forwarding state into six requirements (R1...R6). In this chapter, we first discuss the one decision that shapes all others, then we walk through each of our mechanisms, tying them back to the requirement they answer. This chapter is about design; Chapter 6 discusses code.

5.1. Design overview

Recall that our threat analysis points to two problems. First, even if the contents of a table are hidden, the *pattern* of accesses reveals the nature of the request (R1, R2). Second, the identity of the table being accessed reveals information on its own (R3). Consider the first naive design, where a separate oblivious data structure is used for each table. That is, one ORAM for the CS, one for the PIT, and one oblivious index for each of them. This satisfies R1 and R2, but not R3, since the two access patterns are still distinguishable with separate trees. For instance, a CS hit and a PIT insert are distinguishable patterns even if each tree is accessed in an oblivious manner.

Combining the structures is what answers R3. Instead of using a separate Path ORAM tree for each table, our design uses a single Path ORAM tree to store the entire forwarding state, including the name index. Each block carries a tag, encrypted like the rest of the payload. This tag distinguishes one table from another, and hence the separation exists in ciphertext that the host cannot read, rather than in a region of memory that it can observe. The single tree is accessed for all operations and the blocks are drawn from the same pool. From the host's perspective it is a single stream of blocks, and it cannot distinguish a cache lookup from a pending-interest insert, or a hit from a miss. We call this *domain separation*. The tables are logically separate but use a single oblivious store.

The boundary that this produces is shown in Figure 5.1. There is a narrow, attested gateway at the front. Behind that, in the enclave, are the forwarding logic, the single ORAM client, and a small private state. The majority, the tree of buckets, never enters the enclave at all. It remains out in untrusted host memory as encrypted, integrity-protected blobs.

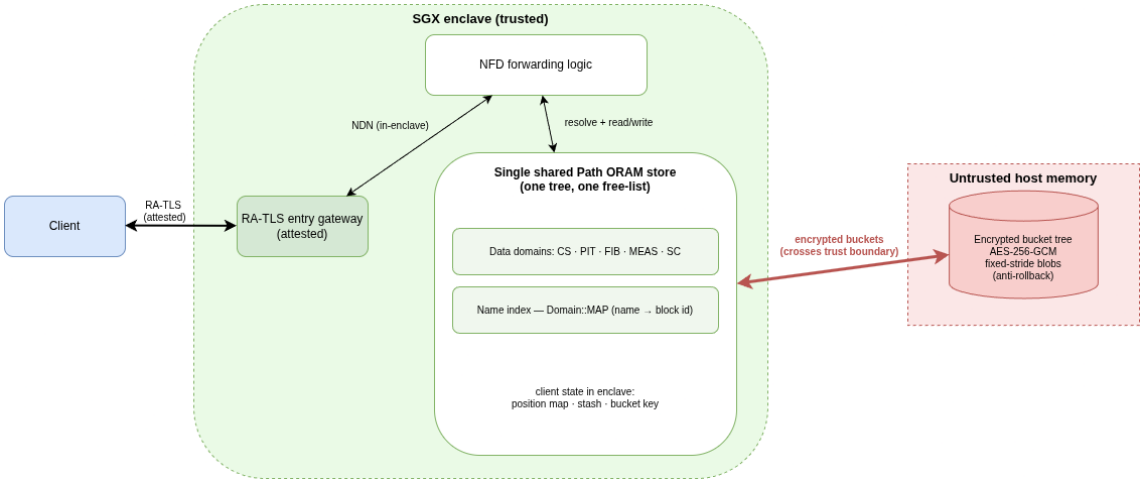


Figure 5.1: Design overview: enclave, single oblivious store, and the boundary with the untrusted host.

Below, we explore the journey of a request. We first navigate to the shared store, which has the data tables and name index as domains of one tree (Section 5.2). The name is looked up obliviously there, so the lookup leaks nothing about the name (Section 5.3). The request is then served end to end without its cost or shape giving away the outcome (Section 5.4), and the tree serving the request remains protected even after it leaves the enclave and resides in the host memory (Section 5.5). We then discuss two points that do not relate to the per-request path. First, we need to attest to the entry point before any requests are handled (Section 5.6). Second, we must show that all this does not change the semantics of what the forwarder does (Section 5.7), which is what lets the design slot in as a modular, non-intrusive replacement rather than a new protocol.

5.2. A single shared oblivious store

At the center is one Path ORAM tree (Section 2.2 in Chapter 2), and every table writes into it. An observer who is given one tree and one free-list, free to count accesses and place them spatially however it likes, cannot associate any of them with a particular table. This section describes how the tables share the tree without colliding and the boundary of what is inside the enclave and what is outside it.

We assume that storage is organized into fixed-size blocks, where the block size is sufficient to contain the largest NDN *Data* packet that the CS may store, 8800 bytes in our prototype (Section 6.2 in Chapter 6 gives the exact parameters). The PIT and index records are far smaller, so they too are stored in blocks of that same size, to ensure that the length of a block does not reveal whether it is storing a PIT record, an index record, or a *Data* record.

Each block is prefixed with a validity flag, a unique identifier, and the domain tag, followed by the content of the table that owns this block. Figure 5.2 illustrates this structure.

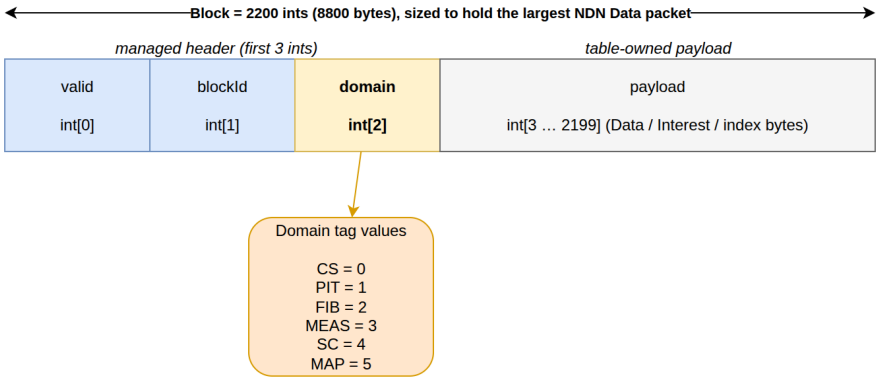


Figure 5.2: Block layout. A validity flag, the identifier, and the domain tag precede the table payload, all inside the encrypted block.

The figure lays the block out in the order it is written: the validity flag first, then the block identifier, then the domain tag, and finally the payload, sized to whatever the owning table needs up to the block limit. The validity flag says whether the block currently holds a live record. A write sets it, and clearing a block, when its entry is evicted and its identifier goes back to the free-list, zeroes the whole block and the flag with it, so a later read of that identifier comes back as a miss instead of returning stale content. All four fields sit behind the same encryption boundary, so what leaves the enclave is one opaque, fixed-length blob; the host cannot tell the flag from the identifier from the tag by looking at the ciphertext, let alone the payload behind them.

The domain tag specifies the logical table to which the block belongs. There are a total of six domains: the CS, the PIT, the FIB, the measurement and strategy-choice tables that NFD maintains, and, finally, the index nodes of Section 5.3. A read specifies the domain it expects and returns the block only on a matching tag, preventing one table from inadvertently reading a block written by another table despite the shared tree. Plus, since the tag is embedded inside the encrypted payload, it never appears in the clear to the host, thereby not aiding in differentiating between the tables.

Regardless of the domain, blocks are drawn from a common free-list. When a table requires storage, it is allocated the next free identifier. When a table entry is evicted, its block is cleared, and the identifier is returned to the pool. Identifiers are drawn from one pool for the same reason the data structure is one tree. Split the identifiers into a separate range per table, and the host could read off which table an access belongs to just from the range

its identifier falls into, then use that to cross-reference every access the table ever makes. One pool closes that: an identifier alone reveals nothing about the table it belongs to, and the allocation pattern of one table looks the same as any other's.

A small private state accompanies the Path ORAM client, namely the position map, which maps each block to its current leaf, the stash, which contains blocks that are in transition between evictions, and the key, which is used to encrypt buckets during eviction. Compared to the data itself, this memory overhead is small, and fits within the enclave's protected memory. The tree of buckets, which stores the blocks, does not fit in the enclave, and is placed in the host memory, where it is protected using the techniques described in Section 5.5. The reason for splitting the store in this manner is that a large forwarding table can be supported using a small enclave, and obliviousness is preserved even with such a split.

5.3. **Oblivious name resolution**

Putting the tables in an oblivious store is not the end of the story. Before a request reaches any table, it must resolve the requested name to an entry, which, as we saw in Section 4.2.3, leaks. This is because an ordinary index walks a different path for each name. The entry that is reached by this walk may be hidden, but the shape of the walk reveals the name. Hence the index, which is in the clear, becomes the place where the name leaks, and it must be made oblivious.

The solution is to store the index alongside the data in a single tree, using an oblivious map keyed by name. The map is an AVL tree, a height-balanced tree whose nodes are simply regular blocks in the shared store under the index domain. Height balance ensures that the tree's depth is at most $1.44 \log_2 n$ [35], where n is the number of entries in the map, which effectively bounds the distance that a lookup travels for any given name. Each node uses block identifiers to name its children as opposed to pointers, so accessing a child node constitutes another ORAM read, with the kernel re-randomizing the path on each access. Therefore, lookup traffic that the host observes is indistinguishable from the random storage traffic it sees for anything else and is unrelated to the name being looked up. This is essentially the ODS of Wang et al. [14], but embedded in our existing tree as opposed to being a separate oblivious data structure.

Two details keep the lookup sound. First, keys are fixed-width digests over (domain, name), so two tables having the same name produce different keys and never collide, which is

domain separation again, but this time inside the index. Second, the walk itself. It always proceeds from the root down a fixed number of levels decided by the height bound, and where the real path is shorter the rest is padded with dummy reads. Be it a hit or a miss, be it one name or the other, the number and shape of ORAM accesses are the same. Listing 5.1 sketches it.

```
lookup(domain, name):
    key  = digest(domain, name)  # from (domain, name)
    node = root
    result = NONE
    repeat MAX_HEIGHT times:     # hit = miss
        block = oram.read(MAP, node)
        if block is real and block.key == key:
            result = block.value # target block id
        node = choose_child(block, key)
    return result
```

Listing 5.1: Oblivious name lookup (sketch). The loop runs a fixed number of times regardless of where, or whether, the key is found.

Listing 5.1 is the whole lookup. The loop always runs `MAX_HEIGHT` times, the height bound fixed above, so its length never depends on where the key sits in the tree or whether it is there at all. Every iteration issues exactly one ORAM read at the current node and one constant-time comparison against the target key. `result` only changes on a real match, and `choose_child` picks the next node without a data-dependent branch, so neither the loop's length nor the comparisons inside it give away anything about the name being resolved.

Figure 5.3 shows that the map shares its tree with the data it indexes. The index node and data block are the same kind of block in the same ORAM, distinguished only by the domain tag.

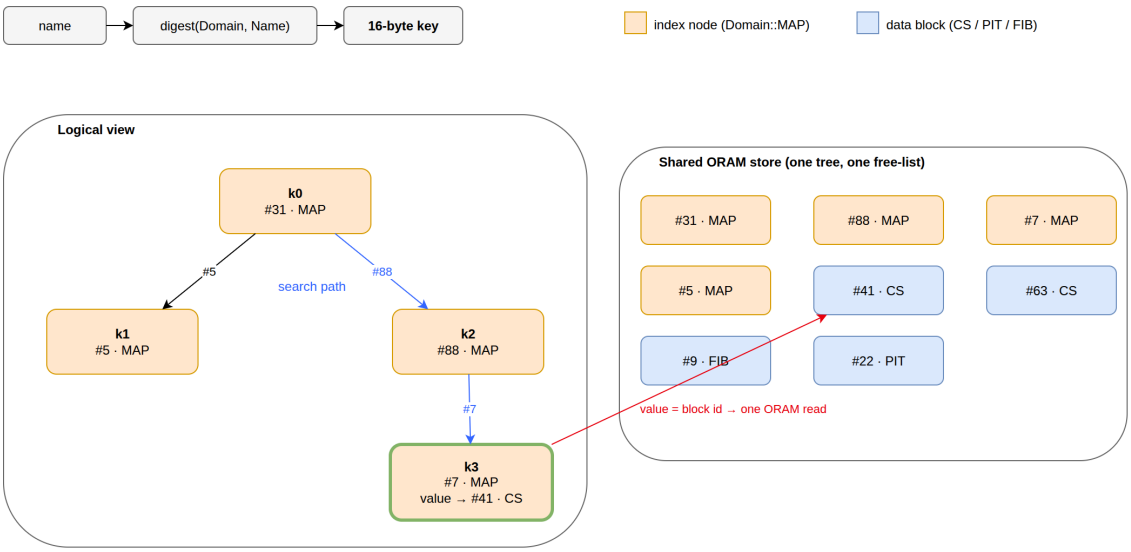


Figure 5.3: The oblivious name index. Nodes are blocks in the shared store, each pointing at its children and a data block by identifier.

