Aalto University
School of Science
Master's Programme in Security & Cloud Computing

Ayoub Chouak

Trustworthy Data Provenance for Enclaves in Heterogeneous Distributed Systems

Master's Thesis
Espoo, July 3, 2022

Supervisors:	Lachlan J. Gunn (University Lecturer), Aalto University
	Prof. Nicola Dragoni, Technical University of Denmark
Advisor:	Edlira Dushku (Postdoctoral Researcher)

Aalto University
School of Science
Master's Programme in Security & Cloud Computing

ABSTRACT OF MASTER'S THESIS

Author:	Ayoub Chouak		
Title: Trustworthy Data Provenance for Enclaves in Heterogeneous Distributed Systems			
Date:	July 3, 2022	**Pages:**	85
Major:	Computer Science	**Code:**	SCI3113
Supervisors:	Lachlan J. Gunn (University Lecturer) Prof. Nicola Dragoni		
Advisor:	Edlira Dushku (Postdoctoral Researcher)		

Trusted execution environments (TEEs) have gained significant traction over the last few years. They allow mutually distrusting systems to entrust each other with data and computation by running applications in strongly isolated containers called enclaves. Different TEEs can run different versions of an enclave platform and their realization depends on the underlying hardware. As enclaves migrate across many different TEEs, their integrity can be compromised. By tracking the provenance of enclaves, TEEs can assess their trustworthiness based on their migration history. However, this requires that the provenance data itself also be trustworthy.

In this work, we leverage the strong isolation guarantees and attestation capability of TEEs to build QUICKPROV, a framework for fast, trustworthy data provenance for enclaves in heterogeneous distributed systems. We first show how we achieve trustworthy data provenance without using blockchains and consensus algorithms, and by using TEE capabilities. We then build a TrustZone-assisted enclave platform to support our provenance framework. Finally, we develop a proof-of-concept (PoC) implementation for QUICKPROV that is minimally intrusive and is tamper-resistant even in the presence of some compromised TEEs.

Keywords: remote attestation, provenance, trusted execution environment, migration, enclave, webassembly

Language: English

Acknowledgements

I would like to thank everyone who has believed in me and helped me maintain the mental strength and perseverance that is required to reach the end of my studies, of which this thesis is the most crowning achievement. I would also like to extend my thanks to my supervisor, Lachlan J. Gunn, for the immense support he has given me over the past few months. My special thanks also go to Nicola Dragoni and Edlira Dushku for supervising my thesis from DTU, and to Wojciech Geisler for being an all-round, great colleague and for writing a part of the code that was instrumental for the evaluation of my work. Finally, I would like to dedicate this work to my parents, who I cannot ever thank enough, and without the support of whom this work would probably not exist.

Espoo, July 3, 2022

Ayoub Chouak

With the support of the
Erasmus+ Programme
of the European Union

Abbreviations and Acronyms

ACL	access control list
API	application programming interface
CSP	cloud service provider
CA	certificate authority
OEM	original equipment manufacturer
OS	operating system
WASM	WebAssembly
WASI	WebAssembly System Interface
WASMI	WebAssembly Interpreter
TCB	trusted computing base
HUK	hardware unique key
SGX	Secure Guard Extensions
GP	Global Platform
REE	rich execution environment
TEE	trusted execution environment
TA	trusted application
WTA	WebAssembly trusted application
EA	Enhanced Authorization
RoT	root of trust
RPMB	replay-protected memory block
BoF	beginning of feed
EoF	end of feed

Contents

Chapter 1

Introduction

The widespread availability of cloud and mobile edge computing (MEC) has drastically changed the way we think of computation by progressively moving data ownership and processing away from our personal devices. This has given rise to a whole new array of security issues that adversaries constantly try to exploit to break the confidentiality, integrity and availability (CIA) of private data and computation. Therefore, there needs to be a means for establishing trust, such that clients and service providers can entrust each other with their data and be assured that co-located malicious applications or a compromised operating system (OS) cannot hijack it. Trusted execution environments (TEEs) enable this by providing an isolated execution environment for confidential computation and data storage.

TEEs have gained significant traction over the last few years and a multitude of applications has been developed on top of them. One of these is secure computation offloading, whereby a mobile device outsources its workload to a TEE in a cloud or MEC provider, usually for performance reasons. These workloads are executed in secure *enclaves* within TEEs and can be migrated multiple times as a result of load balancing or scheduled maintenance [25]. Although TEEs usually adopt a general protection profile [21], distributed systems are heterogeneous by nature, meaning that TEEs can exist in different realizations, depending on the underlying platform hardware. Even the same TEE technology (e.g., TrustZone) can have different implementations, depending on the manufacturer, and run different OSs.

As enclaves migrate across many different—and, possibly, less secure—TEEs, their integrity will depend on their entire migration history. This sequence of enclave migrations, called a *chain of custody* [67], along with all the transformations on its data, constitutes its *provenance*. The provenance of data provides crucial information for evaluating the quality of data and ascertaining its reliability [31]. It also enables TEEs to assess the trustworthiness of enclaves and their data based on their migration history. However, this requires that the provenance data itself also be trustworthy, as insecure provenance can often be more harmful than having no provenance at all [67].

For data provenance to be trustworthy, it needs to be collected and stored securely, and there needs to be a means to convince interested parties of its reliability. We can leverage the strong isolation guarantees and trusted storage capability of TEEs to collect and store provenance data, and use their remote attestation capability to certify its trustworthiness. Attestation allows for establishing trust in the provenance data without relying on slow, expensive consensus algorithms. In this work, we introduce QUICKPROV, a framework for fast, trustworthy data provenance for enclaves in heterogeneous distributed systems.

1.1 Motivation

State-of-the-art solutions for trustworthy data provenance, such as [28, 33], use TEEs for secure provenance collection and blockchains to keep a highly-available, tamper-evident log of the provenance metadata. This log is sometimes backed by an off-chain storage medium as in [64]. Blockchains typically rely on expensive consensus algorithms to attain unanimous agreement on the state of the ledger. To achieve consensus, permissioned blockchains like Hyperledger Fabric [5] rely on Byzantine fault tolerance (BFT) algorithms such as PBFT [12], Tendermint [9] and Streamlet [14]. These algorithms guarantee low-latency, energy-efficient, deterministic finality of transactions, but suffer from scalability issues [4]. In public blockchains, Nakamoto consensus [47] is typically employed instead and coupled with proof of work (PoW), which mitigates Sybil [20] attacks. The probabilistic nature of PoW, however, entails a high confirmation latency, which is impractical for real-time applications.

None of the previous work has tackled the challenge of provenance data for real-time applications like secure computation offloading, where migration time can be critical depending on the application. Consider an enclave that is migrated from a mobile device's TEE to a remote TEE because of battery or CPU limitations [42]. The two TEEs are likely to rely on different platform hardware and, possibly, different versions of the same enclave platform. In this situation, the destination TEE may need to verify the enclave's provenance to assess its integrity before resuming execution.

Conversely, the TEE on the mobile device will want to verify the provenance of its enclave and its data before trusting it again. Depending on the consensus mechanism, TEEs may have to wait seconds—if not minutes—for all provenance data to be confirmed.

Another interesting application of data provenance for enclaves is in Function-as-a-Service (FaaS), where a *function provider* publishes a function on a cloud service provider (CSP) that can be remotely invoked by clients. These functions may operate on confidential data that the client supplies on invocation and that they do not want divulged. Data provenance would enable FaaS clients to audit the IO operations that the invoked function has performed and verify that it has only written data to private objects that can only be accessed by the function itself. In essence, provenance would enable clients to verify the security of TEE-assisted privacy preserving computation, an example of which is malware detection services, where snapshots of the applications installed in clients' devices could be used by malicious parties for profiling [60].

1.2 Contribution

In this work, we leverage the strong security and isolation guarantees of TEEs to build a framework for fast and trustworthy data provenance for secure enclaves. We summarize our contributions as follows:

- We built a TrustZone-assisted enclave platform for portable WebAssembly applications
- We developed a proof-of-concept (PoC) for QUICKPROV, a TEE-assisted framework for trustworthy data provenance that is tamper-resistant to compromised TEEs
- We showed how we achieved trustworthy data provenance without using blockchains and consensus algorithms and by simply using TEE guarantees of strong isolation, trusted storage and attestability

Chapter 2

Background

In this chapter, we present essential preliminary knowledge that will help the reader get a better grasp of the concepts described in the next few chapters. We first give an overview of TEEs (Section 2.1) and how they can enable confidential computing in untrusted environments. We then introduce WebAssembly as a technology that can be leveraged to build more secure, platform-agnostic TEEs (Section 2.2). Finally, we conclude this chapter by presenting provenance (Section 2.3), some of its associated challenges, and existing applications.

2.1 Trusted Execution Environments

A trusted execution environment (TEE) is a computation environment that on which relying parties can place a higher degree of trust than other software components on the same device [26]. TEEs leverage the underlying platform hardware to guarantee strong security and isolation guarantees that enable them to be dependable even in the presence of malicious software or a compromised OS in the untrusted side of the system, which we hereafter refer to as the rich execution environment (REE). This quality makes them more amenable to applications that require treating or storing security sensitive information such as cryptographic material, biometrics and digital wallet data.

In essence, a device is said to support a TEE if it is capable of the following [26]:

- **Isolation**: The device provides an execution environment for trusted code that is strongly isolated from the REE. Software in the REE cannot tamper with the execution of code running in the TEE or access its memory

- **Trusted storage**: The device provides trusted code with the ability of persisting data that is guaranteed confidentiality and integrity, even on an untrusted storage medium shared with the REE
- **Attestation**: The device provides a means for convincing a relying party of the trustworthiness and the characteristics of its isolated execution environment

TEEs exist in different realizations, which depend on where and how the TEE components are allocated in the underlying platform hardware. Figure 2.1 shows that TEEs can be completely implemented by external security co-processors or provided by a combination of CPU extensions and system-on-a-chip (SoC) components. The latter are typically referred to as *processor secure environments* [26].

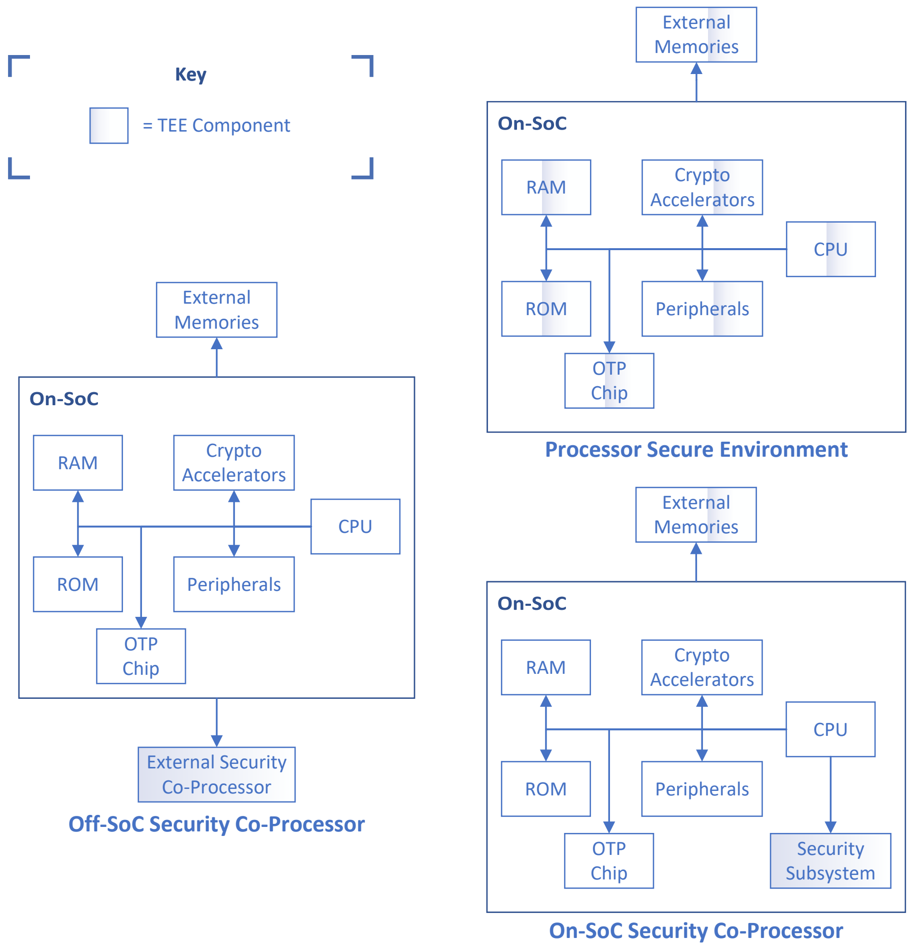
Key
= TEE Component
External Memories
On-SoC
RAM
Crypto Accelerators
CPU
ROM
Peripherals
OTP Chip
External Security Co-Processor
Off-SoC Security Co-Processor
Processor Secure Environment
Security Subsystem
On-SoC Security Co-Processor

Figure 2.1: Possible TEE realizations. Adapted from [21, p. 20].

2.1.1 Processor secure environments

Processor secure environments are a specific realization of TEEs that require extensions to the CPU instruction set architecture (ISA) and, depending on the technology and implementation of its specification, to co-located on-SoC components such as memory controllers. Two of the most widely deployed technologies for processor secure environments are Intel Secure Guard Extensions (SGX) and ARM TrustZone.

2.1.1.0.1 Intel SGX Intel introduced SGX as an ISA extension to the Skylake family of CPUs in 2015. New instructions have been introduced to secure code execution within hardware-enforced enclaves. These enclaves are protected from the rest of the system, whose security and integrity can be jeopardized by malicious applications and compromised OSs. The CPU reserves a portion of memory, called Processor Reserved Memory (PRM), for SGX functionality, and protects it from memory accesses that originate from outside enclaves [16]. This includes memory accesses from the REE OS kernel, hypervisor and System Management Mode (SMM), and direct memory access (DMA) from untrusted peripherals [16]. The REE software is tasked with allocating 4 KB pages to each enclave from a subset of the PRM, referred to as the Enclave Page Cache (EPC). On the other hand, the CPU ensures that each EPC page belongs to exactly one enclave by tracking their state in the Enclave Page Cache Metadata (EPMC) [16]. On REE software's demand, the CPU (1) loads trusted code and its data in the allocated enclave, then (2) marks it as initialized. During initialization, the contents of the enclave are cryptographically hashed to produce a MRENCLAVE value [26]. This hash, along with the signature of the code developer (MRSIGNATURE), form the basis of the SGX attestation mechanism, which can be leveraged to convince relying parties that they are communicating with the expected application, and that the application itself is running in an SGX enclave. A special `ecall` instruction allows for calling a specific enclave function [26] after switching to Ring 0. Conversely, an `ocall` instruction allows for an enclave calling a function in the context of a client application.

2.1.1.0.2 ARM TrustZone ARM first introduced the TrustZone technology for Cortex-A processors in 2004, and, in, 2016, adapted it to Cortex-M to cater for the limitations and requirements of specialized, resource-constrained devices. TrustZone for Cortex-A introduces two new protection domains for code and memory referred to as *secure* and *non-secure world*. The world in which the CPU is running is determined by an extra

33rd *non-secure* (*NS*) bit [53]. Memory transactions propagate this bit along internal and external buses to indicate the world they originate from and to restrict unprivileged code and untrusted peripherals from accessing protected data [53]. This is achieved by the TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA) components, which target on- and off-SoC memories respectively [26]. Finally, a special Secure Monitor Call (SMC) allows for trapping into the *secure monitor* (see Figure 2.2), which performs the necessary context switching operations (e.g., swapping stacks, registers, etc.) before switching to the secure world. Unlike SGX, where enclaves are hardware-defined entities, enclaves are realized logically by trusted software, i.e., an enclave platform, running in the secure world. This software needs to ensure that these logical enclaves cannot interfere with each other as well as with TEE and REE code and data.