5.4. Uniform request handling

A single oblivious access is not the whole story. A request is a short series of accesses, and both their number and their order can depend on what the request turns out to be doing. A cache hit reads one entry and stops there. A cache miss goes on, inserting a pending *Interest* and picking a next hop. Left alone, a hit and a miss walk different numbers of steps, and that difference is visible even when every individual step inside it is oblivious. This section fixes the shape of a request first, then its cost, so that a hit, a miss, a pending-interest insert, and a name-only lookup all look the same from outside.

Shape. Every request, whatever it needs to do, walks the same sequence of steps. It resolves the name through the oblivious index, then touches the CS domain, then the PIT domain, then the FIB domain, always in that order. A domain that has nothing to do for this request is not skipped. It performs its step anyway, against a block of its own kind that carries no real record, what we call a *dummy step*. A miss runs its PIT and FIB steps for real; a hit runs the same two steps as dummies. Each of these steps, dummy or real, is in the end a set of accesses to the oblivious store, and that is what makes the two indistinguishable. An ORAM access reads and rewrites one full root-to-leaf path whichever block it is after, so a dummy access produces exactly the traffic a real one does, by construction rather than by any extra care on the forwarder's part. Whether a given step was real or a dummy is a choice the enclave makes internally, in constant time, so the branch never shows in the memory trace.

Cost. Fixing the order of steps says nothing about how many accesses each step costs, and that is the second thing a request must hide. Two facts keep that number constant. One is that a single ORAM access always costs $O(\log N)$, the height of the tree, regardless of which block it touches. The other is that the number of accesses a request makes is fixed in advance: the forwarder budgets B accesses per request, B being the worst case across the padded index walk and the CS, PIT, and FIB steps that follow it. A request that needs fewer than B real accesses fills out the rest with dummy accesses, the same kind of padding already used for a domain's dummy step, until the count reaches B.

Together, a fixed sequence of domains and a fixed budget of accesses mean that a hit, a miss, and a lookup for any two names produce the same number of accesses, the same bytes crossing the enclave boundary, and close to the same time on the store. The guarantee that held for a single access now covers the whole request. Close, not identical: a hit still copies the entry it found out to its caller, and that copy runs after the oblivious store has done its part, so a small timing difference survives for an outside observer to measure. That residual gap, and the worst-case price every request now pays for uniformity, are both quantified in Chapter 7.

5.5. Protecting externalized state

Recall from Section 5.2 that we pushed the bucket tree out of the enclave and into host memory, which is untrusted. Section 4.2.6 described the adversarial capabilities with respect to host memory, that is, the host can read a bucket, modify it, or return a previously saved copy of the bucket. Obliviousness does not protect against any of these capabilities, because it only hides which bucket an access is directed towards, and not the actual contents of the bucket or whether it is the authentic bucket at all. Thus, the outsourced tree requires more than oblivious access. It requires confidentiality, integrity, and freshness.

A bucket maps to a single AES-256-GCM ciphertext on disk. The key is never out of the enclave and is stored with the other private state described in Section 5.2. A bucket is encrypted right before it is written to disk and decrypted immediately after it is read from disk; the host only ever sees the ciphertext, ensuring confidentiality. The ciphertext encloses blocks, validity flags, and domain tags; the bytes alone do not reveal anything to the host. GCM also produces an authentication tag over the ciphertext, so any bucket that the host attempts to tamper with will fail the tag check when the enclave decrypts it and is rejected, ensuring integrity.

Integrity alone does not prevent replay. A stale bucket is one that the enclave previously wrote to that position, including its tag, and the host can revert the tree by returning that old value where the current value ought to be. To prevent this, each bucket has a version number that the enclave increments each time it writes to that bucket. The version number is bound to the authentication of the bucket as GCM additional authenticated data, so it cannot be moved to another bucket or modified without invalidating the tag. The enclave also keeps track of which version it expects each bucket to have, a small amount of state maintained alongside the position map. If the enclave reads back a bucket with a version number that does not match the expected version number, then the bucket is considered stale and is rejected.

Concretely, every bucket starts at version 0 when the tree is created. Each time the enclave writes to a bucket, whether because a real block landed there during eviction or because the path was touched by a dummy access, it raises that bucket's version by one and remembers the new value in its own state. Suppose bucket b has been written five times and so sits at version 5. If the host now returns the copy of b as it stood after the third write, tagged with version 3, the enclave's expected value of 5 will not match, and the read is rejected before its contents are trusted. A version only ever moves forward, one write at a time, and the enclave is the only party allowed to advance it. The tree cannot be rolled back to any previous state and hence stays fresh.

Figure 5.4 follows a bucket both ways. On the way out, it is encrypted with the enclave key, and both the ciphertext and the version are tagged. On the way in, the enclave decrypts only after it verifies the tag and the version, and it discards the bucket if either of the checks fails.

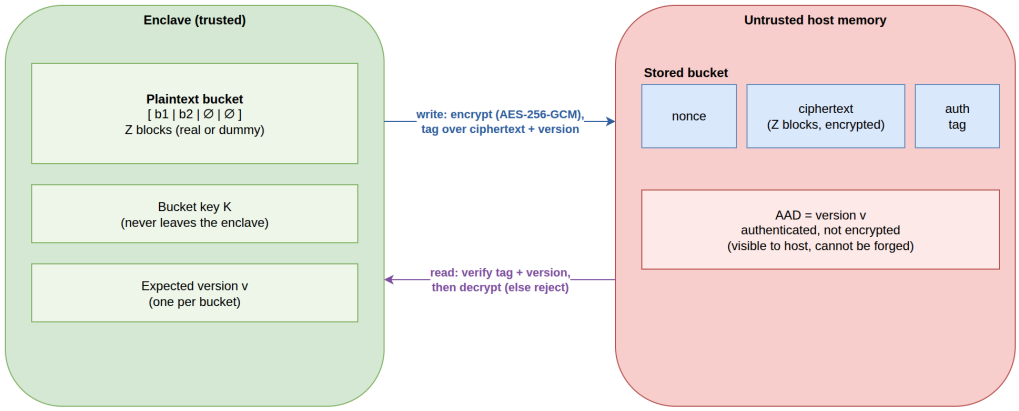


Figure 5.4: Protecting an externalized bucket with AES-256-GCM and a version counter bound as additional authenticated data.

This still leaves the last channel, the amount of data flowing across the boundary. If a request transfers differing amounts of bytes depending on its nature, the host can infer the result directly from the flows. This is not possible here since buckets are of fixed size and each ORAM access reads and writes an entire root-to-leaf path of buckets, thereby equalizing the number of bytes exiting and re-entering the enclave across different requests (Section 5.4). Put the three mechanisms together and the externalized state ends up confidential, tamper-evident, and fresh: the host cannot read a bucket, cannot alter one without the tag failing, and cannot replay an old one without the version check catching it. The fixed size closes the last gap, since a bucket's length never changes with what it holds.

5.6. **Attested entry point**

The above mechanisms provide protection for the forwarding state on an untrusted host. But they do not provide any guarantees to a client that the forwarder is what it believes it to be contacting. Before any request is served, a client must be able to open a secure channel to the enclave itself, not merely to whatever process happens to be listening on the host's socket, which is what this entry point secures. Recall that the enclave executes on the host, and the host controls what is loaded and what listens on the wire. Thus, the host can boot up a modified forwarder that silently logs names, or a simple proxy that forwards traffic while reading it, and a client establishing a merely encrypted connection will see nothing wrong, since the host can terminate that connection at its discretion. A consumer needs to be assured, prior to issuing its first *Interest*, that the entity on the other end is indeed the genuine forwarder, executing the intended code inside a real enclave.

This proof is provided by binding remote attestation to the transport channel, in the RA-TLS style. The gateway that accepts incoming connections from clients executes inside the enclave. During the Transport Layer Security (TLS) handshake, it provides a certificate whose private key was generated inside the enclave, together with an attestation quote signed by the hardware. This quote contains a measurement of the code loaded in the enclave, and a hash of the public key. The client verifies that the quote is signed by the hardware manufacturer, and is thus generated by a genuine enclave, that the measurement corresponds to the forwarder it expects, and that the hash of the public key in the quote matches that of the certificate. Only after these three checks succeed does the client complete the handshake and start sending *Interests*. Figure 5.5 illustrates this exchange, where the enclave provides the certificate and quote, and the client accepts the channel

only after the three checks succeed. The figure shows a client talking to a secure host, but the same exchange applies unchanged to a secure host talking to another host, whether or not that other host is itself secure; the check is on the party being contacted, not on who is doing the contacting.

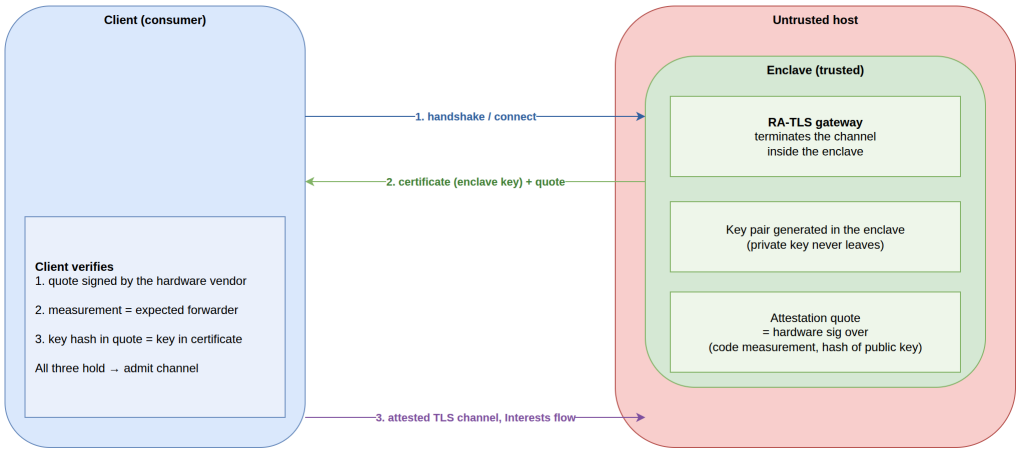


Figure 5.5: Attested entry. The client admits the channel only after checking the enclave's hardware quote and the key bound into it.

Binding the key into the quote is what locks the host out. The private key never leaves the enclave, so the attested channel terminates in the trust boundary, not the host. The host's own key is of no use because it is unable to forge the vendor's signature on a quote. Nor can it slip its key into the enclave's quote, because the measurement would then fail to match. The man-in-the-middle proxy breaks at the handshake rather than reading the traffic in silence. And since attestation is done once, at connection setup, it does not add to the per-request overhead fixed in the previous sections.

5.7. Preserving forwarder behavior

A redesign this deep raises an obvious question: does the forwarder still behave like one? The changes could show up at two levels. One is the forwarding logic inside the process, which now calls into an oblivious store instead of plain tables. The other is the face the forwarder shows the network. This section follows the change from the inside out, and the short answer is that it is absorbed entirely at the first level, as a set of adaptations to how the table operations do their work, and never reaches the second, apart from timing.

Concretely, the internal adaptation is confined to the three table operations. Resolve, insert, and erase keep their name, their arguments, and their return type; only the work done to answer them differs. A plain table walks a pointer-based structure and answers in one touch or a few, while the oblivious store always performs a padded ORAM walk against

the fixed budget B, whether or not there is a real record to find, insert, or remove. Table 5.1 sets the two versions side by side. The caller on the other side of that interface, the CS's eviction logic, the PIT's duplicate-*Interest* handling, the FIB's longest-prefix match, never has to know which of the two answered it.

Operation	Before (plain table)	Now (oblivious store)
resolve(name)	walks a pointer-based index down whichever path the name happens to take	one padded ORAM walk through the shared tree, the same length for every name
insert(entry)	writes the entry into the table's own structure	ORAM write of one block under the table's domain tag
erase(entry)	removes the entry and shrinks the structure	ORAM write of the fixed erase budget B, whether or not an entry actually leaves

Table 5.1: The table interface before and after: the same three operations and signatures, different work underneath.

The forwarding decisions therefore end up where they always did. The longest-prefix match still picks the next hop in the FIB, the PIT still folds *Interests* for a single name and collapses the *Data* coming back, and the CS still serves a repeated request from cache; only the storage beneath those decisions changed, not the decisions themselves.

As a consequence, nothing changes on the wire. Names and *Interest* and *Data* packets keep their formats, and the design introduces no new message types and no forwarding-level handshake. To a consumer or a producer, a protected forwarder answers just as a stock one would. The only new thing a client encounters is the attested TLS channel, used once at connection setup and followed by plain NDN traffic; existing applications work without modification.

One difference does show through: timing. The fixed shape and fixed access budget mean every request pays the worst case, so a protected forwarder answers slower than a stock one even when it gives back the same answer. The effect is on performance, not on behavior.

Chapter 6

Implementation

So far, we have discussed the design of the prototype in Chapter 5. This chapter covers its implementation, that is, how the prototype extends NFD, the reference NDN forwarder, on top of an unmodified ndn-cxx library, so that the oblivious store and the attested gateway run inside a real forwarding daemon and not a model of one. Chapter 5 already motivated why each mechanism is needed.

Here we describe the code that implements it, the parameters it runs with, and the engineering challenges met along the way. The evaluation of Chapter 7 should be read against a transparent account of what was actually built, so this chapter states plainly what has been implemented and which functionalities run only in simulation.

6.1. Prototype

The prototype is a fork of NFD, written in C++17 like the code it extends. The oblivious store and the name index are built into NFD's table subsystem, where they back the CS and the PIT, and the attested gateway is built into the face subsystem through which NFD accepts connections. Forwarding logic, store client, and gateway therefore run in one process and share one address space, exactly as the single-enclave design asks. Only the bucket tree is pushed outside.

We reuse two existing code bases. The first is the forwarder, NFD with its ndn-cxx library. The prototype forks NFD and modifies it directly, in its table subsystem, in its face subsystem where the gateway lives, and in its start-up code, while ndn-cxx is used as released, without changes. The other is the Path ORAM kernel. Here we adopt an open-source C++ implementation [36] that follows the protocol of Stefanov et al. [10]. It is an independent implementation released under the MIT license, not code published by the paper's authors. It provides the block and bucket data types, the read-path-eviction access procedure, the position map, and leaf randomization. What this kernel gives us is ORAM and nothing more, a flat array of blocks whose accesses do not show. It does not give us the ODS that Chapter 2 distinguished from plain ORAM, the oblivious search structure that turns a name

into a block. That layer, the oblivious AVL index of Section 6.3, is our own implementation, written for this work after the construction of Wang et al. [14]. The remaining contributions are likewise built atop the kernel, adding the single shared store that marks blocks with domains and allocates them from one free-list (Section 6.2), the authenticated and anti-rollback externalization of the bucket tree (Section 6.5), constant-time hardening of the client, and the integration into NFD's CS and PIT. The reused kernel required some fixes before it could be shared this way, which we discuss in Section 6.7.

The design targets Intel SGX, which the prototype reaches through Gramine. The VM-level Intel TDX was discussed in Chapter 2 as the alternative TEE. This work does not use it and commits to SGX, whose process-level model fits a design that isolates a single forwarder process. For convenience of implementation, the prototype runs SGX in simulation mode rather than on SGX-enabled hardware. Gramine supports this through its `gramine-direct` loader, which runs the very binary and manifest a hardware enclave would load, only without the hardware behind them. Everything the design functionally relies on is exercised this way. The forwarder runs as one process, its sockets never leave that process, and the bucket tree is served from an ordinary file in untrusted memory. What simulation mode does not provide are the hardware guarantees, memory encryption and a real attestation quote among them, and throughout this chapter we mark every place where this distinction bites.

Figure 6.1 depicts the deployed prototype. The NFD process contains the forwarding logic, the oblivious store client with its own private state, and the gateway. The bucket tree is an authenticated ciphertext file residing in untrusted host memory, and an external client connects to the forwarder only via the attested channel terminated by the gateway.

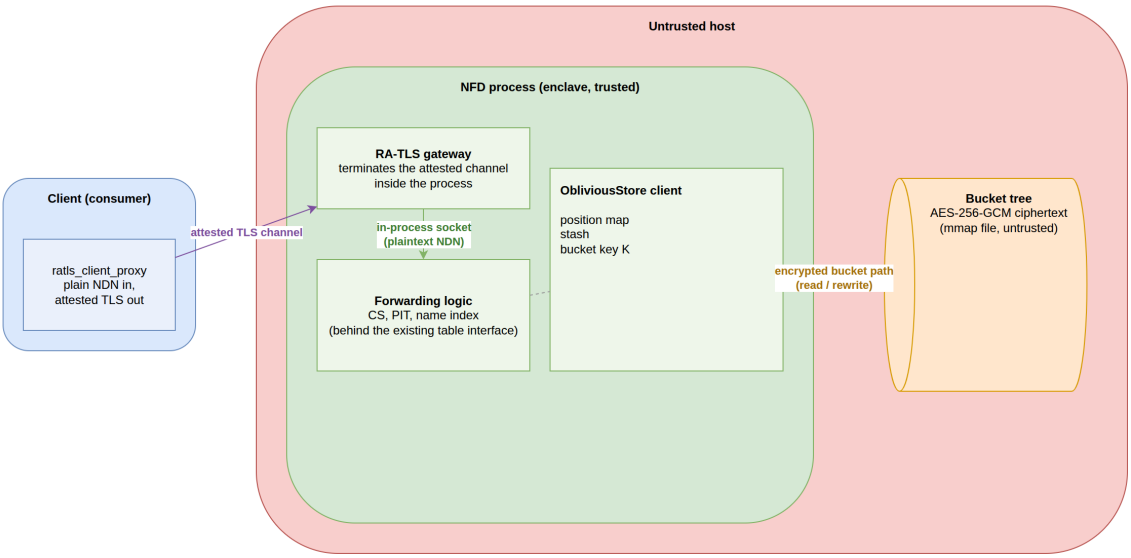


Figure 6.1: The deployed prototype.

The six requirements of Chapter 4 do not map one-to-one onto files. Each is a property of the system as a whole, and the sections of this chapter carry them jointly. Still, each has a place in the code where the work concentrates, and naming that place up front gives the chapter its map.

R1 (oblivious table access) is carried by the shared store in `oram/oblivious-store.{hpp,cpp}`, which routes every table access through the Path ORAM kernel in `oram/OramReadPathEviction.cpp`. Any read or write becomes a full path read with eviction, so which entry was touched never shows in the traffic (Section 6.2).

R2 (oblivious name resolution) lives in `oram/oblivious-map.{hpp,cpp}`, the AVL index kept over the same tree. Resolving a name costs a fixed walk of oblivious reads instead of a plaintext tree descent (Section 6.3).

R3 (indistinguishable tables and outcomes) is what the sharing buys. The store keeps one tree and one free-list for all its domains and tags each block inside its ciphertext, so no access betrays the table it serves, and on the outcome side the CS erases through a conditional erase in `table/cs.cpp` that runs its full budget whether or not it removes anything (Sections 6.2 and 6.4).

R4 (input-independent cost) rests on padding and on the constant-time primitives of `oram/ObliviousOps.h`. Index walks pad to a fixed height, updates pad to fixed budgets, and the in-enclave comparisons those walks make are branchless (Section 6.3).

R5 (isolated forwarding environment) spans the code at the enclave boundary. `oram/`

`ServerStorage.{h,cpp}` together with `oram/aes-gcm.{hpp,cpp}` keeps the externalized tree encrypted, authenticated, and protected against rollback, and the RA-TLS gateway under `face/` makes the attested channel the only way in (Sections 6.5 and 6.6).

R6 (behavior preserved) is enforced where the store meets NFD, in `table/cs.cpp` and `table/pit.cpp`. Both tables keep the interfaces they have always exposed, so the pipeline above them runs unchanged (Section 6.4). The evidence that behavior is preserved comes in Chapter 7, where the oblivious store and a plain one, driven by the same request stream, record the same hits and the same misses at every cache size.

Table 6.1 lists the parameters the prototype runs with. The base system is NFD and ndn-cxx in C++17, and the oblivious kernel is Path ORAM with read-path eviction. Buckets hold $Z = 4$ blocks of 8800 bytes each, of which a three-integer header records validity, block identifier, and domain. Six domains share the store, and the tree is dimensioned for 29824 blocks, which the geometry of Chapter 2 turns into 16 levels and 65535 buckets. Buckets leave the enclave under AES-256-GCM with a 12-byte nonce and a 16-byte tag, and each carries a 64-bit version bound into the authentication against rollback. The enclave itself retains only the position map, the stash, and the bucket key. The runtime is Gramine with SGX in simulation mode, as discussed above. The sections that follow give each of these values its context.

Parameter	Value
Base system	NFD and ndn-cxx, C++17
Oblivious kernel	Path ORAM with read-path eviction
Bucket size Z	4 blocks
Block size	8800 bytes (2200 32-bit integers)
Domains	6 (CS, PIT, FIB, measurement, strategy choice, index)
Shared store capacity	29824 blocks (CS 4096, PIT 1024, CS index 8192, name-tree index 16384, slack 128)
Tree geometry	16 levels, 65535 buckets
Bucket cipher	AES-256-GCM (OpenSSL), 12-byte nonce, 16-byte tag
Anti-rollback	64-bit per-bucket version, authenticated with the position
Enclave-resident state	position map, stash, bucket key
Runtime	Gramine, SGX simulation mode via `gramine-direct`

Table 6.1: Parameters the prototype runs with.

6.2. The shared oblivious store

A process-global object, `ObliviousStore`, encapsulates the reused Path ORAM kernel so that a single tree is used for all tables. This object is instantiated once, on demand, and is shared by the CS, the PIT, and the name index, ultimately leaving a single tree and a single free-list behind all of them. This is the single decision from Chapter 5 realized in code, and it allows the host to see a single stream of accesses instead of a stream per table. Earlier versions of this prototype, built during this work before the tables were merged, kept a separate Path ORAM tree inside each of `Cs` and `Pit`, but combining these into the shared object makes the access patterns of the two tables indistinguishable, and this change led to the discovery of the shared-state bugs from Section 6.7.

Each table reaches the store through a small interface keyed by a domain tag (Listing 6.1). The caller gets one block identifier out of the shared pool, writes a record keyed by its own domain into the block, and later reads that record back using its own domain. The store writes the domain into the block on a write and checks the domain on a read, returning a miss if the stored tag and the requested one differ. The tag is embedded in the block, and

the block is encrypted before it leaves the enclave, so the host never knows which domain an access belongs to.

```
enum Domain { CS, PIT, FIB, MEAS, SC, MAP };

int  alloc();                       // take a block id from the shared pool
void free(int id);                  // return an id to the shared pool

bool write(Domain d, int id, bytes wire);   // stamp block id with domain d
optional<Block> read(Domain d, int id);     // miss on a domain-tag mismatch
```

Listing 6.1: The domain-tagged store interface. All six domains share one tree and one free-list; the domain tag is written inside the block and checked on read.