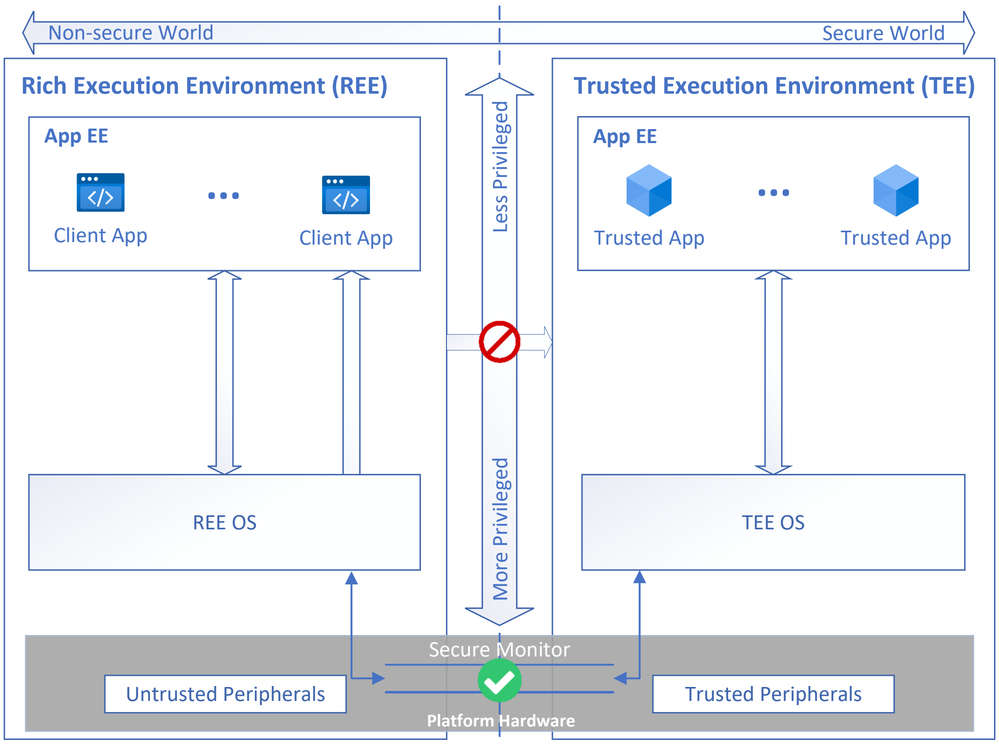

Figure 2.2: Architecture of TrustZone for Cortex-A processors.

2.1.2 Trusted Platform Module

Introduced by the TCG, a Trusted Platform Module (TPM) is an on-SoC tamper-resistant, embedded cryptographic co-processor. It provides a TEE for reporting and attesting to the integrity of system components—such as the bootloader, kernel image, user applications—and storing security or privacy sensitive data that is guaranteed confidentiality and integrity. This data can also be sealed, i.e., bound to the platform where it is stored. Much of TPM functionality is enabled by a set of 16 protected *Platform Configuration Registers* (PCRs) that can be used as accumulators (i.e., the value they contain can only be extended, not replaced) for measurements. The Integrity Measurement Architecture (IMA) proposed by IBM, for instance, uses PCRs extensively to measure and store the hash of binaries before they are executed [3].

Unlike SGX and TrustZone, TPM does not provide a native mechanism for isolated execution [53]. Therefore, it is often only used as a hardware root of trust (ROT) for storage and reporting.

2.1.2.1 Enhanced Authorization

Enhanced Authorization (EA) is an authorization mechanism introduced in TPM 2.0 that allows regular users or administrators to mandate that certain tests or actions be performed before an operation on an entity (e.g., a secret key) can be executed [62]. This is done by defining an authorization policy on a per-entity basis, that captures all the restrictions on the usage, and from which a digest value *authPolicy* is derived and later used to grant (or deny) access to an entity as part of an authorization session. During this session, a *policyDigest* value ($policyDigest_0 = 0 \dots 0$ initially) is extended with each fulfilled assertion as follows:

$$policyDigest' = \mathsf{H}(policyDigest \ \| \ \mathsf{Assert})$$

where Assert is obtained by concatenating the assertion's command code and its argument list. For a simple policy with two assertions, the formula expands to:

$$policyDigest' = \mathsf{H}(\mathsf{H}(policyDigest \ \| \ \mathsf{H}(0 \dots 0, \mathsf{Assert}_0)) \ \| \ \mathsf{Assert}_1)$$

A special case is the TPM2_PolicyOR [62] assertion used for policy disjunction, where the *policyDigest* is reset, and then extended with the entire original list of valid digests as follows:

$$policyDigest' = \mathsf{H}(0 \dots 0 \ \| \ \mathsf{TPM2\_CC\_PolicyOR} \ \| \ \mathsf{DigestList})$$

The resulting *policyDigest* is independent of which of the valid digests was used to fulfill the assertion.

The TPM2_PolicyAuthorize [62] assertion is another special case. It is the only command that can replace a *policyDigest*, although indirectly, instead of extending it. More specifically, TPM2_PolicyAuthorize allows an entity, identified by its public key $\mathsf{K}_{\mathsf{pub}}$, to accept a *policyDigest* provided that it is authorized by the entity's private key [62]. If that is indeed the case, the *policyDigest* is updated (not extended) as follows:

$$policyDigest' = \mathsf{H}(0\ldots0 \;\|\; \mathsf{TPM2\_CC\_PolicyAuthorize} \;\|\; \mathsf{H}(\mathsf{K}_{\mathsf{pub}}) \;\|\; \ldots)$$

At the end of the session, if $authPolicy = policyDigest$, access to the requested entity is granted. As with TPM2_PolicyOR, the final *policyDigest* is independent of the the accepted *policyDigest*, since it is always replaced with a hash of the public key of the authorized entity. This effectively solves a long-standing problem referred to as *PCR brittleness*, which typically arises when data is sealed against PCRs so that it can only be decrypted if the PCR values are correct [62]. The brittleness comes from the fact that data is strictly bound to PCR values, which can change, for example, in case of a BIOS update [62].

In essence, EA seeks to provide a unified way of authorizing access to entities by means of digest *authValue*s that can encapsulate relatively complex authorization policies while incurring storage overhead as little as the digest size. In fact, EA makes it even possible to express policies whose satisfiability is contingent upon certain dynamic values, e.g., the current hash of a secure object.

EA associates policies with entities on their creation, which solves some of the shortcomings of more traditional authorization methods. In fact, EA allows for a multitude of authentication methods to be supported, which includes biometrics like fingerprints [62, p. 128]. Multiple such methods can also be chained to achieve multi-factor authentication, whereby each of the authentication constraints included in a policy is resolved in an authorization session as a deferred check.

2.1.3 Remote attestation

The advent of cloud and grid computing, as well as the unrelenting large-scale expansion of the "internet of things" (IoT), have gradually supplanted the traditional model of local computation and storage in favor of distributed computing and remote storage. This change has, however, engendered a whole new class of possible misbehaviors that computer systems can engage in, which all come down to one central issue: software can be tampered with and there is no silver bullet mitigation to software attacks. This motivated the Trusted Computing Group (TCG) to introduce hardware-based mechanisms such as TPM to enable for the integrity of software to be reliably measured and attested to a relying party. This attestation is signed by the TPM with a TPM-resident attestation identity key (AIK). The resulting attestation certificate typically contains the hash of a target software and can be trusted by other entities by virtue of its signature.

More specifically, remote attestation is a mechanism whereby a prover conveys claims about the properties of a target system by providing evidence to an appraiser over a network [15]. The appraiser is a party that makes decision about other parties, whereas a target is a "party about which an appraiser needs to make such a decision" [15]. The prover and the target system can co-exist in the same system. Analogously, the appraiser and the party that relies on their verification results, can also be the same entity. Multiple models for attestation exist that involve different interaction patterns as well as allocation of roles, and which model is more appropriate largely depends on the specific application.

2.2 WebAssembly

WebAssembly (WASM) is a portable instruction and executable format for a generic stack-based virtual machine (VM). Its original stated objective is to enable resource-intensive applications, which heavily depend on either CPU or GPU acceleration, to run at near-native speed on web browsers. Numerous browser engines, including V8 (Chrome/Chromium), SpiderMonkey (Firefox) and WebKit (Safari), now include support for just-in-time (JIT) compilation and execution of WASM code. Unlike similar projects, most notably *asm.js*, WASM can be targeted by a wide variety of high-level programming languages besides JavaScript. In fact, any language supported by the LLVM toolchain (e.g., C, C++, Rust) with a WASM target can be compiled to WASM. Rust is an extremely popular choice because of its memory-, type- and thread-safety, which, combined with the inherent sandboxed nature of WASM, renders WASM an extremely attractive option for the realization of application sandboxes outside of web browsers. In light of this, several standalone JIT compilers and interpreters have been developed for WASM, including wasmtime [11], wasmer [66] and WebAssembly Interpreter (WASMI) [52].

WASM defines an instruction format for a generic VM, without specifying which "operating system" runs within the VM. This implies that, at least in the first few years after its inception, there was no standardized way for WASM applications to access resources and functionality of the underlying OS or web browser, other than VM-specific *host functions*. This limitation has elicited a decisive change of direction in terms of standardizing a system interface for WASM. This is also attributable to the growing application of WASM for sandboxing, evidenced by noteworthy projects like substrate [59], which uses a WASM VM for smart contract execution. A concerted effort by several stakeholders, especially the W3C, has resulted in the proposal of WebAssembly System Interface (WASI), which endows sandboxed WASM applications with a capability-based, POSIX-like interface to the underlying OS functionality.

2.3 Data provenance

The trustworthiness of data largely depends on the quality and completeness of information that pertains to it, as well as on the verifiability of its ownership [67]. Data provenance effectively serves the trustworthiness of data. In essence, provenance refers to metadata that summarizes the history of data [67]. More specifically, it describes how, when, and where a specific piece of data was created, utilized, transformed, and transferred to different owners over its entire chain of custody.

Provenance has numerous applications, ranging from big data platforms to security and healthcare. The provenance data can be used to estimate data quality, identify errors in data generation and replicate results [31]. By inspecting the entire chain of custody and the transformations (i.e., the *provenance chain*) that some data object has undergone since its creation, a relying party can assess its quality, verify that no violations have been committed with respect to its access control policy and, therefore, determine whether or not it is worthy of trust. However, this process requires that the provenance data also be trustworthy, as incomplete or altered provenance can mislead the auditor into trusting compromised data [67]. For provenance to be trustworthy, it needs to be collected and stored in such a way as to guarantee its confidentiality, privacy of the involved entities, integrity, availability, unforgeability, non-repudiation and chronological order (Zafar *et al.* [67]).

Besides the security requirements, provenance collection and storage entail different challenges, ranging from its intrusiveness with respect to the operating environment, performance and storage overhead, to interoperability. Interoperability is crucial because different provenance auditors may need to query and analyze the provenance data. An important step towards interoperability between provenance systems can be traced back to 2006 with the introduction of the Open Provenance Model (OPM), an interoperable provenance model whose stated object is to allow for provenance data to be exchanged between different systems ″by means of a compatibility layer based on a shared provenance model″ [45].

Chapter 3

Problem statement

As enclaves migrate through many different—and possibly vulnerable and/or compromised—TEEs (see Figure 3.1), their trustworthiness can be compromised. This entails that the integrity of an enclave and its data depends on its entire migration history, i.e., its provenance. The objective of this work is to design and build a fast, trustworthy data provenance framework for enclaves in heterogeneous distributed systems, which are typically comprised of nodes with different hardware and software configurations. The problem this work aims to tackle is two-pronged. First, we want to solve the issue of secure provenance collection in untrusted systems, where co-located malicious applications or compromised OSs can tamper with the collection process. Second, we want to answer the question of whether the collected provenance data can be stored in a trustworthy fashion without resorting to slow and expensive consensus algorithms. The answer to this last question will enable us to build a provenance system that is faster, cheaper to operate, and more apt for real-time applications, where the latency of enclave migration is critical and provenance data may be required in this step.

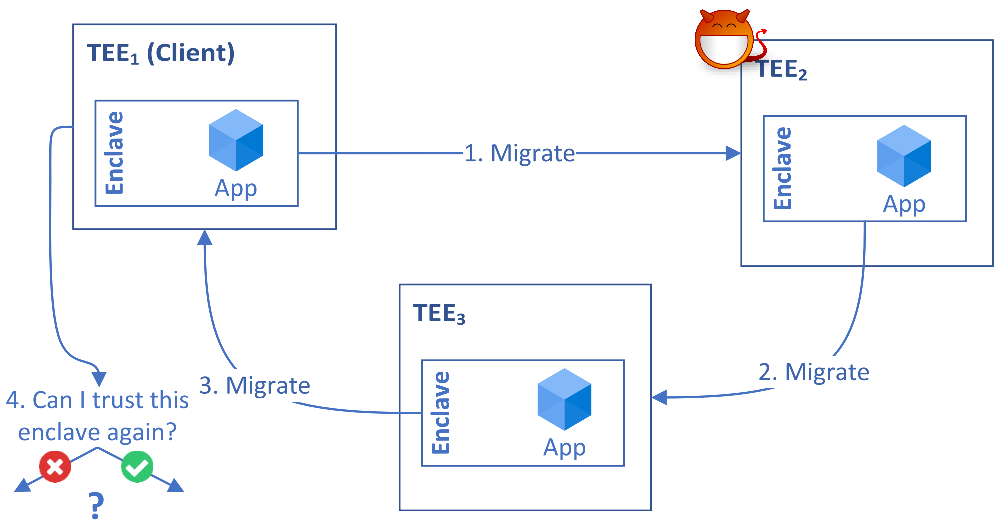

Figure 3.1: The client TEE cannot trust the enclave without knowing its entire migration history.

3.1 System model

We envision a QUICKPROV-enabled node within a distributed system (see Figure 3.2) as being composed of the following components:

- A TEE that provides an isolated execution environment for the execution of trusted applications (TAs). Depending on the realization, the TEE may or may not need (e.g., SGX) a trusted OS. TAs run inside enclaves provided by the TEE's own TA execution environment and can communicate with each other through an appropriate interprocess communication (IPC) mechanism. The TAs can also interface with the outer world through a TEE IPC agent
- A REE on top of which an untrusted OS hosts client applications that can communicate with TAs through a REE IPC agent
- Platform hardware that provides different ROTs to enable TEE functionality. The platform provides, at a minimum, a ROT for storage, which allows a TEE to store data on an untrusted storage medium shared with the REE

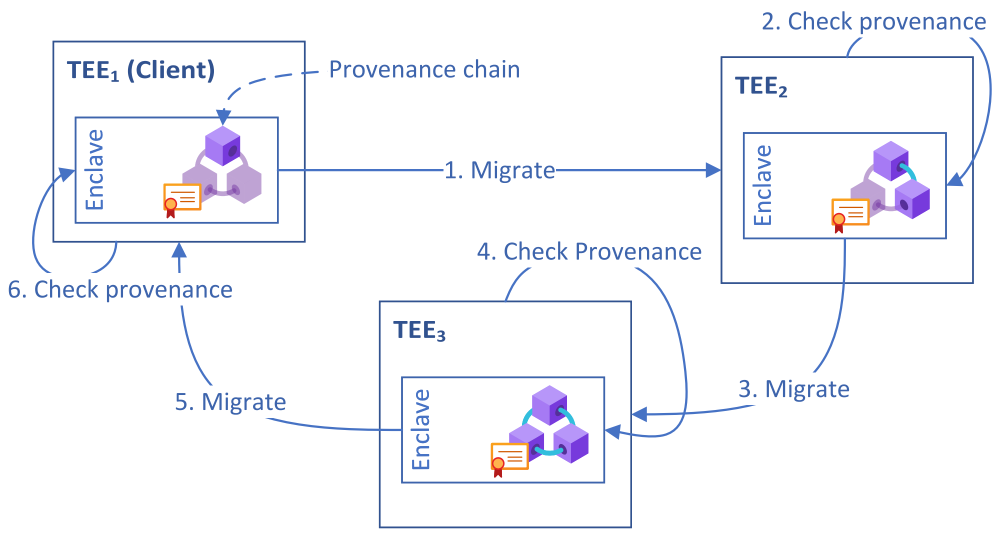

Figure 3.2: QUICKPROV system model. TEEs migrate enclaves among each other along with their provenance chain(s). When a destination TEE receives an enclave, it can check the provenance data to assess the enclave's integrity.