The block is a contiguous memory region of 8800 bytes (2200 32-bit integers). The first three integers open it as a header, validity flag, block identifier, and domain tag. The rest of the block is the payload, the structure of which is dictated by the owning table, and it carries an NDN *Data* packet for the CS, an encoded *Interest* for the PIT, or a serialized index node for the name index. Since the block is sized to accommodate the largest *Data* packet that the CS might cache, the much smaller PIT and index records fit in the same footprint, and the size of a block never indicates what kind of record it stores. Six domains use it: the CS, the PIT, the FIB, the measurement and strategy-choice tables that NFD keeps, and the index domain of Section 6.3. In the prototype three of them hold blocks. The CS stores its *Data* packets, the PIT its encoded *Interests*, and the index its nodes. The FIB, measurement, and strategy-choice tags are reserved in the block format, but those tables keep their records in the ordinary memory of NFD. What is oblivious for them is the name resolution, since the name tree that serves them resolves names through the oblivious index, and that resolution is the FIB step of Section 5.4. Their contents change with routing and configuration rather than with each consumer request, which is why Chapter 3 leaves routing decisions out of scope, and why the rest of this chapter and Chapter 7 speak of the CS and the PIT.

The free-list is a shared pool of block identifiers. When a table needs storage, it is allocated the next identifier from this pool using `alloc`. Removal of an entry from the table clears the block and returns the identifier to the pool using `free`. There is only one pool for the same reason that there is only one tree. An identifier reveals nothing about the table it is used by, so the allocation activity of one table resembles that of any other.

For capacity the tree is built to hold 29824 blocks, the sum of what each structure can need at most: 4096 CS data blocks, 1024 PIT blocks, two CS index nodes per cached entry (8192), since the CS keeps one index by full name and one by plain name, 16384 nodes for the name-tree index, and 128 blocks of slack for the padding nodes of each structure. The name-tree budget is the largest because NFD keeps a name-tree entry for every prefix of every name in use, not one per name. The 4096-entry CS is a modest cache, about 36 megabytes of *Data* at the full block size, chosen to keep the tree at 16 levels. Chapter 7 shows that each further doubling of the tree adds one level and a near-constant slice of latency, so a larger cache is a matter of budget rather than of design. Four blocks to a bucket, and the geometry Chapter 2 described, one leaf for each block and so $2^{15} = 32768$ leaves for 29824 blocks, yields a tree of 16 levels and 65535 buckets. The CS budget of 4096 blocks and the PIT budget of 1024 are budgets within this one shared tree, not separate trees of their own. Little of the store stays inside the enclave, only the position map recording the leaf each block is assigned to, the stash of blocks in transit between evictions, and the key that encrypts buckets. The buckets themselves are the bulk, and they are pushed out to host memory, where Section 6.5 covers how they are protected.

6.3. The oblivious name index

The name index is implemented as `ObliviousMap`, an ODS-style AVL tree whose nodes are ordinary blocks in the shared store under the index domain. It stands in, obliviously, for the plaintext name lookup NFD uses to turn a name into a table entry. A node holds its children by block identifier rather than by memory pointer, which makes descending one level a single oblivious read, and the Path ORAM kernel re-randomizes the path at every access. What the host sees for a lookup is therefore the same random storage traffic it sees for any other access, with no dependence on the name queried. An AVL tree bounds its height at about $1.44 \log_2 n$, so a lookup pads to that height and always performs the same number of oblivious reads. An insert or erase pads likewise, to a fixed access budget that already covers whatever rotations it triggers to rebalance the tree, and neither the key nor the current shape of the tree ends up showing in the cost. The construction follows the oblivious data structure of Wang et al. [14] from Chapter 2, reusing the one tree rather than standing up a second oblivious structure.

Keys are fixed 16-byte digests over the pair of a domain and a name. Give two tables the same name and they still come out with distinct keys, so their index entries never collide.

Domain separation once more, this time applied inside the index rather than to the data blocks.

A lookup always walks from the root down a fixed number of levels set by a height bound, reading a spare index block to fill out the walk wherever the real path falls short. Hit or miss, one name or another, the number and shape of accesses come out the same, which is what a uniform request cost needs from the lookup. Insert and erase pad in the same manner, up to fixed access budgets, so their cost too is independent of the key and of the tree's current shape. On the CS eviction path a variant, `eraseConditional`, runs the full erase budget whether or not it removes an entry, keeping the access pattern silent about whether a removal actually happened. Section 6.4 explains why the CS needs that.

The comparisons and the child-selection that run inside the enclave on these walks are written with constant-time primitives, gathered in `ObliviousOps.h`. They provide branchless equality and less-than masks, a branchless conditional select, and a conditional memory copy that touches every byte no matter what. The aim is to strip data-dependent control flow out at the source level, so the memory trace of a lookup never forks on the key or the outcome. How far this holds once the code is compiled is a limit Section 6.8 records.

6.4. Integrating the Content Store and the PIT

Underneath NFD's existing `Cs` and `Pit` classes the store sits, behind the same operations each already exposes. Above them the forwarding pipeline goes unchanged, which is just what preserving behavior asks for. What changes is where a table keeps its records and how it reaches them, never what the forwarder computes from them.

Within the CS, every insert, lookup, eviction, and erase turns into an oblivious access. Two pieces of per-entry metadata that NFD would ordinarily keep in plaintext beside the index move inside the CS block, the recency counter that drives least-recently-used eviction and the freshness deadline a `MustBeFresh` *Interest* is checked against. With both inside the block, reading recency on a cache hit and testing freshness become ordinary oblivious reads rather than plaintext touches that would leak which entry was used.

Lookups run through two oblivious maps over the shared tree, one keyed by the full name for queries carrying an implicit digest and one keyed by the plain name for exact queries, so both NDN lookup forms are answered obliviously. This preserves the exact-match and freshness semantics NDN defines. Unbounded longest-prefix match over names is the exception, not made oblivious and left as a limitation that Section 6.8 discusses.

Eviction picks its victim by a data-independent scan for the minimum recency, then issues the name-index erase unconditionally through `eraseConditional`, committing the removal only when the chosen entry still owns its block. Whatever entry leaves the cache, removing the victim costs the same and looks the same, so eviction gives away nothing about what was dropped.

In the PIT, a fresh *Interest* is encoded to its NDN wire form and written to the store under the PIT domain, and the block identifier that comes back is attached to the in-memory PIT entry. Should a duplicate *Interest* arrive for a name already pending, the block is read back, decoded, and the entry refreshed. A satisfied *Interest* clears its block with a zero write and gives the identifier back to the shared pool. And when the pool runs dry, the forwarder logs a warning and drops the *Interest* rather than failing outright, so exhausting oblivious storage degrades service instead of crashing the daemon.

6.5. **Externalizing the bucket tree**

The bucket tree is the bulk of the store and will not fit in the enclave, so it lives outside in untrusted host memory and must be protected there, the concern Chapter 5 raised for externalized state. `ServerStorage`, the component that owns it, keeps the tree as a flat region of fixed-stride slots backed by a memory-mapped file. One bucket fills each slot, laid out as a 12-byte nonce, then the ciphertext, then a 16-byte authentication tag. A slot's plaintext is always a full serialization of Z blocks, real blocks first and dummies for the remainder, so every slot runs to the same length, and that length never betrays how many real blocks a bucket holds.

Each bucket is encrypted with AES-256-GCM through OpenSSL, not through any hand-written cipher. Every write draws a fresh random nonce, so rewriting a path with identical contents still yields unlinkable ciphertext, as Path ORAM requires. Confidentiality comes from this, the blocks and their tags sitting inside the encrypted payload, and so does integrity, a bucket the host has altered failing its authentication check when the enclave reads it back.

Integrity by itself does not stop replay, since a stale bucket is one the enclave wrote and authenticated earlier. To catch a rolled-back bucket, every slot carries a monotonic version number that the enclave raises on each write and keeps to itself, about eight bytes a bucket alongside the position map. On a write, the slot's position and its new version are bound into the authentication as additional authenticated data. Replay a bucket, roll it back, or

relocate it, and authentication fails on the next read, because the stored version no longer matches the position and version the enclave expects. Here is the freshness guarantee that authenticated encryption on its own does not provide. Listing 6.2 sketches the two paths. A write bumps the slot's version, draws a fresh nonce, and seals the serialized bucket with the position and new version bound into the tag. A read reverses this, and the decryption fails if the slot was altered, relocated, or rolled back.

```
write(pos, bucket):
    version[pos] += 1                    // counter stays in the enclave
    nonce = random(12 bytes)
    ct, tag = AES-256-GCM-seal(key, nonce, serialize(bucket),
                               aad = (pos, version[pos]))
    file[pos] = nonce || ct || tag       // fixed stride, same length
    always

read(pos):
    nonce, ct, tag = file[pos]
    plain = AES-256-GCM-open(key, nonce, ct, tag,
                             aad = (pos, version[pos]))
    // fails here if the slot was altered, moved, or rolled back
    return deserialize(plain)
```

Listing 6.2: The externalized write and read paths.

The key is a fresh random value, generated inside the enclave on each run and neither written out nor persisted. An ephemeral key suits a volatile cache, and it throws in cross-restart freshness for nothing. Ciphertext left from a previous run is unreadable under the new key, so the host cannot wind the tree back to a state from before a restart. Carrying the tree across restarts would instead demand a sealed, hardware-derived key, and the per-bucket version counters would have to be sealed along with it, since zeroing them on restart under a stable key would re-open the very rollback they were added to close.

Where the untrusted-memory assumption becomes concrete is the backing file. By default it is an anonymous file under `/tmp`, unlinked the moment it is mapped so the kernel reclaims it even if the process is killed, and its path can be overridden for deployment. Under real SGX it has to be a host-visible path that the Gramine manifest lists as allowed, which puts it outside the enclave just as the threat model intends. In simulation mode it is an ordinary file. The protection it carries, encryption and anti-rollback both, is identical either

way, applied by the enclave code before a single byte is written.

6.6. The attested entry gateway

The measures so far protect the forwarding state, yet they still leave a client unable to tell a genuine forwarder from a proxy the host has slipped into its place. Open an ordinary encrypted connection and the client notices nothing, the host being free to terminate that connection itself. Closing this is the gateway's job, done by binding remote attestation to the transport channel in the RA-TLS style of Chapter 5.

Inside the NFD process the gateway runs, in front of NFD's local face, switched on by an environment variable and built with a dedicated configure flag. A client reaches it through a small local proxy. That proxy speaks plain NDN to the application and an attested TLS session to the gateway, and the gateway relays the decrypted NDN on to NFD's own socket. Gateway and NFD run in one Gramine process, and Gramine emulates that socket inside the enclave, so the decrypted NDN never leaves the process at all. The attested certificate is minted by the official `gramine-ratls` tool, which lets the gateway load a certificate and key instead of carrying attestation code of its own, and the client checks the SGX quote against the expected code measurement before any traffic flows.

This gateway is where the prototype and a real-SGX deployment part ways most sharply. Its encryption half runs fully in simulation mode, so the channel is established, the traffic is carried, and the relay into the in-process socket works. Genuine attestation is the half that does not run, a real hardware quote and the client's verification of it, both of which want SGX hardware. Simulation mode mints no hardware quote, so the gateway falls back to a self-signed certificate, which leaves the transport in place while the hardware root of trust is absent. On attested hardware the same code path presents a real quote instead, and that is the one change the design intends between the prototype and a deployment on real SGX hardware.

6.7. Correctness and cost of sharing the tree

Merging the two former per-table trees into a single shared store surfaced a class of bugs in the reused kernel that the single-instance upstream had never exercised, every one of them shared state that ought to have been per-instance. They are worth recording, being easy to reintroduce and hard to spot, for the code compiles and each table works fine on its own.

The first two shared a shape. In the bucket type, the initialization flag and the maximum size had been declared `static`, so one setting served every bucket in the process. Harmless under one tree. But with the CS and PIT trees now sharing a process, one's settings overwrote the other's. The random-leaf generator had the identical flaw in its leaf bound, so the CS setting its bound to 4096 leaves corrupted the PIT's bound of 1024 and threw out-of-range leaf indices and crashes. Listing 6.3 shows the pattern and its fix, which is simply to make the fields per-instance so that each object owns its own state.

```
// upstream Bucket: one copy of the state for the whole process
static bool is_init;
static int  max_size;

// fixed Bucket: each bucket carries its own state
bool is_init;
int  max_size;
```

Listing 6.3: The shared-static-state bug and its fix.

The third bug lived in the oblivious eviction. Constant-time code encodes a comparison outcome as a mask of all zeros or all ones, so that no branch is ever taken on it. One such mask was being added straight to the counter that tracks how many blocks a bucket already holds. Read as a number, an all-ones 32-bit mask is -1, so the counter ran backwards and the "has room" test never passed again. Listing 6.4 shows the offending line and its correction, which shifts the mask's top bit down to a plain zero or one before adding it.

```
uint32_t take = ct_eq(slot, wanted);  // 0x00000000 or 0xFFFFFFFF

counter += take;          // upstream: adds -1 when the mask is all ones
counter += (take >> 31);  // fixed: adds exactly 0 or 1
```

Listing 6.4: The mask-arithmetic bug and its fix.

Two changes went the other way, cutting the fixed cost of an index operation while leaving everything the host observes untouched. The rebalancing the index performs after an insert or an erase was rewritten to reuse nodes already held in enclave memory rather than read them back from the store, which lowers the fixed access budget of both operations. And each index is now sized to the number of entries it can really hold, not to the whole store, which lowers its height bound and with it the reads per lookup. Both keep the

per-operation cost fixed and key-independent, so obliviousness survives even as it grows cheaper. Chapter 7 reports the budgets they leave.

6.8. Limitations

A set of limitations is left open by the prototype, and naming them precisely matters both for reading the evaluation and for judging what a production build would still need.

Exact-match and freshness queries the name index answers obliviously, but NDN's unbounded longest-prefix match over names it does not. To see where this bites, take an *Interest* for `/a/b` carrying the `CanBePrefix` flag, which NDN lets a cached *Data* named `/a/b/v3/seg0` satisfy. Finding that match means the index has to weigh every stored name under `/a/b`, so the nodes it visits grow in number with how many names share the prefix. Each single visit is oblivious. Their count is not, and that count tracks the prefix population, so how many names share a prefix leaks even though no individual name does. Protecting against this leakage would mean padding every prefix query to the size of the largest sub-branch the index could hold, and that padding has natural performance implications, since every query would then pay the worst case. The prototype does not implement it, and oblivious prefix search is left as future work.

The same problem has a narrower relative on the path from a PIT entry to its next hop. The name index resolves a name to a name-tree node, and the prototype made that resolution oblivious by probing every legal depth in fixed order, padding the probes that fall past the name's length. One caller escapes that treatment. The name tree indexes names only up to a fixed number of components, and it does not index the implicit digest that can close a name. A PIT entry whose name is longer than that limit, or ends in such a digest, therefore resolves to the name-tree node of a shorter prefix of its name, not of the name itself. In that case the forwarder reaches the rest of the name by walking the remaining components one at a time, stopping at the first that is not stored. Take an *Interest* for `/a/b/c/d` whose name-tree node only reaches `/a/b`. The walk probes `/a/b/c`, then `/a/b/c/d`, and had `/a/b/c` been absent it would have stopped after one. Two requests that differ in how far their names run therefore differ in how many probes they trigger, and the count is visible even though each probe is not. What leaks is the depth of the name, not its content, and only for names in that deeper class. Extending the fixed-probe treatment to this caller would close it. It was left open because of scope, not because closing it is hard.

The Path ORAM kernel keeps its position map as a plain in-enclave array indexed by block identifier, and that map is not itself oblivious. Nothing in the construction makes it so. What Path ORAM randomizes is where a block sits in the tree, a fresh random leaf after every access, and that hides the access pattern over the tree in host memory. The map recording those leaves is read at slot i whenever block i is accessed, always the same slot for the same block. Against the host this is harmless, the map never leaving the enclave. Against the stronger observer of Chapter 4, who watches memory-access patterns inside the enclave through a microarchitectural side channel, it is a leak. Take two *Interests* arriving for the same cached name. Each resolves to the same block identifier and thus reads the position map at the same slot, though the block itself is fetched from a fresh random leaf every time. Watch which slot is read and you learn that the two requests asked for the same content, without ever seeing a leaf or a block. Closing this calls for a recursive ORAM, which stores the position map itself in a smaller ORAM accessed obliviously in turn, and that is a separate construction, not a parameter change.

The constant-time primitives take data-dependent control flow out at the source level, yet the language makes no promise that the compiled code stays branch-free once the optimizer has been through it. A build meant for real TEE deployment has to audit the generated assembly, or pin the sensitive operations down with inline assembly, and some of the in-enclave comparisons on the index path are still waiting to be hardened this way.

Attestation, finally, runs only in simulation mode, as Section 6.6 laid out. The gateway's encrypted channel is fully exercised, but the certificate behind it is self-signed rather than backed by a hardware quote, so the client-side verification of a real quote is code the prototype carries and never executes. A deployment on SGX hardware would exercise that path unchanged, and until one runs, the attestation half of the design is implemented but not verified end to end.

One last limitation is cost rather than leakage. The fixed 8800-byte block, sized to the largest NDN *Data* packet, has every slot reserve a full block whatever it actually stores, with every access reading and rewriting a path of such blocks. Sizing the block to the median *Data* packet instead would cut both latency and memory and weaken nothing, at the price of a scheme for the rare packet larger than the block, split across blocks or stored aside. Chapter 7 measures how much of the cost this fixed size accounts for.

Chapter 7

Evaluation

Two things separate the prototype from the plain forwarder it forked, and both carry a cost. The forwarder now runs inside a TEE, which charges for entering and leaving the enclave and for the protection of its memory. And its tables now sit behind ORAM, whose overhead is logarithmic in theory but hard to quantify with precision in practice, since tree height, block size, and access budgets decide what a request actually pays. This chapter measures that cost. Obliviousness itself is not measured, because it follows from the Path ORAM construction of Chapter 2, and a running system either has it or does not. The measurements set the oblivious store against an ordinary non-oblivious one doing the same work under the same workload. They cover the cost of a single oblivious access as the tree grows, the cost of each store operation in time and in bytes, the throughput, hit ratio, and memory of a working cache, and the occupancy of the in-enclave stash. The TEE half of the price is not measured directly, since the runs are native, and the discussion covers what it would add. Section 7.1 lays out the environment and the method, Section 7.2 reports the results, Section 7.3 analyzes the stash, and Section 7.4 weighs what bounds the numbers.

7.1. Experimental setup

The experiments drive the forwarder's content store directly, with no network around it. This section first states what runs and how the pieces connect, then the procedure and everything it measures.

7.1.1 Environment

Everything runs on a single machine, in one benchmark process on one thread. There are no separate consumer or producer processes and no second host. No network is emulated, no faces or links exist, and neither the PIT nor the FIB takes part. The benchmark links the forwarder's content-store code directly and calls it through the same C++ interface the forwarding pipeline calls it through, so what is measured is the store rather than a forwarder in traffic.

The software under test is the prototype of Chapter 6, the fork of NFD 24.07 built on ndn-cxx 0.9.0 in C++17. Its oblivious kernel is the adapted open-source Path ORAM implementation [36], and buckets are encrypted through OpenSSL 3.5.5. The prototype's runtime is Gramine 1.9 with SGX in simulation mode, but the measurements themselves run natively, outside Gramine. This splits what the numbers mean. The oblivious accesses an operation issues and the bytes it moves are properties of the construction and read the same on any platform, while the wall-clock figures are native ones, and Section 7.4 states what real SGX would add to them. Table 7.1 lists the machine and the toolchain.

Hardware	
Processor	Intel Core i7-14700HX, up to 5.5 GHz
Cores and threads	20 cores, 28 threads
Memory	30 GB
Software	
Operating system	Ubuntu 26.04 LTS
Kernel	Linux 7.0.0-27
Compiler	g++ 15.2.0
Cryptography	OpenSSL 3.5.5

Table 7.1: Machine and toolchain the measurements were taken on.

The comparison runs two content stores behind the same interface. The oblivious one is the real NFD `Cs`, its name indexes and data blocks held over the shared Path ORAM store, so a lookup or an insert is the very operation the forwarder performs on the production code path. The baseline is a plain non-oblivious store, an `unordered_map` with an $O(1)$ eviction queue, keyed by the same 128-bit name digest and exposing the same `find`, `insert`, and `erase`. The stock NFD `Cs` could not serve as the baseline because the forwarder was made oblivious in place, a caveat Section 7.4 returns to. The objects moving through both stores are genuine NDN *Data* packets, named, given a freshness period, signed with a key chain, and wire-encoded, exactly as they would be on a real face. They are simply never put on a wire.

7.1.2 Method

A typical NDN evaluation runs a whole forwarder over a simulated or emulated topology, with consumer and producer applications driving traffic across real faces, and reads end-to-end figures such as delay and cache hit ratio [37, 38]. The harness here sits deliberately below that level. Driving the store alone is what lets the cost of one operation be attributed

exactly, with nothing of the surrounding pipeline mixed in, and the end-to-end run inside NFD is left to future work. Concretely, the benchmark links the `Cs` class of NFD, the one the forwarding pipeline calls, together with the shared store and the name index underneath it, and calls it from a dedicated test program instead of through the pipeline. What is validated is therefore the oblivious store and its integration into the table code of NFD, not the forwarder running inside the enclave, and the PIT and FIB take no part in these measurements.

Two families of experiment run over this setup. The first times store operations one at a time. The store is pre-populated with 256 entries carrying 1-kilobyte payloads, and each operation is measured at tree capacities from 2048 to 32768 blocks, meaning 12 to 16 levels, the last being the depth the deployed store uses. A separate kernel-level sweep drives single accesses directly, at capacities from 32 blocks up to the same 32768, and yields the per-access latency curve. A `find` and a name lookup are timed over 200 repetitions after 20 warm-up calls, an insert over 60 after 6, and an erase over 30 after 3, fewer because a single oblivious erase already runs over two thousand accesses. Alongside the wall-clock, the store's own audit hooks record the oblivious accesses each operation issues and the bytes it moves, which is exactly what the host would observe.

The second family is a cache workload. A catalog of 300 *Data* packets is built at start-up, and consumer demand draws names from a Zipf distribution over it, the standard model for content popularity in named-data networks [37, 38]. Under Zipf, the i-th most popular name is requested with probability proportional to $1/i^{\alpha}$, so the exponent α sets how skewed demand is: at $\alpha = 0.8$ requests spread fairly evenly across the catalog, while at $\alpha = 1.2$ a handful of popular names draws most of the traffic. The runs use $\alpha \in \{0.8, 1.0, 1.2\}$ and sweep the cache limit from 5% of the catalog to 100%. Each configuration runs 150 warm-up requests and then 150 measured ones, both stores seeing the identical stream from the same seed, and the baseline run is repeated seven times with the median kept, since a plain run finishes in microseconds. A request issues an exact-match *Interest* against the store, and on a miss the harness inserts the matching packet, standing in for the producer fetch. That packet already sits in memory, so the fetch is free. The simplification cuts in the oblivious store's favor, since a deployed forwarder would charge every miss a round trip, and that shared cost would narrow the distance between the two stores.

In sum, the first family yields the latency of a single oblivious access as the tree grows, the latency, access count, and bytes moved of every operation against the baseline, and the

resident memory of the store against a plain dictionary at three entry sizes. The second yields the throughput and the cache hit ratio of both stores on the identical stream, together with the occupancy the in-enclave stash reaches as the tree grows.

7.2. Results

This section reports the measurements in three parts, the cost of an oblivious access against the size of the tree, the per-operation comparison with the baseline, and the memory footprint.

7.2.1 Access cost and tree size

Figure 7.1 plots the mean latency of a single oblivious access as the tree capacity grows from 32 blocks to the deployed 32768, 6 to 16 levels. It rises from about 0.22 milliseconds to about 1.18, and the rise tracks the level count rather than the block count. Capacities sharing a level cost the same, and doubling the capacity adds one level and a near-constant slice of latency, which is the $O(\log N)$ behavior Path ORAM predicts for an access that reads and rewrites one root-to-leaf path. These are standalone accesses, each paying its per-call setup in full. Inside an operation that fixed work is instead spread across the run of consecutive accesses, and an access averages about 0.7 milliseconds at the deployed depth, the 16-millisecond `find` of Table 7.2 over its 22 accesses; that average figure is the one the rest of the chapter uses. Getting to that depth takes a sizeable store: the deployed tree is dimensioned for 29824 blocks across the six domains, held in 65535 buckets of four 8800-byte slots each, which comes to about 2.3 gigabytes of bucket ciphertext on the host. Section 6.2 explains how the 29824 blocks are reached. With one leaf per block the tree needs 2^{15} leaves, and so 16 levels.