3.2 Assumptions

The QUICKPROV system hinges on the following assumptions to enable trustworthy provenance collection and storage:

- **TEEs can be attested:** TEE can convince relying parties of the trustworthiness of the software that runs within it, and of other components in the chain of trust that the TEE may depend upon. These other components may include a TPM, in which case the TPM itself endorses the components that depend on it, or a harware ROT, i.e., a module that the TEE trusts and relies on to perform cryptographic operations, that holds the TEE's hardware unique key (HUK).

- **TEEs provide trusted storage:** TEEs provide a hardware ROT for persisting data and cryptographic keys on an untrusted storage medium while guaranteeing data confidentiality and integrity. Since not all TEE can guarantee the freshness (i.e., replay protection or detection) of secure storage data, we also assume that the platform hardware provides some means (e.g., replay-protected memory blocks

(RPMBs), non-volatile monotonic counters) for ensuring data freshness. This is a realistic assumption, since most current flash storage mediums come equipped with a RPMB partition [26]

3.3 Adversary model

We introduce two types of adversary than can compromise the QUICKPROV system and we model the threats that they can pose to the system described in the previous section.

3.3.1 Network adversary

A network adversary is a malicious entity that has access to a communication channel between two nodes over a network. For this adversary, we assume the Dolev-Yao [19] threat model, meaning that they can virtually read, alter and forge all traffic between the two nodes. Furthermore, the extent to which the adversary can mount a successful attack is only limited by the strength of the cryptographic methods used to secure the channel.

3.3.2 Local adversary

A local adversary has access to a victim TEE and their objective is to compromise TEE functionality. This can either be an external adversary that exists in the REE (see Figure 3.3), or an internal adversary that resides in the TEE through, e.g., an authorized TA. Cerdeira *et al.* [13] provide an extensive description of the prominent security vulnerabilities that affect TrustZone-assisted TEEs, although many of those vulnerabilities can be found in other TEE realizations and trusted OSs.

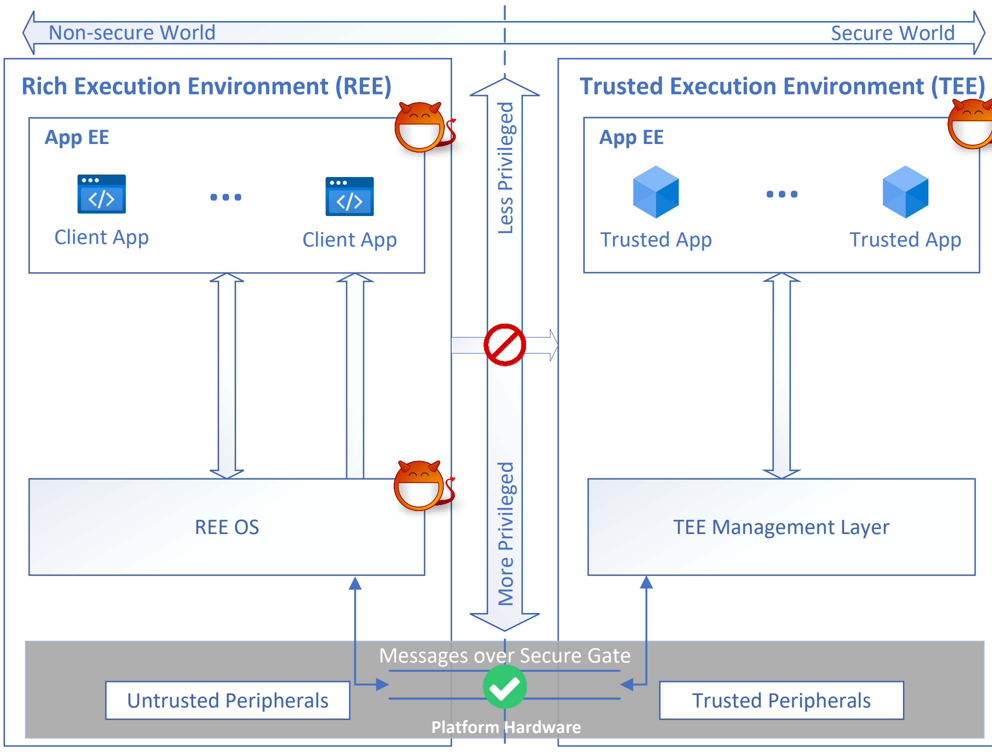

Figure 3.3: REE and TEE adversaries in the local adversary model. In the REE, the adversary can be a malicious client app or rootkit in the OS. In the TEE, an adversary can be a TA.

REE adversary The REE adversary can:

- Clone the TEE, including its trusted storage [63]
- Impersonate an authorized client by replaying its communication with a TA
- Impersonate an authorized TA to gain illegal access to TEE services [63]
- Exploit micro-architectural side-channels (e.g., Meltdown [41], Spectre [36]) to exfiltrate TEE private data or achieve arbitrary code execution
- Alter TEE behavior through a vulnerability (e.g., buffer overflow, use-after-free) in the TEE management layer [63]

TEE adversary The TEE adversary can:

- Exploit micro-architectural side-channels
- Alter the behavior of the TEE or other TAs by exploiting a vulnerability (e.g., buffer overflow, use-after-free)
- Gain illegal access to trusted storage data
- Exploit weaknesses in the TEE's cryptographic functionality

We consider micro-architectural side-channels to be outside the scope of this work.

3.4 QUICKPROV requirements and goals

We subdivide this section for QUICKPROV into security requirements and performance goals.

3.4.1 Security requirements

In order for the provenance to be trustworthy, the QUICKPROV system needs to collect and store provenance data such that the following security properties hold [67]:

PS-1 Confidentiality: The provenance data must not be readable by unauthorized entities since it may contain private information about the data it describes.

PS-2 Integrity: The provenance data must be tamper-evident, such that any alterations can be detected. This includes the corruption and truncation of provenance records.

PS-3 Unforgeability: Unauthorized parties cannot forge valid provenance records by altering existing entries or adding new entries without being detected.

PS-4 Non-repudiation: Authorized parties cannot add provenance records and later deny their involvement.

PS-5 Chronology: Provenance data must be chronologically ordered. Provenance is of little use if the exact order of operations performed on data is not preserved.

PS-6 Attestability: Provenance data must be collected and stored in such a way that the authenticity of the process that performs the collection and the security of the storage medium can be attested to.

In QUICKPROV, we do not consider availability, in the sense that provenance data must be "available at any time from anywhere" [67], as a necessary security requirement. Instead, we assume a weaker notion of availability, which is subsumed under **PS-2**, whereby provenance data cannot be truncated or deleted by unauthorized parties. Finally, privacy is outside the scope of this work.

3.4.2 Performance goals

A provenance system such as QUICKPROV should also aim to achieve certain performance goals [67] such that it does not disrupt or adversely affect normal operation of its execution environment.

PP-1 Minimal intrusiveness: The provenance collection process should not incur minimal performance overhead.

PP-2 Minimal storage overhead: The provenance data should not, ideally, require a considerable amount of storage space.

PP-3 Responsiveness: The provenance data should be complete and retrievable with minimal latency when it is queried.

Chapter 4

Design

This chapter presents the design of the components that constitute the QuickProv system. First, we present a secure enclave platform for TEEs that can enable QuickProv functionality while fulfilling the security requirements and performance goals formulated in Section 3.4. Hence, we describe the QuickProv framework and how it leverages the underlying enclave platform to achieve trustworthy data provenance collection and storage.

4.1 Enclave platform

We design an platform for the execution of TAs in isolated enclaves that can be written once and used everywhere, regardless of the underlying platform hardware and TEE OS. This is a necessary step in building a data provenance system that aims to target heterogeneous distributed systems. The portability is enabled by (*i*) a runtime that embeds a just-in-time compiler or interpreter for TAs written in a portable binary format, such as Java or WASM bytecode, and (*ii*) an abstraction layer through which TAs can access the underlying TEE functionalities, such as enclave-specific data and remote attestation, in a platform-agnostic fashion. The platform guarantees that enclaves are isolated from each other, the underlying runtime and the TEE. Enclave-level IPC and communication with the outer world is only possible through a runtime-enclave interface that exports a subset of the runtime functionalities that enclaves are allowed to utilize. This effectively configures a *two-way sandbox*: REE and untrusted peripherals cannot interfere with the TEE, and the TAs cannot escape the enclaves they reside in except through operations explicitly permitted by the runtime-enclave interface. The resulting architecture is illustrated in

Figure 4.1.

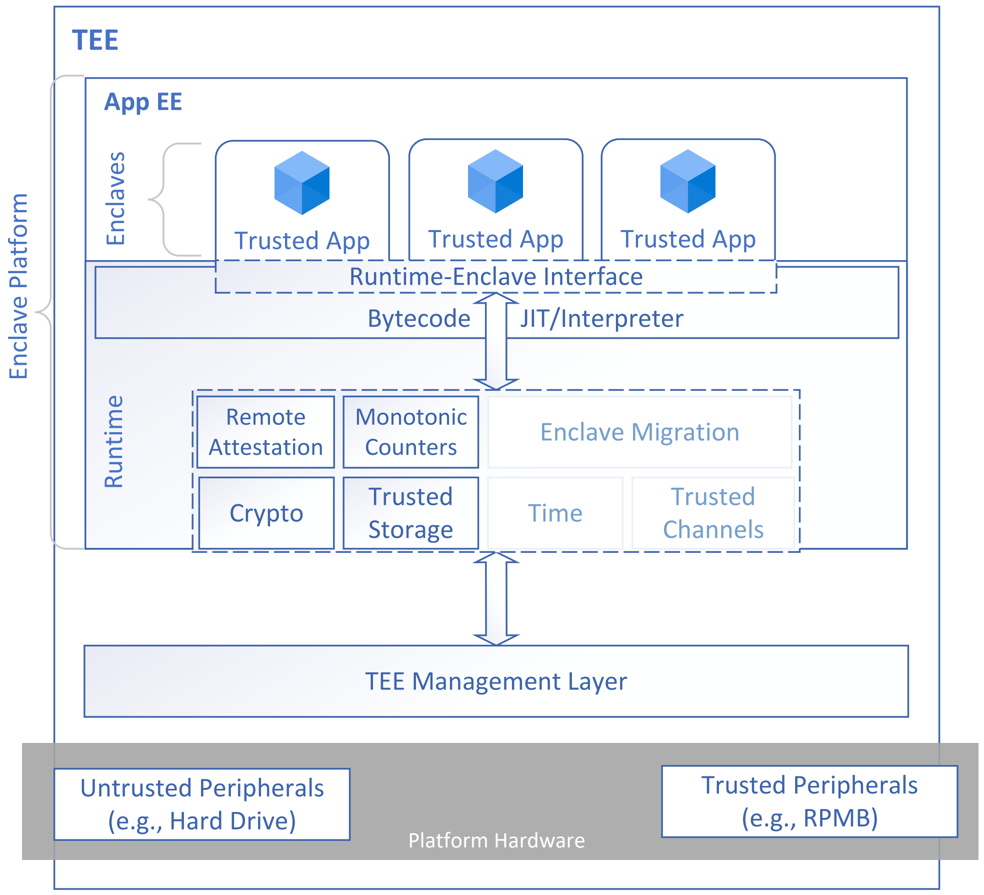

Figure 4.1: Architecture of the QUICKPROV enclave platform.

In the next few sections, we will describe enclave-specific data and remote attestation functionality. These will enable enclave-specific data and keys whose security can be remotely attested to, which is a fundamental building block for QUICKPROV's provenance attestation mechanism. Since we focus specifically on the collection, storage, and attestation aspects of QUICKPROV, trusted channels, enclave migration, and secure time providers (colored differently in Figure 4.1) are not considered—or are only partly covered—in the design.

4.1.1 Runtime- and enclave-specific data

We begin by describing the design of cryptographic and trusted storage providers that will enable both TA functionality, as well as functionality of the QUICKPROV system. These providers are part of the runtime and serve as an abstraction layer on top of the TEE's trusted core frameworks, which are accessible through an internal core application programming interface (API).

First, we need to endow enclaves with a notion of identity that can be used by QUICKPROV to identify enclaves when collecting provenance data as well as to restrict enclave access to trusted storage data based on their identity. The latter is necessary so that enclaves cannot interfere with each other's data, unless explicitly permitted. The same holds for runtime-private data, such as QUICKPROV provenance chains, which enclaves are not authorized to write. Hence, we design an authorization mechanism that will enable the runtime and TAs to express sufficiently rich access control policies to protect resources, such as persistent data objects, cryptographic keys and provenance chains. Lastly, we must design a runtime-enclave interface for TAs to enable access to this functionality.

4.1.1.1 Enclave identity

Each enclave must possess a unique identity that can be used to distinguish between enclaves running different TAs, and between different instances of the same TA. An individual instance of an enclave can be identified using a simple random identifier, but when a resource must be shared between several enclave instances (such as when a long-term secret must survive reboots), a purely random identifier will not suffice. In this case, we can deterministically derive the enclave identifier from properties of the enclave itself. These include:

- The hash of the TA binary hosted in the enclave
- The ID of the TA provided by the developer
- The TA developer's certificate
- The TA developer's full certificate chain

A good combination for a unique identifier that is also invariant to TA updates can be the pair given by the TA ID and developer's certificate or certificate chain. Additional properties can be added to disambiguate enclaves hosting binaries that are identical but are characterized by, e.g., a different certificate chain. This is enough to uniquely identify enclaves, at least locally. In QUICKPROV however, enclaves and data, including the provenance data thereto associated, can migrate through different TEEs, meaning that a global identifier is necessary to uniquely and consistently identify an enclave across migrations. To address this, we also add a public key to the identity derivation that uniquely identifies the TEE where the enclave originates from. This key can be, for instance, derived from the platform's HUK, or provisioned by the CSP.

4.1.1.2 Authorization

We propose an authorization model that enables the runtime and TAs to access TEE resources (e.g., persistent data objects, cryptographic keys, QUICKPROV provenance chains) protected by highly-expressive policies. Expressiveness provides more agency over who can access a TEE entity and how. For instance, TAs should be able to express whether an entity is private or shared among select other TAs. TAs should also, for example, be able to grant access to a specific TA or to any TA from a specific developer, as described in Section 4.1.1.1. Finally, the runtime needs to store private data, such as certificates and QUICKPROV provenance data, that needs to be appropriately protected from access by TAs and internal local adversaries (see Section 3.3.2).

To cater to these requirements, we design a mechanism based on EA (see Section 2.1.2.1) authorization model. Compared to regular access control lists (ACLs), EA has the following advantages:

1. Policies of arbitrary size and complexity can be applied without necessarily storing the assertions: in fact, only a hash representation of the policy is stored and associated with the entities. This hash is then checked against a reference value instead of matching a set of requested privileges with an ACL. This allows creation of policies of arbitrary complexity without increasing the memory footprint of storing them.

2. A convenient consequence of (1) is that EA policies can enable namespacing of resources: two or more resources with conflicting IDs can exist at any one time as long as their policy hashes are distinct. This allows trusted storage clients, for instance, to create objects that are private or shared among different applications (see Section 4.1.1.3), such that multiple namespaces, one for each possible $authPolicy$, can exist where data can be stored. This allows us to be arbitrarily expressive when specifying policies for shared entities.

Our design of EA provides assertions on enclave-specific properties—which include the instance and TA ID, the hash of the TA binary, its developer certificate and/or certificate chain—and trusted storage objects, such as access rights (read and/or write) and whether replay protection (e.g., RPMB) is enabled for a specific object.

Policy construction An EA policy can be described as a directed acyclic graph (DAG) where the nodes represent assertions. More specifically, it is a rooted tree, where the root is either the first assertion of a non-compound policy (i.e., one branch - see Figure 4.2) or the disjunction assertion of a compound policy (i.e., multiple branches - see Figure 4.3). It is worth noting that a disjunction assertion with only one branch is also a well-formed policy, although it represents a different yet functionally equivalent policy compared to the policy that only contains the one branch without the disjunction assertion as the root.

A `Command` describes an EA assertion, that is a node in the policy tree. A disjunction assertion is represented as an `Or` variant, whereas regular assertions are simple `Check` variants of a `Command`.

An `Or` is an ordered sequence of `OrBranches`, where each of the branches is either another policy (as depicted in Figure 4.2) or a previously computed policy digest, which can be thought of as a collapsed branch (see Figure 4.4).

A `Check`, on the other hand, is a regular assertion that may optionally take parameters. `Checks` can be made on properties that pertain to enclaves, the TAs they host, or trusted storage objects, such as:

- Enclave ID (i.e., TA instance ID)
- TA ID
- TA binary hash
- TA developer certificate
- TA developer certificate chain
- Trusted storage type (e.g., shared with REE or RPMB)
- Trusted storage object access rights (read and/or write)
- Trusted storage object sharing [22]

Some of these `Checks` make assertions of TA-specific properties that cannot be reliably supplied by the TA itself. For instance, the binary hash and ID properties of a TA must be provided by the runtime instead.

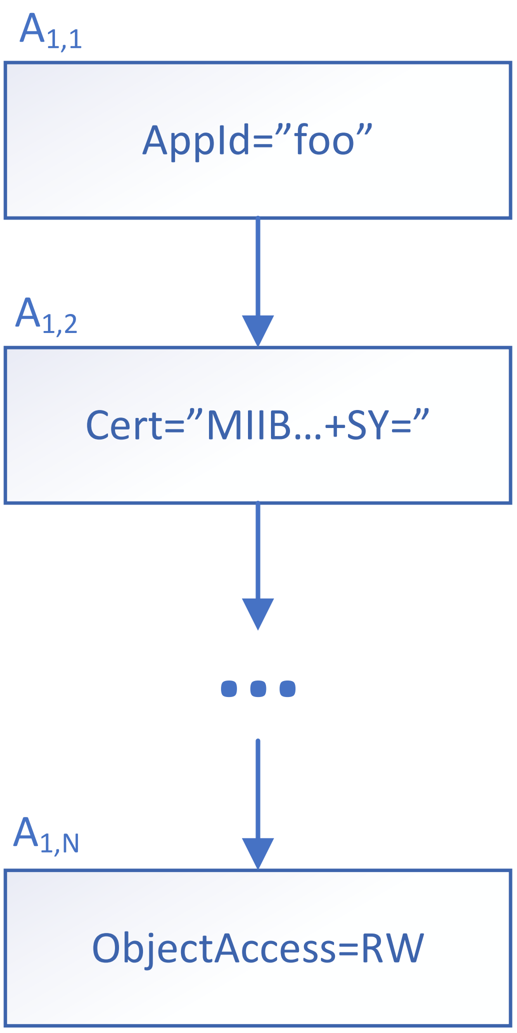

Figure 4.2: The tree of a non-compound policy. The root node is a simple assertion on the TA ID. Every node is a `Check` (i.e., an assertion).

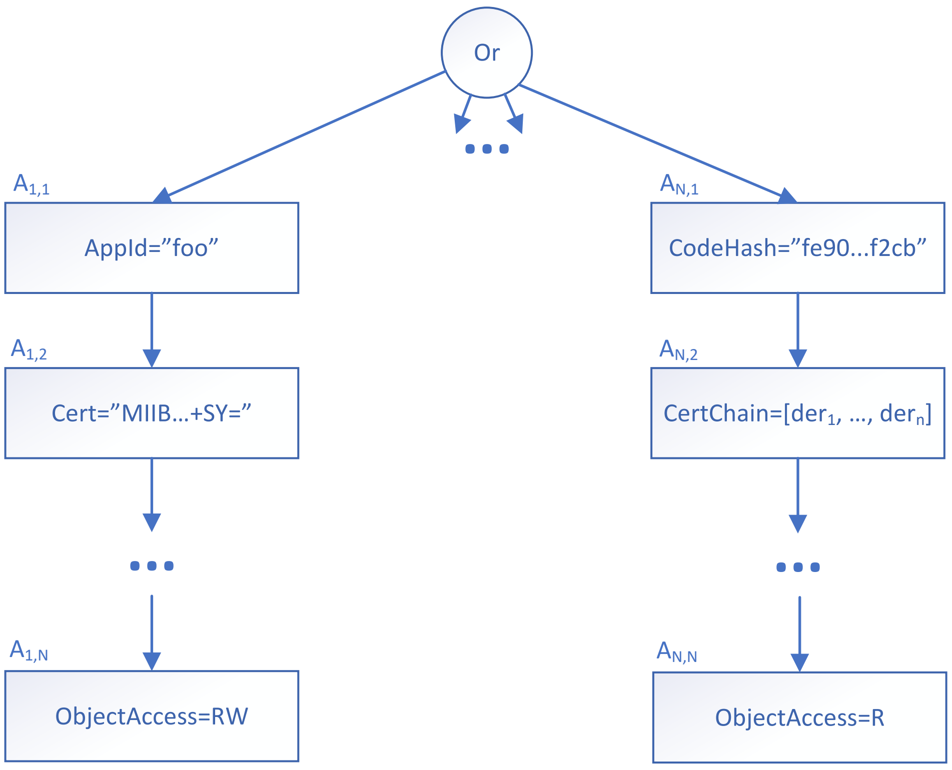

Figure 4.3: The tree of a compound policy. The root node is the `Or` assertion. The tree does not contain collapsed branches, i.e., branches with a precomputed digest.

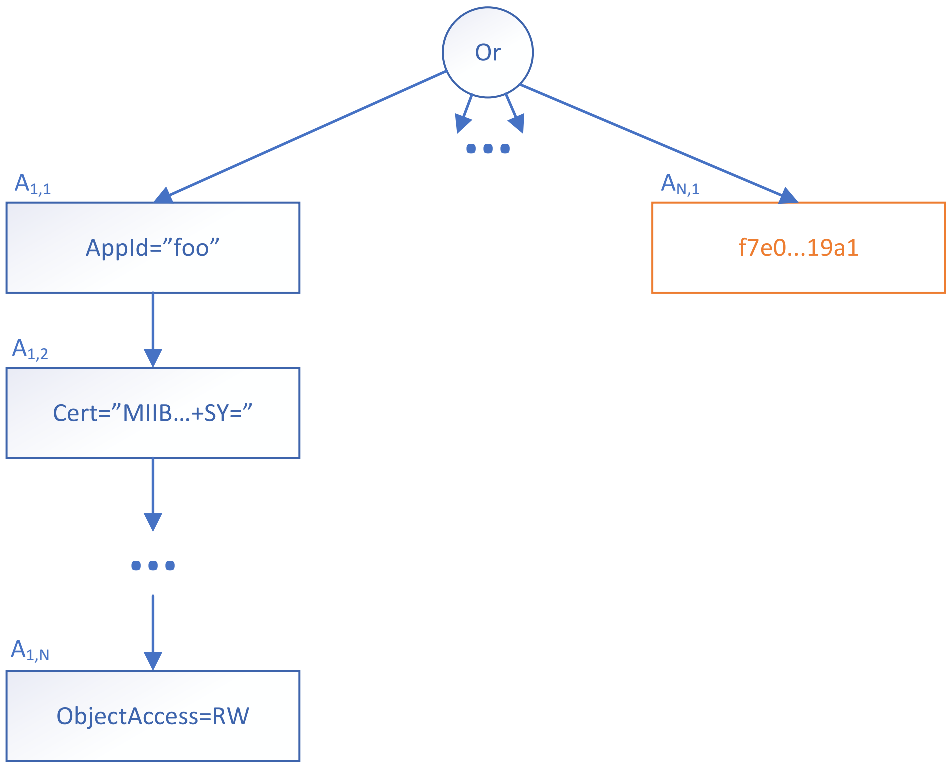

Figure 4.4: The tree of another compound policy. The root node is the `Or` assertion. The tree contains one collapsed branch, the rightmost one.

Policy sessions A policy session computes a policy digest from a sequence of assertions using the method described in Section 2.1.2.1. A session allows TAs to generate a policy digest by sequentially adding `Commands` to the session's state and, once the session is finalized, compute and retrieve its final digest. This digest can either be a *authPolicy* value or an *policyDigest*. An *authPolicy* is what is assigned to a resource when it is created, whereas a *policyDigest* is what is computed as part of an authorization session, i.e., when a TA tries to access the resource. The difference between the two is that *policyDigest*s are only updated when the `Checks` are fulfilled. Because of this, two different types of policy session exist, a `TrialSession` to generate *authPolicy* values, and an `AuthSession` to compute *policyDigest*s.

An `AuthSession` is, essentially, a `TrialSession` where all the `Checks` need to be valid, i.e., the assertions need to be fulfilled. Whereas in a

`TrialSession` the `Checks` need not be valid because the session is only used to compute an *authPolicy* value, an `AuthSession` has to actually evaluate the `Checks` before updating the *policyDigest* value of the underlying session. This is because some of these `Checks` can be used to express capabilities for, e.g., object access rights, that need to be checked at runtime.

4.1.1.3 Trusted storage

We design an interface that enables the runtime and TAs to store and retrieve persistent data, cryptographic keys and QUICKPROV provenance chains that are guaranteed confidentiality, integrity and freshness. This interface leverages the EA authorization mechanism described in Section 4.1.1.2 and the underlying TEE trusted storage facilities to provide a flexible and trustworthy storage mechanism. Since the storage model proposed by Global Platform (GP) only provides one private storage space per TA, the main issue to tackle is allowing multiple TAs running within the same runtime to have their own private storage namespaces.

The EA mechanism provides us with a convenient way of enforcing namespacing, which effectively allows TAs to create and store objects in namespaces that are either private—and, thus, inaccessible to other TAs—or in shared namespaces containing TAs specified in their objects' associated policies. In the underlying storage, we prefix every object identifier with an *authPolicy* value (i.e., the 32-byte SHA256 policy digest) of the policy assigned to the object.

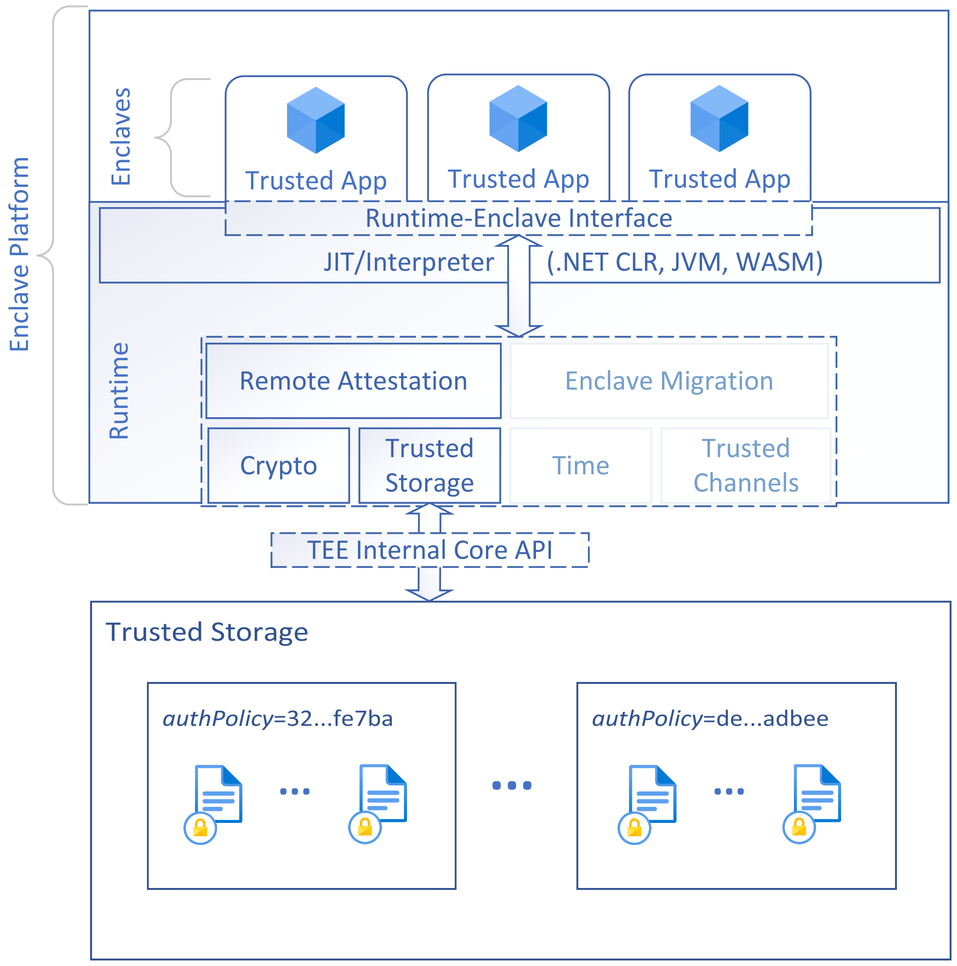

Figure 4.5: The proposed trusted storage model extends the standard GP model with EA policy-based authorization. Each TA (and the runtime) owns a set of *policyDigest* values (or is able to recompute part of them) which can grant them access to specific namespaces.

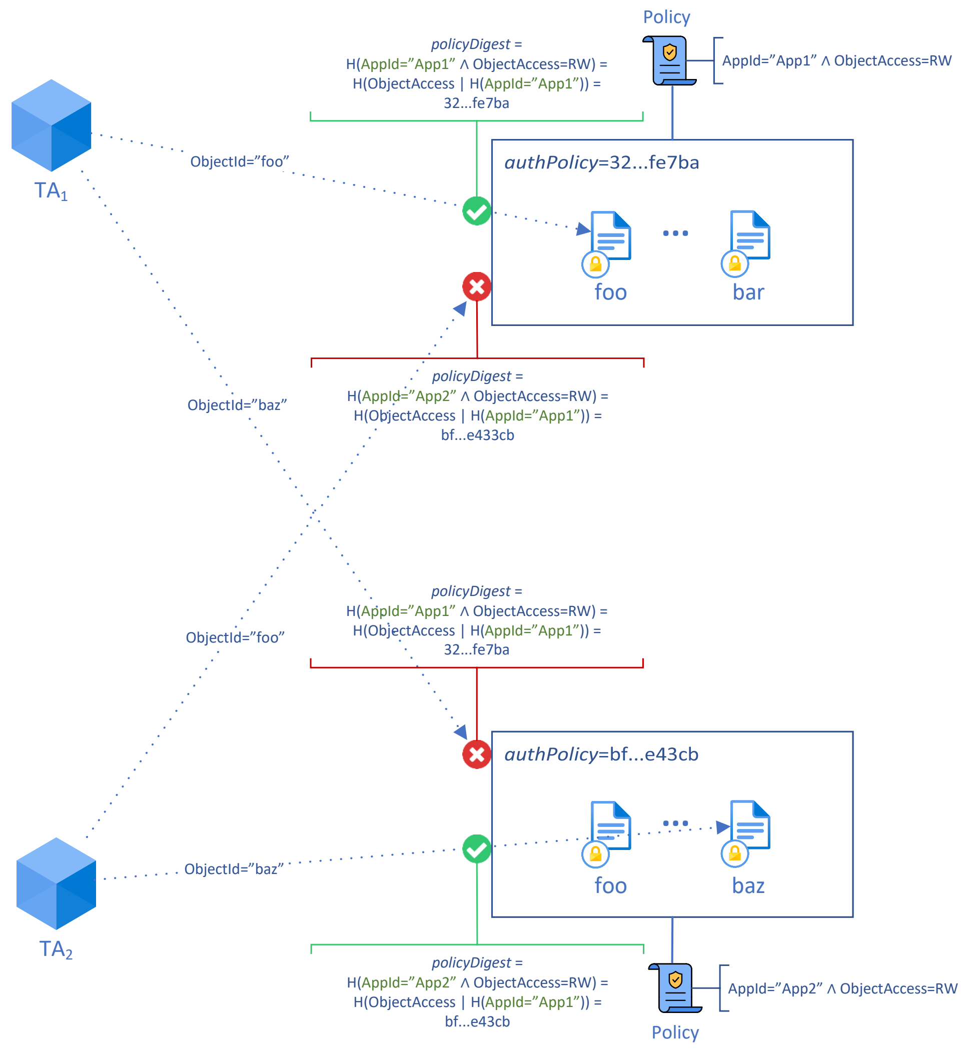
Policy
AppId="App1" ∧ ObjectAccess=RW
policyDigest = H(AppId="App1" ∧ ObjectAccess=RW) = H(ObjectAccess | H(AppId="App1")) = 32...fe7ba
authPolicy=32...fe7ba
foo
bar
TA1
ObjectId="foo"
ObjectId="baz"
policyDigest = H(AppId="App2" ∧ ObjectAccess=RW) = H(ObjectAccess | H(AppId="App1")) = bf...e433cb
policyDigest = H(AppId="App1" ∧ ObjectAccess=RW) = H(ObjectAccess | H(AppId="App1")) = 32...fe7ba
ObjectId="foo"
authPolicy=bf...e43cb
foo
baz
TA2
ObjectId="baz"
policyDigest = H(AppId="App2" ∧ ObjectAccess=RW) = H(ObjectAccess | H(AppId="App1")) = bf...e43cb
AppId="App2" ∧ ObjectAccess=RW
Policy

Figure 4.6: Two different TAs trying to access objects from different namespaces. There is a one-to-one correspondence between policies and namespaces. Assertions highlighted in green are resolved by the runtime. Access is granted only if the *policyDigest* value computed from the authentication session matches the *authPolicy* of the tentatively accessed namespace and there exists, within this namespace, an object with the specified `ObjectId`. TA_1 and TA_2 fail to access objects `baz` and `foo`, respectively, because *policyDigest* $\neq$ *authPolicy*.