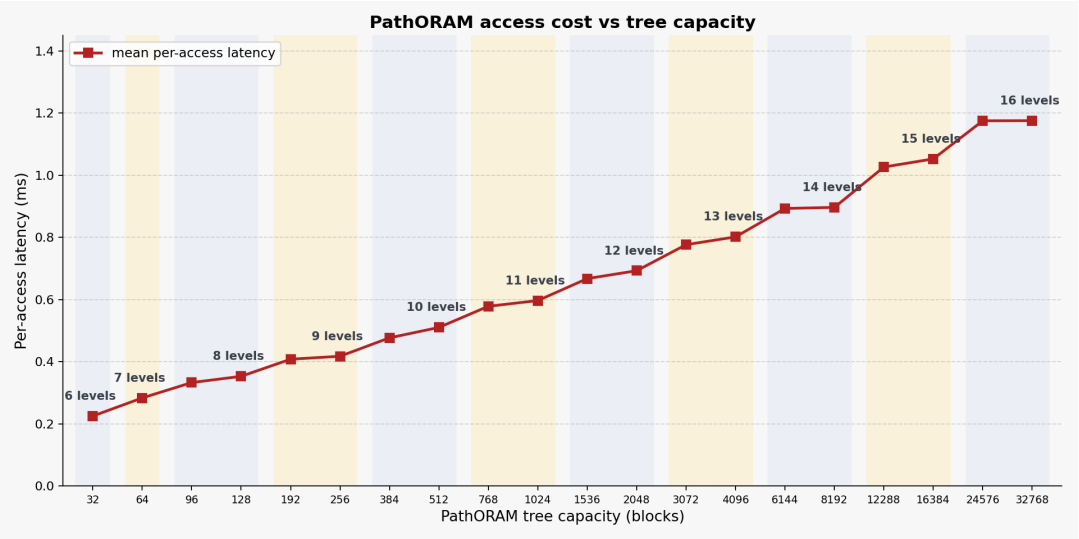


Figure 7.1: Latency of a single oblivious access against tree capacity.

7.2.2 Comparison with the baseline

A forwarder operation is not one access but a fixed run of them. A `find` spends a budget of 22 oblivious accesses, a name lookup 20, an insert 337, and an erase 2229, all measured at the 16-level tree. These budgets are built from the height bound of the name index. An index lookup always reads $h = 20$ nodes, the padded AVL height for the index sizes of Section 6.2, and an index insert and erase are padded to $6h + 48 = 168$ and $52h + 64 = 1104$ accesses, upper bounds that cover the worst-case rebalancing. A `find` is one index lookup followed by one read of the data block and one write that updates its recency, or a dummy read and write on a miss, so $20 + 2 = 22$. A name lookup is the index walk alone, 20. An insert enters the new entry in both of the store's indexes, by full name and by plain name, and writes the data block, so $2 \times 168 + 1 = 337$. An erase removes the entry from both indexes, with one lookup first to check that the plain-name index still points at this block, and then clears the block, so $1104 + 20 + 1104 + 1 = 2229$. None of these counts depends on the key or on whether the entry is present, so the cost of a request says nothing about its content. Table 7.2 sets those operations against the baseline, in latency and in bytes moved. Bytes moved counts the bucket ciphertext an operation carries across the enclave boundary: each access reads one full path of buckets, decrypts it inside the enclave, and writes the same path back re-encrypted, and both directions count. The baseline answers each in well under a microsecond and moves about one entry, roughly a kilobyte. The oblivious store answers a `find` in about 16 milliseconds and moves near 25 megabytes, and an erase costs close to a full second and moves two and a half gigabytes, because it runs the whole conditional-erase budget every time so that whether a key was there stays hidden.

Operation	Baseline	Oblivious	Accesses	Bytes moved
find (hit)	0.07 µs	16 ms	22	24.8 MB
find (miss)	0.06 µs	14 ms	22	24.8 MB
name lookup	0.03 µs	11 ms	20	22.6 MB
insert	0.14 µs	160 ms	337	380 MB
erase	0.10 µs	1.0 s	2229	2.5 GB

Table 7.2: Per-operation cost of the oblivious store against a non-oblivious baseline, at the deployed 16-level tree. The baseline issues one access and moves about one entry.

Figure 7.2 shows the same latencies side by side, and its right-hand panel condenses them into a single summary, the overhead factor, oblivious latency over baseline latency for each operation. In the left panel the baseline bars sit on the floor, tens of nanoseconds, while the oblivious bars stand orders of magnitude above them, and the distance widens from `find` through insert to erase as the fixed budget grows. The overhead panel reads that distance directly: about five orders of magnitude for a `find`, six for an insert, closer to seven for an erase. Read as an absolute the oblivious `find` is still only tens of milliseconds, which the working-cache figures below put in context.

The size of that distance asks to be set against other ORAM systems, and the comparison has to be made at the right level. At the level of a single access, published SGX-based ORAMs report costs in the range ours lands in. These figures are the ones the authors report, measured on their own hardware and workloads. We did not rerun those systems, so the comparison places our numbers within the published range rather than benchmarking the systems against each other. ZeroTrace serves one access to a 10-gigabyte store in 1.2 to 3.4 milliseconds depending on the block size [15], Oblix answers a complete oblivious index search over 2^{24} key-value pairs in about 5 milliseconds and an insert in about 6 [16], and our access at 16 levels costs about 0.7 milliseconds on average within an operation, or about 1.2 measured standalone. That per-access comparison does not carry over to whole operations, though. Those systems price a single access or a single padded descent, while a forwarder operation here is a fixed run of them, 22 accesses for a `find` and 2229 for an erase, paid in full every time so that the outcome stays hidden. The honest check is therefore to multiply: 22 accesses at ZeroTrace's 1.2 to 3.4 milliseconds each would cost 26 to 75 milliseconds, and our 16-millisecond `find` sits below that band. The per-operation totals are heavy because the budgets are, not because the primitive underneath is slow. Two local choices still push the absolute numbers up. The block is

8800 bytes, sized to the largest *Data* packet, so one access moves about 1.1 megabytes of path where a system built on small blocks moves kilobytes, and the baseline is an in-memory hash map answering in tens of nanoseconds, a harsher reference point than the encrypted or remote stores the literature compares against. So the prototype is not an outlier in what an oblivious access costs. It is expensive in how many accesses it buys per operation and in how large each one is, the first being the price of hiding outcomes, the second a recoverable cost that Chapter 8 returns to.

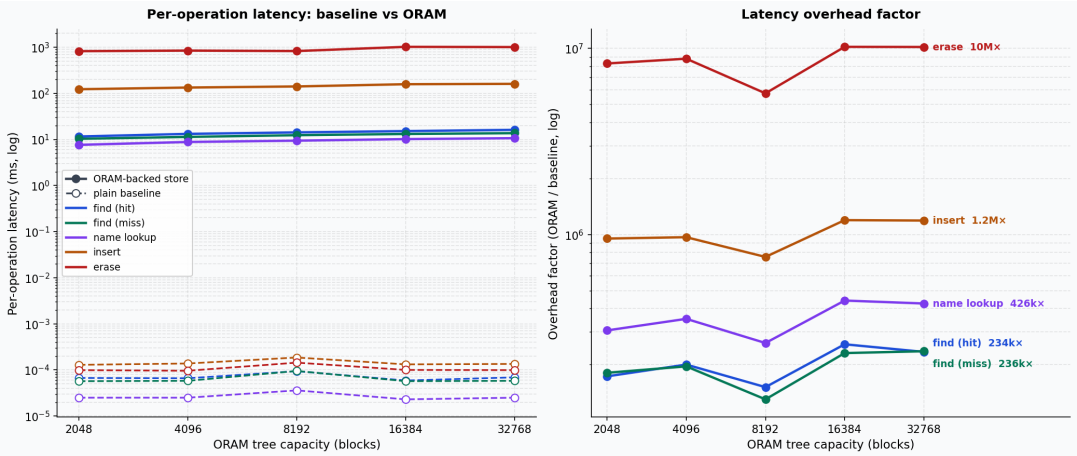


Figure 7.2: Per-operation latency of the oblivious store against the baseline (left) and the resulting overhead factor (right).

The bandwidth in that table is the dominant cost and the reason for the latency. A baseline operation touches one hash bucket and copies one entry. Every oblivious access instead reads and rewrites a full path of AES-GCM buckets, about 1.1 megabytes at 16 levels, so even a `find` moves tens of megabytes across the enclave boundary and an erase moves gigabytes. This is the canonical Path ORAM cost, and unlike the wall-clock it does not vary with the platform. A hit and a miss move the identical 22 accesses and 24.8 megabytes, a point taken up below.

Put back into a working cache, these per-operation costs set the throughput. Figure 7.3 plots operations per second against cache size, for both stores, under the three Zipf exponents. The baseline sustains several million operations a second, the oblivious store single digits to low tens. The gap is easy to state: a plain cache answers a hit in effectively no time, so the more hits a workload produces the faster it runs, while the oblivious store charges a hit and a miss the same worst-case price and can never cash that advantage in. Its curve does still rise with cache size and skew, since a fuller cache turns expensive evicting inserts into cheaper `find`s, but the six orders of magnitude between the stores

stand. Throughput is also the figure a hardware deployment would most change, and Section 7.4 explains in which direction.

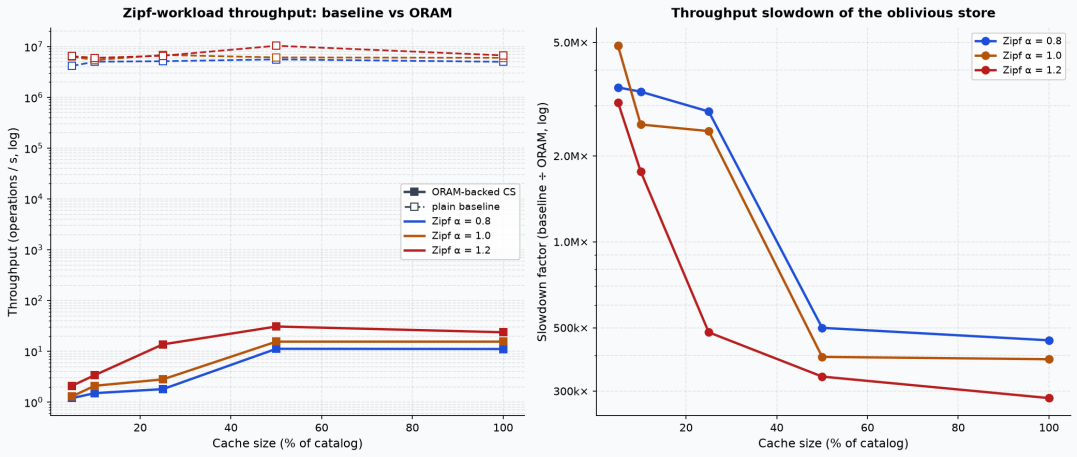


Figure 7.3: Throughput against cache size, oblivious store against the baseline.

Table 7.2 showed a hit and a miss costing the same, and Figure 7.4 widens that check across cache sizes. The outcome of a request is the sharpest thing a plain cache betrays, through the difference between a fast answer and a slow one. Here, hit or miss, a `find` issues the same 22 accesses and moves the same 24.8 megabytes, and the two lines lie one on top of the other across the whole sweep, which confirms the implementation did not break the symmetry the construction provides. One residual difference does survive in raw wall-clock. A hit runs a little longer than a miss, by about two milliseconds on a find of some sixteen, because it still has to copy out the entry it found, work that sits outside the oblivious store. Closing that last timing gap with a constant-time return path is left as a refinement. The insert line in the same figure sits far higher, which is the price of an oblivious write rather than a leak, peaking when the cache is small and nearly every insert must evict.