For instance, an object with identifier *foo* can exist in multiple namespaces (private or shared), each protected by a different EA policy. In order for a TA to access a specific object, besides knowing its identifier, they need to be able to reconstruct its assigned *authPolicy* value, as depicted in Figure 4.6.

4.1.2 Remote attestation

This section presents a remote attestation mechanism that will enable remote parties to ascertain the identity of an enclave—and, transitively, the TA contained within—it is communicating with. Besides ascertaining the identity of an enclave, this mechanism will also provide the remote party with evidence on the state of the underlying platform. This will enable the remote party, for instance, to verify that certain security properties hold of the enclave's data.

We design this functionality by reusing much of the runtime- and enclave-specific data machinery presented in Section 4.1.1. This enables us, for instance, to design an attestation functionality that certifies a private key persisted on trusted storage as being protected by a particular EA access control policy.

4.1.2.1 Key attestation

We describe a key attestation mechanism that allows for binding the public component of a keypair to the EA policy that protects its private counterpart. An enclave can only use this keypair if it fulfills all the assertions specified in its policy. This mechanism will enable a remote party to establish a communication channel with an enclave that is contingent upon certain security properties of the private key with which the enclave binds and secures the channel. For instance, the remote party may demand that this private key be only accessible by a specific enclave (i.e., TA instance), which ensures that the key is ephemeral and can only be used to bind the channel, and, not less importantly, it cannot be used by any other TEE entity.

Evidence of such security claims, e.g., a private key is only acccessible by a specific enclave, is bundled in an X.509 certificate. X.509 certificates enable us to leverage a public-key infrastructure (PKI) that makes it possible to establish a chain of trust for attestation, whereby evidence can be bundled and signed by an endorsing party. These certificates can be easily integrated with transport layer security (TLS) to establish trusted channels between enclaves, i.e., TLS channels that mandate one-way or mutual attestation in the handshake process. In particular, we leverage the X.509 v3 format [32] custom extensions field to bundle the claims on the security of a key, where we simply embed its EA access control *policyDigest*.

Finally, the enclave runtime, which handles the EA-based authorization for data and keys persisted on trusted storage, manages a local certificate authority (CA) that signs the key attestation certificates. This certificate is signed by another higher-level CA that, in addition, attests to the correct operation of the device and the runtime.

4.1.2.2 Attestation layers

Before trusting the claims about a key's storage security contained in an attestation certificate, remote parties first need to ascertain the integrity of the runtime that produced it. The runtime integrity depends, in turn, on the trustworthiness of the underlying trusted computing base (TCB). In our model, we assume that the integrity of these components is measured either before or at runtime startup. Depending on the platform, this may require for an original equipment manufacturer (OEM) or some other trusted third party to obtain evidence on the measurements. Since this mechanism is usually vendor-specific, remote parties communicating with an enclave are oblivious to the internals of the enclave's platform attestation process.

The vendor-specific attestation certificates form the right-most part of the X.509 chain of trust depicted in Figure 4.7. As part of platform verification process, the OEM signs the local CA key that the runtime utilizes to generate the left-most part of the chain, which is composed by the sole key attestation certificate. This allows for verifying the OEM's endorsment using publicly-available PKI tooling, such as OpenSSL [61] or *rustls* [1]. Figure 4.7 illustrates the complete chain of trust between the attestation certificate of an individual key and a root CA recognized by remote parties.

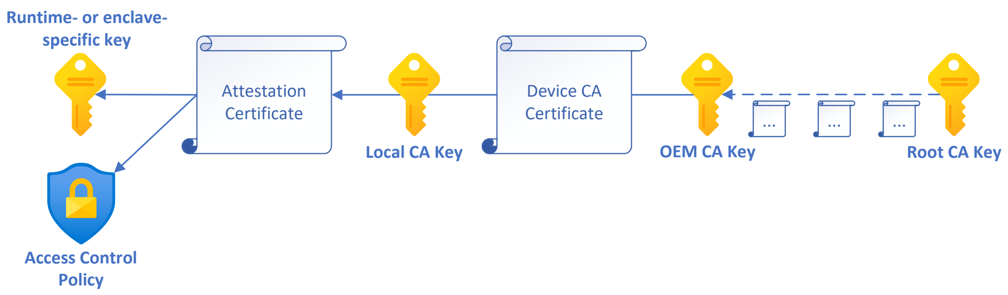

Figure 4.7: The attestation evidence chain of trust. The leaf, i.e., left-most, certificate contains the EA access control policy of a keypair and its public component. This certificate is signed by the local CA. Finally, the local CA is endorsed by a further chain of authority certificates—from the OEM CA to the root CA—produced when initializing the runtime.

4.2 QUICKPROV

We begin by presenting the structure and the properties of a QUICKPROV provenance chain, as well as the various types of records it contains. Hence, we describe how QUICKPROV leverages the underlying TEE's enclave platform to achieve trustworthy provenance collection and storage. Finally, we explain how a TEE can convince a remote party that some provenance data is trustworthy through a provenance attestation mechanism. This is important since QUICKPROV provenance data can also migrate through systems along with the enclaves or objects they are associated with.

4.2.1 Provenance chain

A provenance chain is a sequence of interlinked, append-only logs that hold provenance data specific to the log's owner, i.e., the TEE that owns the log. The entries of each log are cryptographically signed with a signing key generated by the TEE, which ensures that no other party can append data to it, while enabling clients to verify its integrity using its signing key's public component. The next few subsections describe the components that constitute a provenance chain in greater detail.

4.2.1.1 Record

A record is a tuple $\langle index, previous, data, signature \rangle$ [35] where:

- *index* is the position of the record in the log
- *previous* is the hash of the preceding record, if present
- *data* is the actual payload of the record. A provenance record, for instance, contains provenance information
- *signature* is the signature of the current record, which includes *index*, *previous* and *data*

4.2.1.2 Log

A log is a single-writer sequence of records signed with the owner's private key. The public key counterpart identifies the log (and its owner) and is used to verify the integrity of the log, since any unauthorized modifications would break the signature check, and forging records requires having the owner's private key. This effectively enables tamper-evidence. Finally, the log is monotonic (append-only), meaning that a record can only be added at the end of the log and are cryptographically bound to the predecessor by its *signature*. A log is well-formed if [35] it is totally ordered, i.e., no two records have the same predecessor, connected, i.e., all records are (transitively) connected, and all the records are signed with the same key.

4.2.1.3 Chain

A chain is a multi-writer sequence of logs that are mutually linked with each other. Two special marker records, beginning of feed (BOF) and end of feed (EOF), signal the point where a log in the chain starts, and where it ends. These records also point to the preceding and succeeding logs in the chain respectively, identified by their respective public keys. Practically, the BOF and EOF markers are records where *data* contains either the previous or next authorized public key. When the current writer of the chain finalizes their log, e.g., because the resource it describes needs to be migrated, they authorize the next writer by placing their public key in the EOF marker. Once the change of ownership is complete, the new writer adds a new log to the chain and primes it with a BOF marker that contains the public key of the previously authorized writer. These markers allow for reconstructing the entire chain from any log as well as ensuring that only the authorized writers can write to it.

4.2.2 Provenance collection

We want to collect records for IO operations and enclave migrations, which would enable TEEs to reconstruct the provenance history of a specific object or enclave of interest. The provenance collection process needs to be trustworthy, i.e., fulfill the security requirements in Section 3.4.1, lightweight, and efficient, meaning that it needs to be minimally intrusive (see Section 3.4.2). Since the TEE functionality providers are inside the runtime, the provenance collection module must be hardwired into the runtime. This module needs to handle the following events of interest:

- Trusted storage data (e.g., data objects, cryptographic keys, etc.) migration
- Trusted storage IO
- Enclave migration

When an object or enclave is created, a fresh provenance chain is instantiated. A provenance chain for a trusted storage object, for instance, will contain information on the IO operations performed on it as well as its migrations. Whenever a chain is instantiated or migrated to another TEE along with the resource it tracks, the QUICKPROV collection module creates and adds a new log to the chain as follows:

1. Generate a fresh keypair. The private key signs the provenance records, while the public key identifies the log
2. Create a new log and bind it to the new keypair
3. Prime the log with a BOF marker. The marker contains the public key that identifies the previous log in the chain. If this is the first log in the chain, then BOF points to *null*.

Once the log is primed, provenance records can be added to the log whenever an event of interest occurs.

4.2.3 Provenance storage

The provenance chains need to be persisted while guaranteeing their confidentiality, integrity and freshness. This enables provenance data to survive TEE reboots and protects it from unauthorized access by local adversaries and offline replay attacks. The QUICKPROV storage module can be either a separate enclave that the runtime entrusts with storing the collected provenance or a component hardwired into the runtime itself. This module leverages the enclave platform's trusted storage functionality and its authorization model to securely store the chains. More specifically, it creates a policy that grants it full exclusive access, and it assigns it to the chain. This prevents other entities in the TEEs, such as TAs, from reading or writing to the chain. The chain is then persisted on trusted storage, which guarantees its confidentiality, integrity, and freshness, and protected from unauthorized access by TAs with the access control policy. If the TEE does not guarantee freshness natively, we assume that the platform hardware provides a freshness mechanism (see Section 3.2) for protecting the chains from offline replay attacks. For instance, the provenance storage layer can increment a hardware monotonic counter whenever a chain is updated and then store the new records (*inc-then-store*, see Figure 4.8) along with the counter value on the trusted storage. Alternatively, if the hardware provides a RPMB, the storage layer can use it to securely store the latest state of the chain, e.g., the *signature* field of the last record in the chain.

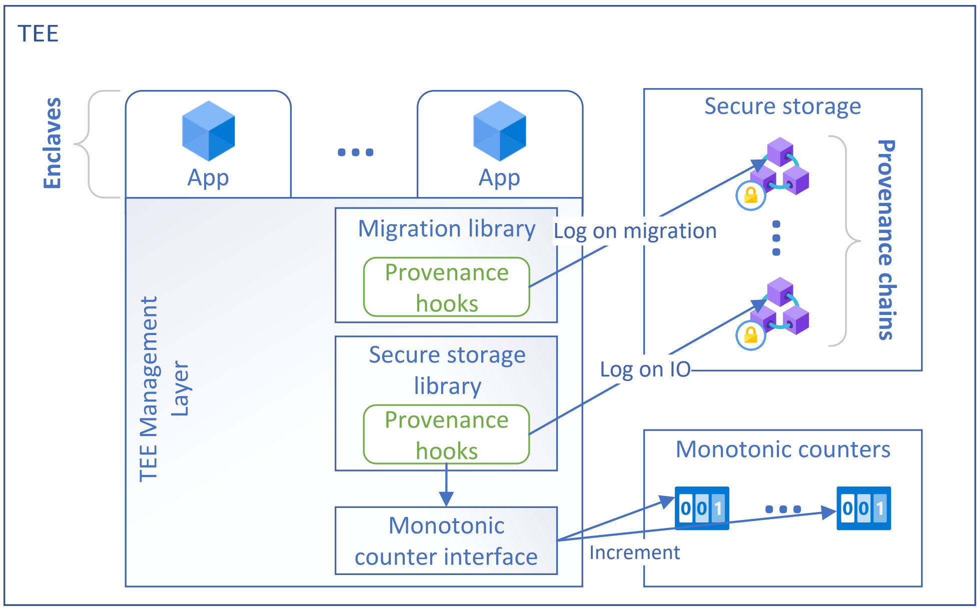

Figure 4.8: Provenance freshness can be guaranteed by first incrementing a monotonic counter and, then, storing the updated provenance data along with the counter value.

4.2.4 Provenance attestation

It is not enough to provide trustworthy provenance collection and storage. One of the requirements in Section 3.4.1 is that QUICKPROV be attestable, meaning that there needs to be a way to prove to an interested party that the provenance data was collected and stored securely. Provenance data is meaningless if there is no reliable means for verifying the trustworthiness of the process that generates it and the security of the storage medium it is persisted on. Therefore, we propose a provenance attestation mechanism that QUICKPROV-enabled TEEs can leverage when migrating provenance data to another TEE. The provenance attestation extends the key attestation mechanism presented in Section 4.1.2 by adding evidence on the following security claims:

- **Collection security**: The QUICKPROV collection module is trustworthy. This guarantees that the collection process is correctly isolated and unauthorized TEE entities cannot interfere with it

- **Private key security**: The private key used to sign the associated provenance chain is only accessible to QUICKPROV collection module. This ensures that no other TEE entity can use it to forge valid provenance records

- **Chain security**: The provenance chain is only accessible to the QUICKPROV storage module. This prevents unauthorized TEE entities from reading or writing to the chain

Since the collection module is part of the runtime, the evidence of its trustworthiness (e.g., a signed hash measurement of the runtime) needs to be bundled in the device CA certificate, which is signed by the OEM CA. As for private key and chain security, the platform can provide the evidence to these claims in the form of EA access control *policyDigest*s that certify that these resources are only accessible by QUICKPROV modules. For a remote party to trust an attested provenance chain, they can check the EA policy digests of the keypair and the chain against a list of known-good reference values.

Chapter 5

Implementation

This chapter presents a PoC implementation of the enclave platform and QuickProv system presented in Chapter 4.

5.1 Enclave platform

We have developed an enclave platform based on the OP-TEE [40] trusted OS and, on top of it, we built a runtime for WebAssembly trusted applications (WTAs) based on the WASMI [52] WASM interpreter. We emulated the ARM TrustZone [7] hardware-based isolation that OP-TEE is based on using a QEMU [54] configuration that virtualizes an ARMv8-A board. We also leveraged the Teaclave TrustZone SDK [6] for OP-TEE, which "provides abilities to build safe TrustZone applications in Rust". Rust provides strong guarantees for memory and type safety, which adversaries can try to compromise by exploiting buffer overflows or type confusion attacks. The resulting architecture is depicted in Figure 5.1.

5.1.1 Isolation barriers

The OP-TEE trusted OS isolates code and data from the REE (i.e., the untrusted OS) by leveraging TrustZone hardware-based isolation, while the runtime isolates the WTAs from each other and from the outer world. This effectively provides a secure environment for two-way, sandboxed execution of WTAs inside enclaves, which effectively realizes the design presented in Section 4.1. In particular, the two-way sandbox guarantees that enclaves cannot be tampered with by REE or TEE adversaries and, at the same time, the WTAs that runs inside the enclaves cannot misuse REE and TEE resources. In fact, the sandbox provides layered isolation across three execution worlds, namely, between WTAs, between WTAs and TEE, and between TEE and REE.

5.1.1.0.1 Inter-WTA isolation Each WTA runs within a separate enclave hosted by a WASMI instance launched by the runtime. The runtime ensures full inter-WTA isolation. This is because the WASMI interpreter assigns a separate linear memory space, i.e., a contiguous array of bytes, to each enclave. This memory space can only be manipulated with load and store instructions and by specifying a relative virtual address, i.e., an offset relative to the base address, within this space. In fact, there is no concept of raw pointers, which ensures that WTAs cannot reference arbitrary memory outside of their assigned linear space.

5.1.1.0.2 WTA-TEE isolation The partitioning of memory into isolated linear spaces for each WTA offered by WASMI, combined with the inherently limited instruction set provided by WASM, which does not support raw pointers and trap instructions, guarantee that WTAs, at least theoretically, cannot evade the enclave. The only way for WTAs to communicate with the outer world is through host functions provided by the hosting WASMI instance. This means that, in order to escape the sandbox, a WTA must exploit a vulnerability in the WASMI interpreter or runtime, a difficult task given WASM's emphasis on security due to its original use as a browser-based platform.

5.1.1.0.3 TEE-REE isolation Isolation between the TEE and REE is achieved at the bus controller level by the TrustZone-based secure privileged layer provided by OP-TEE. Code within this layer runs in the *secure world*; memory transactions are restricted using an extra bit (*NS* bit) that prevents the CPU and other other DMA masters and peripherals from hijacking TEE memory. The isolation level that OP-TEE provides is, thus, stronger than hypervisor-based isolation, although practical side-channel attacks, both digital [68] and physical [10], have been demonstrated against TrustZone.

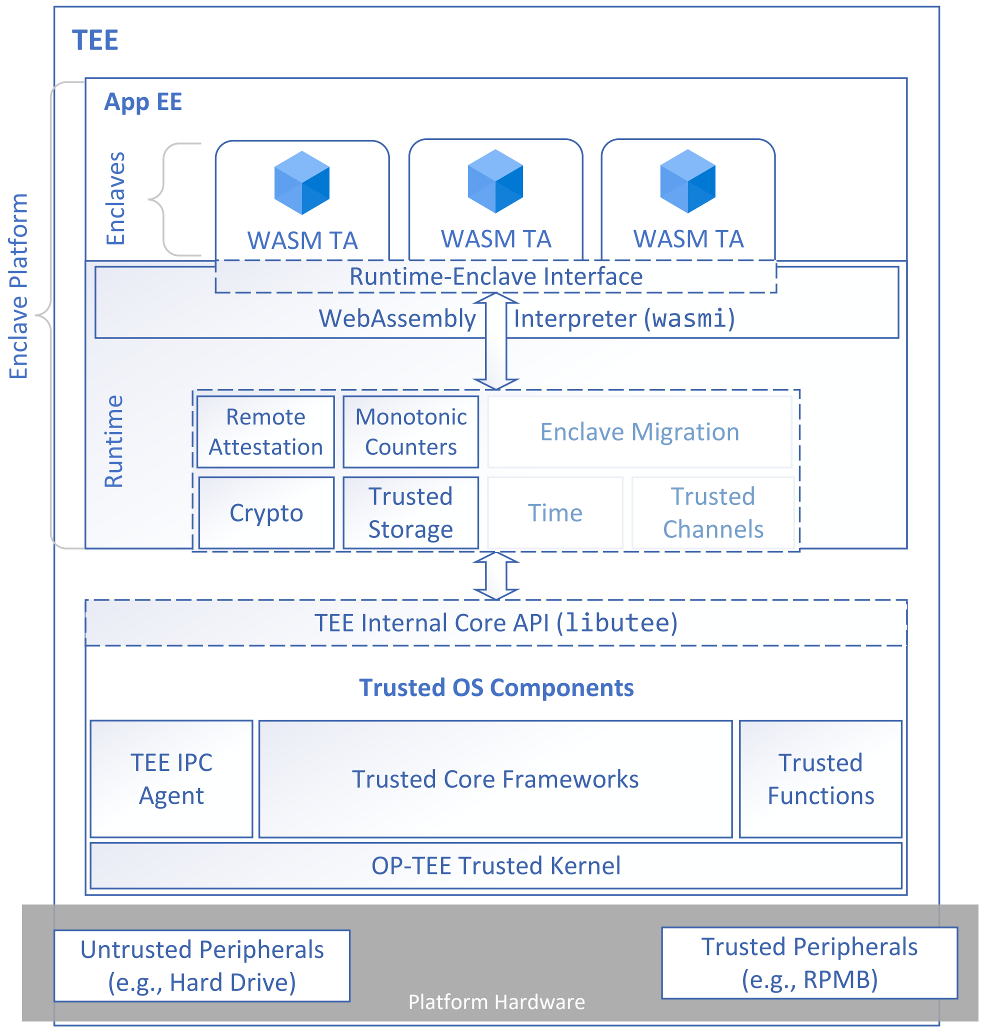

Figure 5.1: Implementation of the QUICKPROV enclave platform.

5.1.2 WebAssembly Trusted Application

A WTA is an application written in any high-level language (e.g., C++, Rust, etc.) that supports the WASM target (i.e., *wasm32-unknown-unknown*) of the *LLVM* [37] compiler infrastructure. This application needs to come packaged with a JSON manifest file that can be generated with a WTA signing tool. The manifest file contains the following fields:

- Application ID: This uniquely identifies the WTA

- Code signature: The signature of the WTA binary
- Developer certificate: This contains the developer's public key, which the runtime uses to verify the signature of the WTA when it is loaded
- Developer certificate chain: The chain of trust that comprises the root CA and the intermediate certificates
- Manifest signature: The signature of the application ID and code signature attribute pair

An example of WTA manifest file is shown below:

```
{
  "manifest": {
    "app_id":"99558fd0-47d4-47de-a827-14cdb3915c1c",
    "code_hash":"e4df...0507"
  },
  "developer_certificate": "...",
  "developer_certificate_chain": ["...", .., "..."],
  "manifest_signature": "a8bd...8402"
}
```

Listing 5.1: WTA manifest file

Since WTAs are built for the *wasm32-unknown-unknown* target, they do not have access to the WASI. The runtime provides the WTAs with WASI-like functionality through a runtime-enclave interface realized with a set of WASMI host functions. WTAs can access these functions through the Rust bindings crate (*wasm-enclave-bindings*) provided by the runtime (see Section 5.2.2).

5.1.3 Runtime

The runtime (*wasm-enclave-runtime*) is an OP-TEE TA written in Rust that embeds a WASMI runtime that is designed for accommodating multiple WTAs in separate enclaves, although the PoC currently supports the execution of only one WTA. Each enclave is an instance of a WASMI module equipped with a separate memory space and a set of host functions that enable the WTAs to access runtime and TEE functionality.

5.1.3.0.1 Runtime The `Runtime` data type contains global state that is shared among WTAs and runtime modules.

```
struct Runtime {
    /// Resolver for trusted storage host functions
    sse_resolver: SseExternalsResolver,
    /// Resolver for cryptography host functions
    crypto_resolver: CryptoExternalsResolver,
    /// Shared trusted storage instance
    storage: Arc<Mutex<StorageExternal>>,
    /// Shared runtime context
    context: Arc<RwLock<RuntimeContext>>,
}

struct RuntimeContext {
    /// List of loaded WTAs
    apps: Vec<App>,
    /// Currently active WTA
    current_app_index: usize,
    /// Enclave platform CA certificate
    ca_cert: Certificate,
}
```

Listing 5.2: `Runtime` data type

An implementation on top of `Runtime` is provided that allows for loading WTAs from disk or memory.

```
impl Runtime {
    /// Reads a WASM binary from the supplied path, then creates
    /// and returns an [`App`](wasmi::Module) from the binary.
    pub(crate) async fn load_app_from_path(
        &mut self,
        binary_path: &Path,
        manifest_path: &Path,
    ) -> Result<App, RuntimeError>;

    /// Reads a WASM binary from the supplied buffer, then creates
    /// and returns an [`App`](wasmi::Module) from the binary.
    pub(crate) async fn load_app_from_slice(
        &mut self,
        code: &[u8],
        manifest: &[u8],
    ) -> Result<App, RuntimeError>;

    /// Migrates an enclave and its state to a remote runtime
    /// instance
    pub(crate) async fn migrate_app<A: std::net::ToSocketAddrs>(
        &self,
        to: A,
        instance: ModuleInstanceRef,
    ) -> Result<()>;
}
```

Listing 5.3: `Runtime` data type

5.1.3.0.2 App The `App` data type represents a WTA instance loaded in a WASMI enclave.

```
pub(crate) struct App {
    /// WTA code
    code: Vec<u8>,
    /// Wasmi enclave
    module: wasmi::Module,
    /// WTA manifest
    manifest_envelope: ManifestEnvelope,
    /// Enclave-specific keypair
    instance_kp: Keypair,
    /// Enclave identity
    instance_id: Uuid,
    /// Provenance data for the enclave
    prov_feed: MultiFeed<RandomAccessSse>,
}
```

Listing 5.4: `App` data type

5.2 Runtime- and enclave-specific data

We have developed a Rust crate (`libsse`) that implements the EA authorization mechanism and trusted storage interface described in Section 4.1.1.2 and Section 4.1.1.3. The runtime links statically against this library and exports its functionality to the enclaves through a set of WASMI host functions, which enables WTAs to create policy sessions and use their associated handles to perform IO on trusted storage objects.

5.2.1 Rust API

We provide a Rust API that enables the runtime and its modules to create EA policy sessions and use them as authorization tokens when accessing trusted storage objects.

5.2.1.0.1 Command The `Command` data type represents an EA assertion, as described in Section 4.1.1.2.

```
pub enum Command {
    /// Regular EA assertion
    Check(Check),
    /// EA policy disjunction assertion
    Or(Vec<OrBranch>),
}
```

Listing 5.5: `Command` data type

5.2.1.0.2 OrBranch The `OrBranch` data type represents an EA policy disjunction assertion, as described in Section 4.1.1.2.

```
pub enum OrBranch {
    SessionDigest(PolicyDigest),
    Session(AuthSession),
}
```

Listing 5.6: `OrBranch` data type

5.2.1.0.3 Check The `Check` data type represents a regular EA assertion.

```
pub enum Check {
    InstanceId,
    CodeSignatureCertificate,
    CodeSignatureCertificateChain {
        der: Vec<u8>,
    },
    CodeHash,
    ApplicationId,
    ObjectStorage {
        id: u32,
    },
    ObjectAccess {
        flags: u32,
    },
    ObjectSharing {
        flags: u32,
    },
    ObjectExtra {
        flags: u32,
    },
    IsMigrationAuthority,
}
```

Listing 5.7: `Check` data type

5.2.1.0.4 AuthSession The `AuthSession` data type maintains a SHA-256 hasher and the list of EA `Commands` that were supplied to the hasher.

```
pub struct AuthSession {
    /// Hasher for the commands
    hasher: Sha256,
    dirty: bool,
    /// List of commands run in this session
    commands: Vec<Command>,
}
```

Listing 5.8: `AuthSession` data type

An implementation on top of `AuthSession` is provided that enables the runtime to instantiate an EA policy session, add assertions by invoking `run_command`, and retrieve the final digest representation of the policy it incorporates.

```
impl AuthSession {
    pub fn new() -> Self;

    /// Runs a command in the context of this session
    /// with the supplied [‘CheckResolver‘]
    pub fn run_command<R: CheckResolver<Error = E>, E>(
        &mut self,
        command: Command,
        resolver: &R,
    ) -> Result<(), AuthError<E>>;

    /// Finalizes the session and returns its policy digest
    pub fn finalize(self) -> PolicyDigest;

    /// Returns the list of commands run in this session
    pub fn get_commands(&self) -> &[Command] {
        return &self.commands;
    }
}
```

Listing 5.9: `AuthSession` implementation

5.2.1.0.5 PersistentObject The `PersistentObject` data type contains information on an open trusted storage object. The PoC implementation does not currently store a copy of the internal `TEE_ObjectHandle` received from the OP-TEE internal core API. Instead, it keeps a regular file-system file that emulates a trusted storage object. Furthermore, cryptographic keypair objects are currently mocked with an ED25519 keypair that is randomly generated on object creation.

```
/// Persistent object types
pub enum PersistentObject {
    /// Cryptographic key object
    CryptoKey(CryptoKeyObject),
    /// Cryptographic keypair object
    CryptoKeypair(CryptoKeypairObject),
    /// Regular data object
    Data(DataObject),
}

/// Cryptographic key object
pub struct CryptoKeyObject {
    inner: PersistentObjectInner,
}

/// Cryptographic keypair object
pub struct CryptoKeypairObject {
    inner: PersistentObjectInner,
    /// Mocked keypair
    keypair: Ed25519KeyPair,
}

/// Regular data object
pub struct DataObject {
    inner: PersistentObjectInner,
}

/// Persistent object structure
pub struct PersistentObjectInner {
    /// Extended object information
    info_ex: PersistentObjectInfoEx,
    /// Object ID
    oid: ObjectId,
    /// Internal handle is replaced with a file
    file: Option<File>,
}

/// Persistent object information
pub struct PersistentObjectInfo {
    pub object_type: u32,
    pub object_size: u32,
    pub max_object_size: u32,
    pub object_usage: u32,
    pub data_size: u32,
    pub data_position: u32,
    pub handle_flags: u32,
}

/// Extended persistent object info
pub struct PersistentObjectInfoEx {
    /// Internal info
    info: Option<PersistentObjectInfo>,
    /// The EA session associated with this object
    auth_session: AuthSession,
}
```

Listing 5.10: `PersistentObject` data type

5.2.1.0.6 Storage The `Storage` Rust interface exposes methods for performing basic IO on trusted storage `PersistentObjects`. These objects can be regular data objects with an application-specific structure or cryptographic keys and keypairs. As anticipated in the previous paragraph, the PoC implementation of the `Storage` interface does not currently invoke the TEE internal core API for IO and instead mocks trusted storage objects with regular files.

```
impl Storage {
    /// Creates a persistent object given an auth-
    /// entication session, an object id, and
    /// associated flags and attributes.
    pub fn create_object(
        &mut self,
        session: AuthSession,
        obj_id: ObjectId,
        obj_type: PersistentObjectType,
        storage_id: u32,
        flags: u32,
        attributes: Option<&PersistentObject>,
    ) -> Result<PersistentObject, StorageError>;

    /// Opens a persistent object given an authentication
    /// session, an object id and flags.
    pub fn open_object(
        &mut self,
        session: AuthSession,
        obj_id: ObjectId,
        storage_id: u32,
        flags: u32,
    ) -> Result<PersistentObject, StorageError>;

    /// Closes a persistent object by taking ownership
    /// of it
    pub fn close_object(
        &mut self,
        obj: PersistentObject,
    ) -> Result<(), StorageError>;

    /// Closes a persistent object by taking ownership
    /// of it, then deletes it from the storage
    pub fn close_and_delete_object(
        &mut self,
        obj: PersistentObject,
    ) -> Result<(), StorageError>;

    /// Reads ‘size‘ bytes of data from the object
    /// from the current seek position
    pub fn read_object_data(
        &self,
        obj: PersistentObject,
        size: usize,
    ) -> Result<Vec<u8>, StorageError>;

    /// Writes ‘data‘ to the object at the current
    /// seek position
    pub fn write_object_data(
        &self,
        obj: PersistentObject,
        data: &[u8]
    ) -> Result<(), StorageError>;

    /// Moves the seek position for the object at
    /// ‘offset‘ relative to ‘whence‘
    pub fn seek_object_data(
        &self,
        obj: PersistentObject,
        offset: usize,
        whence: Whence,
    ) -> Result<(), StorageError>;
}
```

Listing 5.11: `Storage` interface

5.2.2 WebAssembly API

The runtime re-exports the Rust API in Section 5.2.1 to the enclaves through host functions, i.e., the runtime-enclave interface, that accept opaque handles to runtime-managed resources (e.g., EA authorization sessions and persistent objects). On top of these host functions, higher-level Rust bindings are also provided that abstract away the marshaling and unmarshaling of the arguments across the WASM application binary interface (ABI) boundary. We illustrate the WASM API for `AuthSession` in the following paragraphs.