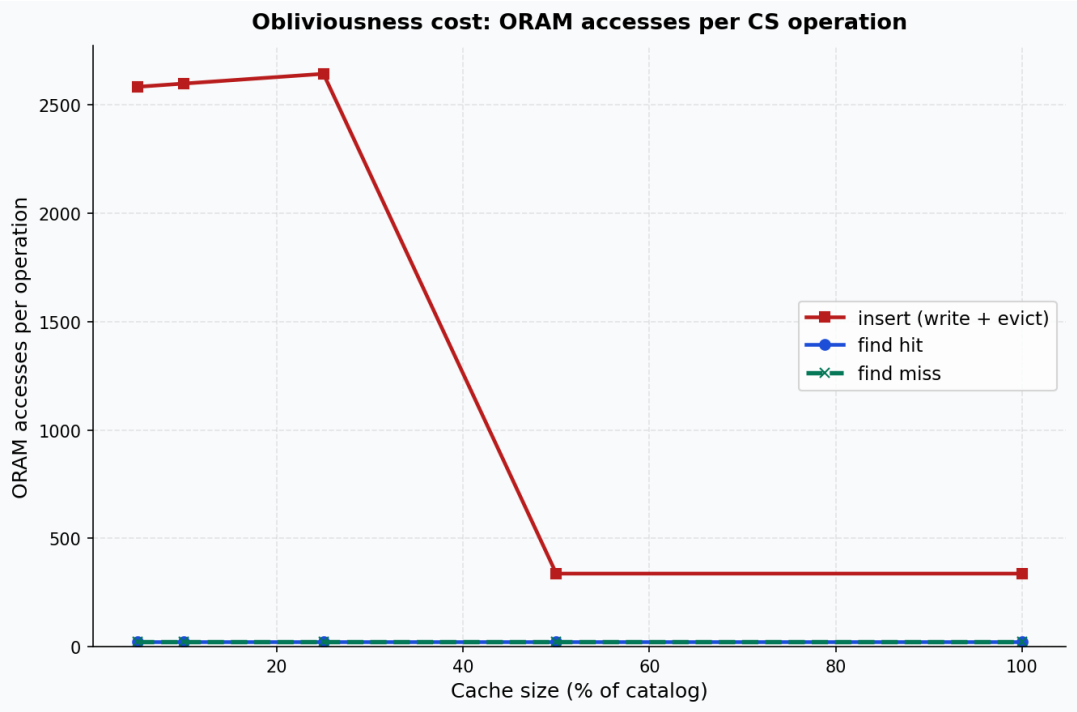


Figure 7.4: Oblivious accesses per CS operation, with the hit and miss lines coinciding.

The last comparison is the cache behavior itself. On the shared request stream the two stores recorded the same hits and the same misses at every cache size and every exponent, with the hit ratio following the concave rise the caching literature reports [37, 38] and reaching about 0.88 at the most skewed workload with a cache a quarter the size of the catalog. The store makes the same caching decisions a plain one would, which is what keeps the comparison fair, and the obliviousness hides the access pattern without distorting the behavior above it.

7.2.3 Memory footprint

Memory is the other standing cost. Four blocks to a bucket, and each block fixed at 8800 bytes, the largest *Data* packet the NDN protocol admits, so no packet can outgrow its block, and the tree reserves far more space than the data it stores. Figure 7.5 sets the measured footprint of the oblivious store against a plain dictionary holding the same entries, for three entry sizes. At 8192 entries the store uses about 582 megabytes. Against small 256-byte entries that is a 200-fold blow-up, against 1-kilobyte entries 63-fold, against 4-kilobyte entries 17-fold. The blow-up shrinks as the stored entries approach the block size, which is why sizing the block to the median packet rather than the largest, a limitation Chapter 6 records, would win back much of this memory. That smaller block would no longer fit the

rare oversized packet in one piece, so it would have to be split across blocks or set aside, which is the complication the fixed maximum-size block buys its way out of.

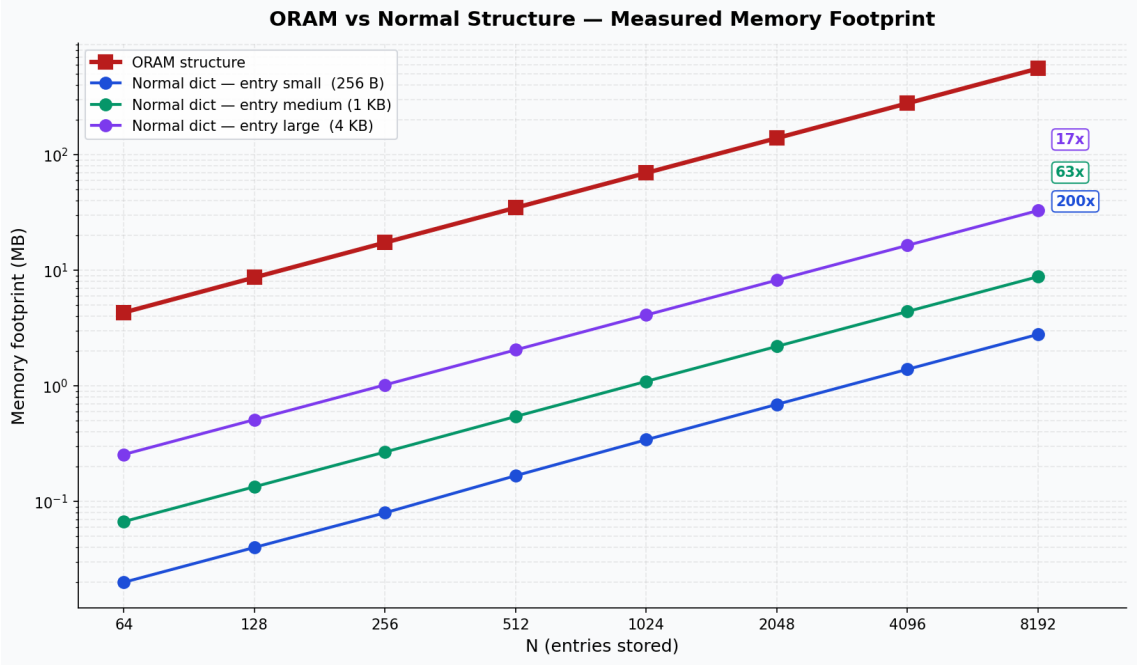


Figure 7.5: Memory footprint of the oblivious store against a plain dictionary.

The cost, then, is heavy but bounded, and it is driven by the size of the tree rather than by the load. An operation pays the same fixed budget whether the store holds ten entries or ten thousand near its capacity, which is the very thing that keeps the cost from leaking what the store contains. That is the trade the design makes, protection bought at a fixed, worst-case price.

7.3. Stash analysis

The stash is a point any discussion of ORAM has to treat with particular care, and this section evaluates how it grows on the running kernel. Path ORAM keeps in enclave memory a small buffer of blocks that could not be written back to the tree on the last access, and the construction leans on that buffer staying small. Were the stash to grow with the tree, the client would eventually lack protected memory to hold it, and blocks would have nowhere safe to wait. The theory says this does not happen, the stash stays below a small constant except with negligible probability [10], but that is an asymptotic result over an idealized model, which makes watching it in operation worthwhile.

Figure 7.6 plots the maximum stash occupancy observed as the tree grows from 64 blocks to the deployed 32768, a factor of 512. The maximum stays in the single digits, peaking at

ten blocks with no upward trend across the whole sweep, and at the deployed 16-level size it was six. The mean occupancy sits below 0.04 blocks everywhere. The running kernel behaves as the theory says it should. In bytes the margin is wide. Ten 8800-byte blocks is under 90 kilobytes, three orders of magnitude below even the roughly 93 usable megabytes of the smallest SGX enclave page cache. The stash is not a growing cost, and enclave memory holds it comfortably at any tree size the prototype uses.

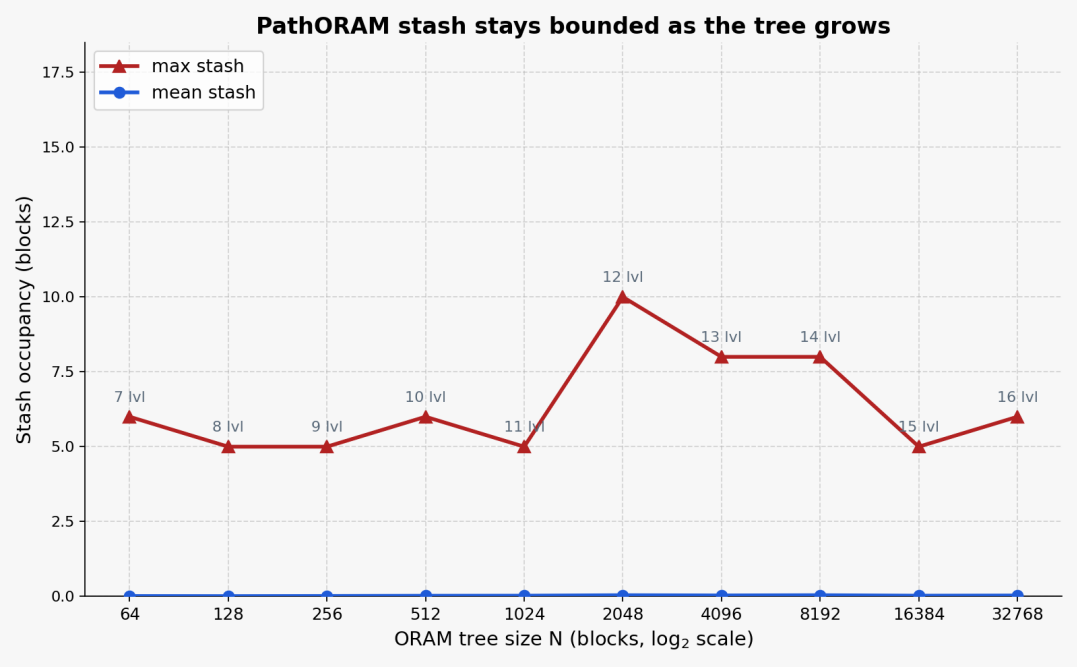


Figure 7.6: Stash occupancy against tree size.

7.4. Discussion

The cost figures are firm in their shape but bounded in what their absolute values carry, and the platform is the clearest reason. The measurements run natively rather than inside a real enclave. The counts do not depend on that, so the oblivious accesses per operation, the bytes moved, and the memory blow-up would read the same on hardware. The wall-clock would not. On real SGX the processor encrypts and checks the memory the enclave uses, and the oblivious store moves a great deal of it, about 1.1 megabytes per access. Every system call and every exchange with the host also leaves the enclave and comes back, a transition native runs never pay. Both effects can only add time, so a hardware run would push the oblivious latencies up and the throughput down. The figures here are therefore a lower bound on the latency of the oblivious store, which is the sense in which they are optimistic.

The scope of the harness bounds them further. Driving the store directly, as Section 7.1 described, is what lets an operation's cost be attributed exactly without charging it for the rest of the NFD pipeline, and the price is that these are the costs of the store alone. An end-to-end forwarding path would add face handling, packet parsing, and the PIT and FIB work a miss triggers, and it would charge every miss a real fetch from a producer, which the harness gives away free. That last one matters most for the throughput comparison, since a shared per-miss cost of milliseconds would compress the distance between the two stores considerably. The cache experiments lean on a small catalog too, 300 objects, enough to bring the curves out clearly but far short of a deployed forwarder. A full end-to-end evaluation inside NFD, with a Zipf workload driven over real faces, is left for future work, as is a run on SGX hardware once it can be had.

The baseline itself carries one caveat worth stating. The stock NFD `Cs` cannot be built alongside the oblivious one, since the forwarder was made oblivious in place, so the baseline reproduces its behavior with a plain map and an $O(1)$ eviction queue behind the same interface. The identical hit and miss counts on the shared workload are what make the comparison fair, since they show the two stores made the same decisions and only the storage differs.

The evaluation covers, finally, the guarantees the prototype implements, not the ones left open in Chapter 6. The exact-match and freshness paths are measured. The unbounded longest-prefix match over names is not made oblivious, and so is not evaluated here. Within that scope the picture holds together. The design delivers the obliviousness the requirements set out, and it does so at a bandwidth and memory cost that is high but bounded, which is the trade the rest of the thesis argued for.