5.2.2.0.1 create_session The `create_session` host function creates a fresh `AuthSession` and returns a `Handle` to it on success. A `StorageError` describes the reason why the operation failed.

```
fn __rt_create_session(
    // Pointer to the variable for the returned handle
    out_handle: *mut u32
) -> i32;
```

Listing 5.12: `create_session` host function signature

```
pub fn create_session() -> Result<Handle, StorageError>;
```

Listing 5.13: `create_session` Rust binding function signature

5.2.2.0.2 session_run_command The `session_run_command` host function takes an EA assertion packaged in a `Command` and the `Handle` of a session within which the runtime will run the supplied `Command`. A `StorageError` describes the reason why the operation failed. The `Command` parameter is marshaled into JSON using *serde* [57] before passing it to the host function.

```
fn __rt_session_run_command(
    // Handle to the trial or auth session
    session_handle: u32,
    // Pointer to the marshaled command buffer
    cmd: *const u8,
    // Length of the marshaled command buffer
    cmd_len: u32
) -> i32;
```

Listing 5.14: `session_run_command` host function signature

```
fn session_run_command(
    // Handle to the trial or auth session
    session: Handle,
    // Command to run within the supplied session
    cmd: Command
) -> Result<(), StorageError>;
```

Listing 5.15: `session_run_command` Rust binding function signature

5.3 Remote attestation

We have implemented the key attestation mechanism described in Section 4.1.2, which allows the runtime or an enclave to attest to the security of an Ed25519 keypair. This keypair can be used either to bind a trusted communication channel with another TEE or to sign QuickProv chains. We have split the remote attestation functionality into two separate Rust crates that the runtime depends on, one for cryptographic functionality (`libcrypto`) and one specifically for remote attestation primitives (`libra`).

The `libcrypto` crates provides cryptographic primitives for random number and Ed25119 keypair generation as well as the generation and verification of X.509 certificates. The `libra` crate depends on `libcrypto` and provides primitives for key attestation and attested signature. The latter allows for generating signatures with an Ed25519 keypair that is certified by a key attestation certificate.

5.3.1 Rust API

We provide a Rust API that enables the runtime and its enclaves to obtain key attestation certificates signed by the local CA. Besides certificate generation, the API exports an auxiliary method for computing attested Ed25519 signatures.

5.3.1.0.1 AttestationEvidence The `AttestationEvidence` data type represents a claim that can be bundled as evidence in the custom extensions field of an X.509 attestation certificate. The `Keypair` variant holds `PolicyEvidence` for a cryptographic keypair, while the `CodeHash` variant represents a hash measurement for an arbitrary binary, which can be the runtime or a WTA. The `AttestationEvidence` and `PolicyEvidence` come with `*Asn1` variants that can be serialized in the ASN.1 format, which is a requirement for X.509 custom extensions.

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum AttestationEvidence {
    /// PolicyEvidence for a cryptographic keypair
    Keypair(PolicyEvidence),
    /// Code hash measurement
    CodeHash(Vec<u8>),
}

#[derive(Clone, Debug, PartialEq, Eq, asn1::Asn1Read, asn1::Asn1Write)]
pub(crate) enum AttestationEvidenceAsn1<'a> {
    Keypair(PolicyEvidenceAsn1<'a>),
    CodeHash(&'a [u8]),
}

#[derive(Clone, Debug, PartialEq, Eq)]
pub struct PolicyEvidence {
    /// EA policy digest
    pub policy_digest: Vec<u8>,
    /// List of assertions in the policy
    pub policy_commands: Vec<u8>,
}
```

```
#[derive(Clone, Debug, PartialEq, Eq, asn1::Asn1Read, asn1::Asn1Write)]
pub(crate) struct PolicyEvidenceAsn1<'a> {
    pub policy_digest: &'a [u8],
    pub policy_commands: &'a [u8],
}
```

Listing 5.16: `AttestationEvidence` and `PolicyEvidence` data types

5.3.1.0.2 AttestedCrypto The `AttestedCrypto` data type maintains a trusted storage instance shared with the runtime that can be used to retrieve cryptographic keys.

```
/// Facade class that provides attested cryptographic
/// operations on top of a trusted [`Storage`] instance
pub struct AttestedCrypto {
    /// Shared trusted storage instance
    storage: Arc<Mutex<Storage>>,
}
```

Listing 5.17: `AuthSession` data type

5.3.1.0.3 AttestedSignature An `AttestedSignature` implementation on top of `AttestedCrypto` is provided that exposes methods for obtaining key attestation certificates and generating attested signatures. The implementation packages an attestation claim in the `AttestationEvidenceAsn1` data type described above, serializes it into ASN.1 format, and embeds them as a custom extension field in the X.509 key attestation certificate. Two unique object identifiers (OIDs) are used to discern the custom extension field type:

- `OID_KEYPAIR_POLICY_EVIDENCE`: The custom extension field contains a `PolicyEvidence` claim
- `OID_CODE_HASH_POLICY_EVIDENCE`: The custom extension field contains a code hash measurement

```
pub trait AttestedSignature {
    /// Signs data with the supplied keypair
    fn sign(&self, data: &[u8], keypair: &CryptoKeypairObject) -> Result<Vec<u8>>;

    /// Returns a PEM-formatted X.509 certificate containing
    /// attestation claims for the supplied keypair
    fn attest_key(
        &self,
        keypair: &CryptoKeypairObject,
        ca_cert: &SignerCertificate,
    ) -> Result<String>;
}
```

Listing 5.18: Implementation of the `AttestedSignature` trait for `AttestedCrypto`

5.4 QuickProv

We have developed a PoC implementation of the provenance collection and storage layers of QuickProv system presented in Section 4.2 as a Rust crate (*wasm-enclave-prov*) that the runtime depends on. The PoC implements provenance chains on top of the tamper-evident, append-only logs provided by the Rust implementation [18] of the Hypercore [30, 48] protocol. Currently, the PoC only supports logging enclave migrations and IO operations on trusted storage objects, although additional records can be added easily. The generation and verification of the provenance attestation certificate described in Section 4.2.4 has not been implemented. However, adding support for this would simply involve reusing the attestation mechanism for key attestation mechanism implemented in Section 5.3.

5.4.1 Rust API

We provide a Rust API that the runtime can leverage to access QuickProv functionality and perform the following operations:

- Create a provenance chain with an associated keypair
- Log provenance records on the chain
- Retrieve provenance records from the chain
- Store the chain and the keypair on trusted storage
- Finalize the chain and, optionally, authorize a next writer identified by their public key

In the next few paragraphs, we describe the implementation of the record, log and chain structures presented in Section 4.2.

5.4.1.0.1 Record The `Record` data type represents a provenance record that describes two possible types of event: the migration of an enclave (`MigrationRecord`) and an IO operation (`IoRecord`).

```
/// Base type for provenance records
#[derive(Clone, Serialize, Deserialize)]
pub enum Record {
    /// Enclave migration record
    Migration(MigrationRecord),
    /// Trusted storage IO record
    Io(IoRecord),
}

/// Enclave migration record
#[derive(Clone, Serialize, Deserialize)]
pub struct MigrationRecord {
    /// Status of the migration
    status: MigrationStatus,
    /// Base64-encoded public key of the source TEE worker
    src_pk: String,
    /// Base64-encoded public key of the destination TEE worker
    dst_pk: String,
}

/// Trusted storage IO record
#[derive(Clone, Serialize, Deserialize)]
pub struct IoRecord {
    /// Identity of the enclave that initiated the IO operation
    src_enclave_id: Uuid,
    /// Base64-encoded SHA-256 hash of the object before the IO operation
    obj_hash: String,
    /// The IO operation
    io_op: IoOperation,
}

/// Trusted storage IO operation
#[derive(Clone, Serialize, Deserialize)]
pub enum IoOperation {
    /// Write operation
    Write(IoOperationWrite),
}

/// Trusted storage IO write operation
#[derive(Clone, Serialize, Deserialize)]
pub struct IoOperationWrite {
    /// Base64-encoded SHA-256 hash of the written payload
    write_hash: String,
}
```

```
/// Enclave migration status
#[derive(Clone, Copy, Serialize, Deserialize)]
pub enum MigrationStatus {
    /// Migration is started
    Start,
    /// Migration is aborted
    Abort,
    /// Migration is completed
    Complete,
}

/// Records are serializable to JSON
impl JsonTrait for Record {}
```

Listing 5.19: `Record` data type and variants

5.4.1.0.2 Feed The *hypercore* Rust crate provides a `Feed` data type that implements a tamper-evident, append-only log. The log is internally represented as a signed Merkle tree but it acts an append-only list by providing operations for appending and retrieving data. The library provides a `RandomAccess` trait that allows for persisting logs on any storage medium that supports random access IO. Besides the `Feed` data (i.e., the records), Hypercore persists the following auxiliary structures:

- The merkle tree that encodes the data contained in the `Feed`. The leaf nodes of this tree contain, from left to right, the hashes of the respective records in the `Feed`. This structures enables tamper detection if the contents or the order of the records is altered
- A bitfield that efficiently represents the merkle tree and the data it describes
- The signatures of the `Feed` data
- The keypair that signs the data

5.4.1.0.3 MultiFeed We provide a `MultiFeed` structure based on `Feed` that implements a chain as described in Section 4.2.1.3. A `MultiFeed` is, in essence, an ordered sequence of `Feed`s persisted as separate objects on the trusted storage:

```
#[derive(Debug)]
pub struct MultiFeed<T>
where
    T: RandomAccess<Error = Box<dyn std::error::Error + Send + Sync>>
```

```
        + std::fmt::Debug + Send,
{
    /// Last entry is EOF?
    finalized: bool,
    /// Sequence of feeds
    feeds: Vec<Feed<T>>,
}
```

Listing 5.20: `MultiFeed` data type

`Feed`s, in turn, consists of an ordered sequence of entries, which can be regular data entries (i.e., provenance `Record`s), or BOF/EOF markers:

```
/// An entry in the [‘MultiFeed‘], which can be either a
/// [‘MultiFeedMarker‘] or user-defined data
#[derive(Serialize, Deserialize)]
pub enum MultiFeedEntry {
    Marker(MultiFeedMarker),
    Data(Vec<u8>),
}

/// [‘Feed‘] marker
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum MultiFeedMarker {
    /// Beginning of multi-feed
    BOF {
        /// Pointer to the prev feed in the chain
        prev_feed: Option<MultiFeedLink>,
    },
    /// End of multi-feed
    EOF {
        /// Pointer to the next feed in the chain
        next_feed: Option<MultiFeedLink>,
    },
}

/// A pointer to the previous or next [‘Feed‘] in the
/// [‘MultiFeed‘] chain. The direction of the pointer is
/// specified by the enclosing [‘MultiFeedMarker‘]
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct MultiFeedLink {
    /// ID of the prev/next feed in the chain
    feed_id: String,
    /// Base64-encoded public key of the prev/next
    /// authorized writer in the chain
    auth_pk: String,
}
```

Listing 5.21: `MultiFeedEntry` data type and variants

Finally, an implementation on top of `MultiFeed` is provided that allows for appending and retrieving records from the chain, as well as finalizing the chain and, optionally, authorizing a next writer.

```
impl<T> MultiFeed<T>
where
    T: RandomAccess<Error = Box<dyn std::error::Error + Send + Sync>>
        + std::fmt::Debug + Send,
{
    /// Returns the size of the multi-feed, i.e., the number of feeds
    pub fn size(&self) -> usize;

    /// Returns the total number of records in the multi-feed chain
    pub fn len(&self) -> u64;

    /// Gets the i-th element in the multi-feed
    pub fn get<'a>(
        &'a mut self,
        index: u64,
    ) -> impl Future<Output = anyhow::Result<Option<Vec<u8>>>> + 'a;

    /// Gets a reference to i-th feed in the multi-feed chain
    pub fn get_feed(&self, index: usize)
        -> Option<&Feed<T>>;

    /// Gets a reference to the last feed in the multi-feed chain
    pub fn last_feed(&self)
        -> Option<&Feed<T>>;

    /// Gets a mutable reference to i-th feed in the multi-feed chain
    fn get_feed_mut(&mut self, index: usize)
        -> Option<&mut Feed<T>>;

    /// Gets a mutable reference to the last feed in the multi-feed chain
    fn last_feed_mut(&mut self)
        -> Option<&mut Feed<T>>;

    /// Appends data to the multi-feed
    pub fn append<'a>(
        &'a mut self,
        data: &'a [u8],
    ) -> impl Future<Output = anyhow::Result<()>> + 'a;

    /// Appends a typed record to the multi-feed
    pub fn append_record<'a, R>(
        &'a mut self,
        data: &'a R,
    ) -> impl Future<Output = anyhow::Result<()>> + 'a
    where
```

```
        R: AnyhowFromStr + AnyhowToString;

    /// Finalizes the last feed in the chain by adding an EOF
    /// [‘MultiFeedMarker‘]
    pub fn finalize<’a>(&’a mut self)
        -> impl Future<Output = anyhow::Result<()>> + ’a;

    /// Finalizes the last feed in the chain by adding an EOF
    /// and authorizes a new writer identified by ‘auth_pk‘
    /// to write a feed identified by ‘feed_id‘
    pub fn finalize_and_auth<’a>(
        &’a mut self,
        feed_id: &str,
        auth_pk: &PublicKey,
    ) -> impl Future<Output = anyhow::Result<()>> + ’a;
}
```

Listing 5.22: Concise view of the `MultiFeed` implementation

5.4.2 Provenance collection

We perform provenance collection by hooking the host functions for the trusted storage Rust API and the `migrate_app` enclave migration function described in Section 5.4.1. The hooks retrieve the enclave-specific provenance chain from the `App` structure and log either an `IoRecord` or a `MigrationRecord`.

For instance, logging an enclave migration event is done as follows:

```
pub(crate) async fn migrate_app<A: std::net::ToSocketAddrs>(
    &self,
    to: A,
    instance: ModuleInstanceRef,
) -> anyhow::Result<()> {
    // Log the migration
    {
        // Lock the runtime context for writing
        let mut context = self.context.write();
        // Get the public key of the local platform
        let src_pk = base64::encode("Public key of local platform CA here");
        // Get the public key of the remote platform
        let dst_pk = base64::encode("Public key of remote platform CA here");
        // Get the provenance chain for the current enclave
        let prov_feed = context.current_app_mut().prov_feed_mut();

        // Log the migration
        prov_feed
            .append_record(&Record::Migration(MigrationRecord::new(
                MigrationStatus::Start,
                &src_pk,
                &dst_pk,
            )))
            .await?;
    }

    // ...Do migration
}
```

Listing 5.23: Provenance hook for enclave migration in `migrate_app`

On the other hand, we log IO write operations on trusted storage objects as follows:

```
pub(super) fn write_object_data(
    &self,
    handle: Handle,
    data: &[u8],
) -> Result<(), StorageError> {
    // Lock the runtime context for writing
    let mut runtime_ctx = self.runtime_ctx.write();
    // Get the enclave identity
    let instance_id = runtime_ctx.current_app().instance_id();
    // Get the provenance chain for the current enclave
    let prov_feed = runtime_ctx.current_app_mut().prov_feed_mut();

    // Log the IO operation
    prov_feed
        .append_record(&Record::Io(IoRecord::new(
            instance_id,
            base64::encode("hash of object referenced by ‘handle‘ here"),
            IoOperation::Write(
                IoOperationWrite::new(
                    base64::encode("hash of ‘data‘ here")
            )),
        )))
        .await
        .map_err(|_| StorageError::Error)?;