Chapter 8

Conclusion and Future Work

What this thesis set out to do was hide what a named-data forwarder does with the requests it handles. A CCN/NDN forwarder keeps its state in three tables, and two of them leak consumer behavior to anyone who can watch the host. The CS betrays what has been requested through the difference between a hit and a miss, and the PIT betrays which names are in flight through its occupancy. Chapter 3 and Chapter 4 turned that observation into a threat mapping, and then into six requirements any countermeasure would have to meet. The rest of the thesis met them with a design, and measured what that cost.

8.1. Summary

The design places the whole forwarder in one SGX enclave and backs its CS, PIT, and name index with a single Path ORAM tree. Every table access moves a full root-to-leaf path rather than one entry, and the name index walks a padded height, so neither the entry touched nor the outcome of a request shows in the access pattern or the cost. Whatever state has to live in host memory is encrypted, authenticated, and versioned against rollback. And a client checks that it is talking to the genuine forwarder through remote attestation bound into the transport channel.

Read against the requirements of Chapter 4, the outcome is as follows. R1 and R2 are met by construction, every table access and every index walk being a padded run of Path ORAM accesses. R3 is met by the single tree and the single free-list, with the domain tags hidden inside the ciphertext. R4 is met in accesses and bytes, a hit and a miss both costing 22 accesses and 24.8 megabytes, with the copy-out of about two milliseconds on a hit as the one residual in time. R5 is met for the confidentiality, integrity, and freshness of the bucket tree, and the attested channel is implemented, though only its encryption half runs in simulation mode. R6 is met, the oblivious store and a plain one making the same caching decisions throughout. Two parts of the forwarding state stay outside what was verified, prefix matching and the FIB, whose records remain outside the oblivious store with only their name resolution made oblivious.

All this is built on NFD and the ndn-cxx library. The prototype forks the forwarder and changes it in place, and it adapts an open-source Path ORAM kernel rather than writing one from scratch. What the work adds on top of that kernel is the shared store that tags blocks by domain and draws them from one free-list, the oblivious name index over the same tree, the authenticated anti-rollback externalization of the bucket tree, the constant-time hardening of the client walks, and the integration of all of it into NFD's CS and PIT behind an attested gateway. For convenience of implementation the prototype runs SGX in simulation mode through `gramine-direct`, which loads the same binary and manifest as real SGX would.

Chapter 7 put a price on that protection, measured against a non-oblivious store driven by the same workload behind the same interface. The gap is wide. A plain `find` answers in tens of nanoseconds by touching one hash bucket, while the oblivious one takes about 16 milliseconds and moves near 25 megabytes, and an erase runs close to a second and moves two and a half gigabytes because it pays its full budget every time. Under a Zipf request stream the baseline sustains millions of operations a second where the oblivious store sustains single digits to low tens. In memory the store costs 17 to 200 times a plain dictionary, depending on how large the stored entries are against the fixed block. Set against published SGX ORAM systems the per-access cost is in line, with ZeroTrace serving an access to a 10-gigabyte store in 1.2 to 3.4 milliseconds and Oblix answering an index search over 2^{24} pairs in about 5 milliseconds [15, 16], and the wider per-operation gap comes from the large fixed block and the padded budgets that hide each operation's outcome. The cost is driven by the size of the tree rather than by the load, which is exactly what keeps it from leaking what the store holds.

The same chapter confirms that the running implementation still exhibits the property the construction guarantees. A CS `find` issues the same 22 oblivious accesses and moves the same bytes whether it hits or misses, and the in-enclave stash stays bounded as the tree grows by a factor of 512. Driven on one request stream the oblivious store and the plain one recorded the same hits and misses throughout, so the hiding costs nothing in caching behavior. The protection is real, and its cost is high but bounded, which is the trade the thesis argued for.

8.2. Future work

Several directions are left open by the prototype, some needed before a production build and some worth measuring in their own right.

The most immediate limitation is prefix matching. Exact-match and freshness queries the name index answers obliviously, but NDN's unbounded longest-prefix match it does not. An *Interest* carrying `CanBePrefix` makes the index visit every stored name under the queried prefix, and while each visit is oblivious, the count of visits tracks how many names share that prefix. A forwarder that has to route arbitrary prefixes obliviously would need an index covering prefix search without letting the visit count depend on the prefix population. Here is the main piece of functionality standing between the prototype and a general deployment.

A narrower version of the same limitation sits between a PIT entry and its next hop. When a name runs deeper than the node the index returned, the forwarder walks the remaining components one at a time and stops at the first that is missing, so the probe count follows how deep the name went. Extending to that caller the fixed-probe walk the rest of the index already performs would close it, and that is a much smaller change than oblivious prefix search.

A second direction is the platform, expanding the prototype from SGX in simulation mode to a non-simulated deployment on real hardware. The results that rest on the construction, the equal access cost of a hit and a miss, the bounded stash, the scaling curves, hold as measured and would read the same on hardware. The absolute latency and throughput would not, since real SGX pays for memory encryption and for entering and leaving the enclave, and in simulation mode only the encryption half of the attestation gateway runs at all. Run the prototype on SGX hardware and it would yield the true latency and throughput, and would exercise the attestation path end to end.

The evaluation itself can be widened. Its measurements drive the CS class and the Path ORAM kernel directly, in one process and without the network, which is what lets a hit or a miss be attributed exactly but leaves out the rest of the forwarding pipeline. A full evaluation inside NFD, with a Zipf workload driven over real faces and a catalog nearer the scale of a deployed forwarder, would show the cost of an end-to-end oblivious forwarding path rather than of the store alone.

Two smaller changes would recover cost without weakening the guarantee. The fixed block is sized to the largest NDN *Data* packet, so every slot reserves a full block whatever it holds. Sizing the block to the median packet instead would cut both latency and memory, and Chapter 7 shows how much of the footprint that fixed size accounts for. The enclave, separately, uses a fresh ephemeral key each run, which suits a volatile cache and gives

cross-restart freshness for free. A forwarder wanting to keep its cache across restarts would need a sealed, hardware-derived key together with sealed version counters, so that the anti-rollback guarantee survives a restart under a stable key.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, “Networking named content,” in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2009, pp. 1–12. [Cited on pages 1 and 6.]

[2] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, “Named data networking,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014. [Cited on pages 1, 6, and 17.]

[3] A. Afanasyev, J. Shi, B. Zhang, L. Zhang *et al.*, “NFD developer’s guide,” NDN Project, Tech. Rep. NDN-0021, Revision 10, 2018. [Cited on pages 1 and 7.]

[4] G. Acs, M. Conti, P. Gasti, C. Ghali, and G. Tsudik, “Cache privacy in named-data networking,” in *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013, pp. 41–51. [Cited on pages 2, 17, 18, and 26.]

[5] A. Mohaisen, X. Zhang, M. Schuchard, H. Xie, and Y. Kim, “Protecting access privacy of cached contents in information centric networks,” in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013, pp. 173–178. [Cited on pages 2, 17, 18, and 26.]

[6] V. Costan and S. Devadas, “Intel SGX explained,” 2016, cryptology ePrint Archive, Report 2016/086. [Online]. Available: https://eprint.iacr.org/2016/086 [Cited on pages 2 and 13.]

[7] S. Pinto and N. Santos, “Demystifying Arm TrustZone: A comprehensive survey,” *ACM Computing Surveys*, vol. 51, no. 6, pp. 130:1–130:36, 2019. [Cited on pages 2 and 13.]

[8] Intel Corporation, “Intel trust domain extensions (Intel TDX) whitepaper,” https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023. [Cited on pages 2 and 14.]

[9] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious RAMs,” *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996. [Cited on pages 2 and 10.]

[10] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: An extremely simple oblivious RAM protocol,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013, pp. 299–310. [Cited on pages 2, 10, 11, 43, and 66.]

[11] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang, “A case for stateful forwarding plane,” *Computer Communications*, vol. 36, no. 7, pp. 779–791, 2013. [Cited on page 7.]

[12] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, “PHANTOM: Practical oblivious computation in a secure processor,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013, pp. 311–324. [Cited on pages 12 and 18.]

[13] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, “Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 103–116. [Cited on page 12.]

[14] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, “Oblivious data structures,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 215–226. [Cited on pages 12, 35, 44, and 49.]

[15] S. Sasy, S. Gorbunov, and C. W. Fletcher, “ZeroTrace: Oblivious memory primitives from intel SGX,” in *Network and Distributed System Security Symposium (NDSS)*, 2018. [Cited on pages 19, 62, and 70.]

[16] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, “Oblix: An efficient oblivious search index,” in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 279–296. [Cited on pages 12, 19, 62, and 70.]

[17] D. Kaplan, J. Powell, and T. Woller, “AMD memory encryption,” Advanced Micro Devices, White Paper, 2016. [Cited on page 13.]

[18] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution,” in *27th USENIX Security Symposium*, 2018, pp. 991–1008. [Cited on pages 13, 22, and 28.]

[19] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, “LVI: Hijacking transient execution through microarchitectural load value injection,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 54–72. [Cited on pages 13, 22, and 28.]

[20] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A practical library OS for unmodified applications on SGX,” in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 645–658. [Cited on page 14.]

[21] S. Kent and K. Seo, “Security architecture for the internet protocol,” Internet Engineering Task Force, RFC 4301, 2005. [Cited on page 16.]

[22] Y. Yu, A. Afanasyev, D. Clark, K. Claffy, V. Jacobson, and L. Zhang, “Schematizing trust in named data networking,” in *Proceedings of the 2nd ACM Conference on Information-Centric Networking (ICN)*, 2015, pp. 177–186. [Cited on page 17.]

[23] Z. Zhang, Y. Yu, S. K. Ramani, A. Afanasyev, and L. Zhang, “NAC: Name-based access control in named data networking,” in *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN)*, 2017, pp. 186–187. [Cited on page 17.]

[24] P. Gasti, G. Tsudik, E. Uzun, and L. Zhang, “DoS and DDoS in named data networking,” in *22nd International Conference on Computer Communication and Networks (ICCCN)*, 2013, pp. 1–7. [Cited on page 17.]

[25] A. Compagno, M. Conti, P. Gasti, and G. Tsudik, “Poseidon: Mitigating interest flooding ddos attacks in named data networking,” in *38th Annual IEEE Conference on Local Computer Networks (LCN)*, 2013, pp. 630–638. [Cited on pages 17, 18, and 22.]

[26] E. Stefanov and E. Shi, “ObliviStore: High performance oblivious cloud storage,” in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 253–267. [Cited on page 18.]

[27] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, “Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 837–849. [Cited on page 18.]

[28] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, “ShieldBox: Secure middleboxes using shielded execution,” in *Proceedings of the Symposium on SDN Research (SOSR)*, 2018, pp. 1–14. [Cited on page 19.]

[29] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, “SafeBricks: Shielding network functions in the cloud,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 201–216. [Cited on page 19.]

[30] B. Trach, R. Faqeh, O. Oleksenko, W. Ozga, P. Bhatotia, and C. Fetzer, “Slick: Secure middleboxes using shielded execution,” arXiv:1709.04226, 2017. [Cited on page 19.]

[31] Z. Yang, X. Li, L. Wei, C. Zhang, and C. Gu, “SGX-ICN: A secure and privacy-preserving information-centric networking with SGX enclaves,” in *2020 3rd International Conference on Hot Information-Centric Networking (HotICN)*. IEEE, 2020, pp. 142–147. [Cited on page 19.]

[32] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 640–656. [Cited on page 22.]

[33] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: SGX cache attacks are practical,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017. [Cited on page 22.]

[34] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70. [Cited on page 22.]

[35] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998. [Cited on page 35.]

[36] ObliviousRAM Project, “PathORAM: A C++ implementation of the path ORAM protocol,” https://github.com/obliviousram/PathORAM, accessed 2026. [Cited on pages 43 and 58.]

[37] E. T. d. Silva, J. M. M. Macedo *et al.*, “NDN content store and caching policies: Performance evaluation,” *Computers*, vol. 11, no. 3, p. 37, 2022. [Cited on pages 58, 59, and 65.]

[38] S. Shailendra, S. Sengottuvelan, H. K. Rath, B. Panigrahi, and A. Simha, “Performance evaluation of caching policies in NDN – an ICN architecture,” arXiv:1612.00352, 2016. [Cited on pages 58, 59, and 65.]