    // ...Do object write
}
```

Listing 5.24: Provenance hook for IO object write in `write_object_data`

5.4.3 Provenance storage

We provide a `RandomAccessSse` implementation of the `RandomAccess` trait that allows for persisting provenance chains on trusted storage. This implementation stores the chain data—and the auxiliary structures described in Paragraph 5.4.1.0.2—as separate objects protected by EA policies. The trusted storage functionality of OP-TEE guarantees the confidentiality and integrity of these objects by encrypting data blocks with AES-GCM and a FEK that is secured in memory and encrypted with an associated TSK (see Figure 5.2). By specifying an EA policy that includes the RPMB storage type, OP-TEE will store the chain on disk and a metadata file on a RPMB partition that describes its latest state after every update. Thanks to the metadata, these structures can be deleted or replayed but any such alterations are detectable.

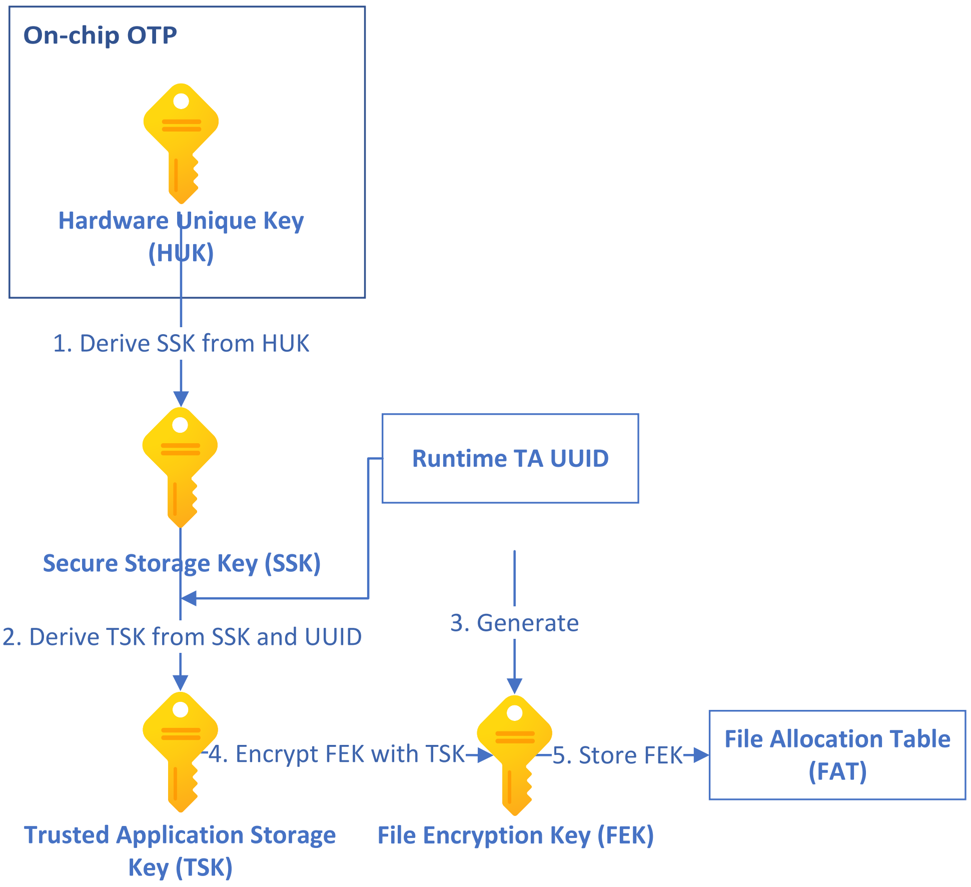
On-chip OTP
Hardware Unique Key (HUK)
1. Derive SSK from HUK
Secure Storage Key (SSK)
Runtime TA UUID
2. Derive TSK from SSK and UUID
3. Generate
4. Encrypt FEK with TSK
5. Store FEK
File Allocation Table (FAT)
Trusted Application Storage Key (TSK)
File Encryption Key (FEK)

Figure 5.2: The FEK derivation process. A 128-bit secure storage key (SSK) is derived from the HUK, i.e., the ROT for storage, during OP-TEE initialization. This is kept in secure memory and never persisted to disk. A TA storage key (TSK) is derived from this key and the runtime TA UUID. For each trusted storage object, a random file encryption key (FEK) is generated, encrypted with the TSK, and stored in the object's file allocation table FAT entry [49]

The runtime creates a provenance chain and specifies its EA access control policy for it when an enclave is first initialized. The process can be summarized as follows:

1. Generate a random enclave-specific Ed25519 keypair
2. Derive an enclave-specific identity from the keypair
3. Create an EA policy session for the enclave-specific provenance chain and specify the following checks:
    - `IsMigrationAuthority`: The data is only accessibly by the migration authority, i.e., the runtime
    - `ObjectStorage = STORAGE_PRIVATE | STORAGE_RPMB`: The data is stored on disk with guaranteed confidentiality and integrity. Metadata is stored on RPMB that guarantees deletion and replay detection.
    - `ObjectAccess = READ | WRITE | WRITE_META`: The data can be opened with full access rights
4. Instantiate the enclave-specific provenance chain by building a `MultiFeed` and binding it to the keypair generated at step (1).

By default, the runtime does not wait for written records to be flushed to disk before returning. To change this behavior and ensure that records are definitively flushed to disk before signaling success, the `auto_sync` option must be enabled for the chain's backing `RandomAccessSse` instance. While automatic synchronization, coupled with OP-TEE's guarantee of atomic writes, ensures that the chain on trusted storage is always up-to-date and correct, flushing the chain data on every write may be expensive depending on the logging frequency. Therefore, in the PoC we only manually flush the chain to disk in case of enclave migration.

Chapter 6

Evaluation

We evaluate our PoC implementation of QUICKPROV with respect to the security requirements and performance goals formulated in Section 3.4. The QUICKPROV system must enable readily-usable, trustworthy data provenance for mobile enclaves in heterogeneous distributed systems. For the provenance data to be readily usable, the time it takes for it to be ready after a migration is first initiated must be relatively minimal with respect to the total migration time. In order for the provenance data to be trustworthy, it must be secured from unauthorized access and there must exist a means for convincing other parties of its reliability.

6.1 Security analysis

In this section we will analyze the security requirements articulated in Section 3.4.1 with respect to the PoC and the local adversary model formulated in Section 3.3. As mentioned in Section 5.4, the PoC only implements provenance collection and storage functionality. Therefore, we will discuss the attestability aspect (**PS-6**) in Chapter 7.

6.1.1 Confidentiality (PS-1)

The QUICKPROV system must guarantee that provenance data be unreadable by unauthorized entities. Confidentiality must be guaranteed during both provenance collection and storage, and from both REE and TEE adversaries.

The runtime protects the collection process from REE adversaries by leveraging OP-TEE's TrustZone isolation, which prevents the CPU, untrusted DMA masters and peripherals from probing TEE memory (see Section 5.1.1).

The runtime stores the collected provenance data using OP-TEE's trusted storage functionality, which guarantees its confidentiality and integrity through AES-GCM authenticated encryption.

The runtime shields the collection process from TEE adversaries by taking advantage of *wasmi*'s sandboxing capability (see Section 5.1.1). It also prevents enclaves from accessing the provenance data by storing in its own private trusted storage namespace, i.e., by specifying an EA policy that includes the `IsMigrationAuthority` check.

6.1.2 Integrity (PS-2)

The QUICKPROV system must guarantee tamper detection for the provenance data in case any provenance records are corrupted or truncated by, e.g., replaying a proper prefix of a provenance chain.

Besides OP-TEE's guarantee of integrity for trusted storage, corruption detection is provided at the application level by Merkle trees and cryptographic signatures (see Paragraph 5.4.1.0.2) that back each provenance chain and enable detection of provenance forgery, re-ordering and corruption.

To provide replay detection, we persist the provenance chains and their auxiliary structures with an EA policy that includes the `ObjectStorage` = `STORAGE_PRIVATE` | `STORAGE_RPMB` check. This configuration mandates that OP-TEE store the provenance data on REE storage and a special `dirfile.db.hash` file on RPMB that holds a hash representing the latest state of the REE storage. This effetively enables the detection of deletion and replay attacks by REE adversaries.

6.1.3 Unforgeability (PS-3)

The QUICKPROV system must prevent unauthorized parties from forging valid provenance records by altering existing entries or adding new entries. Forging detection is guaranteed at the application level by cryptographic signatures of the provenance chains. Since REE and TEE adversaries cannot access the keypair that the runtime uses to sign the provenance data, they cannot forge phony provenance records without being detected.

6.1.4 Non-repudiation (PS-4)

The QUICKPROV system must prevent authorized parties, i.e., runtimes, from adding provenance records and later denying their authorship. Non-repudiation is guaranteed by cryptographically signing each provenance chain with a unique keypair that only the runtime can access. While the public component of this keypair identifies the chain, the public key must also be cryptographically bound to the runtime for it not to be able to deny its involvement. This is achieved with the provenance attestation mechanism described in Section 4.2.4. However, this has not been implemented by the PoC.

6.1.5 Chronology (PS-5)

The QUICKPROV system must guarantee that the provenance records be collected and stored in a chronological order. The runtime runs in a single thread, thus ensuring mutual exclusion for IO and enclave migration. Since there cannot be interleaving, the runtime guarantees that provenance records are collected in the correct order. Furthermore, provenance chains are append-only, meaning that the collected records are always stored at the end, which preserves the chronological order on storage as well.

6.2 Performance analysis

In this section we will analyze the performance goals formulated in Section 3.4.2. We have evaluated the performance of QUICKPROV on a machine equipped with a 6-core Intel® Core™ i7-10750H CPU and 16 GB of RAM. Due to complications[1] in getting the enclave platform with the QUICKPROV PoC to run correctly on OP-TEE with QEMU, we have run our performance experiments on Linux. This implies that provenance data is stored unprotected on regular files on the REE filesystem. For the test, we have developed a minimal WTA that performs various IO operations and triggers a migration[2]. The runtime loads and runs this WTA in a *wasmi* enclave, and logs a provenance record for each of these operations. The runtime logs a total of 4096 provenance records for the entire execution of the WTA: 4095 `IoRecords` and 1 `MigrationRecord`.

```
#[no_mangle]
pub extern "C" fn test_prov() -> i32 {
    // Trigger 4095 IoRecords
    for i in 0..4095 {
```

[1]The complications stem from QEMU limiting the amount of TrustZone secure memory (i.e., `TZDRAM_SIZE`) to 15 MB.

[2]The migration code was developed by a colleague working on a separate sub-project.

```
        test_create_close_and_delete_object();
    }

    // Trigger a MigrationRecord
    trigger_migration();

    ERRNO_OK
}
```

Listing 6.1: The `test_prov` function of the test WTA

6.3 Minimal intrusiveness (PP-1)

The QUICKPROV collection process should incur minimal performance overhead. We have measured the time it takes to collect and log a provenance record with respect to the execution time of the `create_object` function. The execution time of `create_object` is, on average, approximately $450\mu s$. On the other hand, the time it takes to log a record is, on average, $230.18\mu s$ (± 91.5). Since the log is based on a Merkle hash tree, we expect the logging time to grow as quickly as $\mathcal{O}(\log n)$, where n is the total number of records in the chain, since every append operating requires updating all the intermediate hash nodes from the added leaf up to the root node.

6.4 Minimal storage overhead (PP-2)

The collected provenance data should not require a considerable amount of storage space. The test WTA produces, in total, 2177 KB of provenance data, meaning approximately 544.25 bytes per record. We attribute the considerable record size to the the data serialization code of the QUICKPROV PoC. We argue that, by adopting a binary encoding of the records instead of JSON, we can achieve a total provenance data size of approximately 320 KB for the same workload.

6.5 Responsiveness (PP-3)

Finally, migrating the provenance data to another TEE worker should not significantly affect the total migration time of an enclave. We have measured the migration time of an enclave that hosts the test WTA. This includes creating a snapshot of the enclave, transferring it to the destination worker node, and re-loading the enclave and its provenance data from the snapshot. The results are shown below:

Migration step	**With QuickProv** (ms)	**Without QuickProv** (ms)
Snapshot creation	2.25 (±0.20)	1.43 (±0.09)
Snapshot transfer	467.51 (±12.16)	362.95 (±8.32)
Snapshot restore	99.55 (±5.22)	93.54 (±3.48)

Table 6.1: Enclave migration time with and without QuickProv. The average and standard deviation values are computed over a sample of six different workload runs.

As expected, transferring the 2177 KB of provenance data requires, approximately, an extra 100 ms. On the other hand, creating and restoring a snapshot of the enclave does not seem to be noticeably affected by QuickProv. This is also expected, since creating and restoring a snapshot of an enclave only requires copying its memory and writing its provenance data on disk.

Chapter 7

Discussion

The objective of QUICKPROV is to enable fast, trustworthy data provenance for mobile enclaves. As enclaves migrate through many systems, systems can check the provenance data to check their history and decide whether to trust them or not. The PoC we developed addresses all the security requirements of QUICKPROV except attestability (**PS-6**). QUICKPROV needs to be attestable in order to convince other systems that provenance data is actually trustworthy. Therefore, part of future work consists in implementing the provenance attestation scheme described in Section 4.2.4. This scheme will also need to include a *challenge* in the attestation certificates to provide freshness, which will ensure that cached certificates cannot be replayed.

Another security aspect that is relevant to QUICKPROV but has not been covered in detail in this work is that of key erasure, that is whether or not a keypair must be erased once the chain it signs is migrated. Every TEE generates a new keypair when an enclave is first initialized or when it resumed after a migration. This keypair is then used to sign a new log in the chain. Erasing a keypair once a chain is migrated ensures that, if the owning TEE is ever to be compromised, it will be unable to reuse the old key to produce valid provenance records or overwrite history from a chain it signed in the past. This prevents the TEE from pretending that a log starts from an older point in the chain than it actually does. Attesting that a key was indeed deleted from the TEE requires a proof of secure erasure (PoSE) [34]. We argue that we can leverage the TEE security guarantees to provide this proof as a claim that can be included in attestation certificates. This would allow a TEE to perform key erasure that can be verified by remote parties, as proposed by Hao *et al.* [27].

One of the performance goals for QUICKPROV is for it to be minimally intrusive (**PP-1**). The results from our experiments reveal that provenance logging accounts for about 33% of an IO operation, and, theoretically, the logging time will grow logarithmically with respect to the length of the provenance chain. However, the experiments did not take into account the overhead of replay protection, which generally requires incrementing a monotonic counter, either directly or indirectly (e.g., RPMB).

Matetic *et al.* [44] show that incrementing monotonic counters in non-volatile memory is generally slow (80–250 ms).

We also argue that we can achieve a much lower storage overhead (**PP-3**) by adopting a more optimized representation for the provenance records (e.g., binary) and compressing the chains and their auxiliary structures.

Experiment results on QUICKPROV's responsiveness (**PP-3**) show that each KB of provenance data adds, on average, a delay of approximately $48\mu s$ to the migration process. We argue that we can achieve significantly lower figures by optimizing QUICKPROV's storage overhead, since provenance transport accounts for about 94% of the migration latency. The reported times, however, do not account for provenance verification, that is traversing an enclave's provenance chain (e.g., to inspect its migration history). Since the ownership information is encoded in the BOF and EOF markers of each log in the chain, we believe that verifying an enclave's history will hardly have any impact on the measured times.

Chapter 8

Related work

The first work on automatic secure provenance collection and storage relied on user processes or the OS kernel as the “root of trust“ for the collection and storage of provenance data. In 2006, Muniswamy-Reddy *et al.* introduced a provenance-aware storage system known as PASS [46]. PASS implemented transparent provenance collection and storage in the kernel as a module and a virtual file system (VFS) called PASTA, respectively. PASTA was a provenance-aware VFS that, unlike provenance systems utilizing a separate user-space database (e.g., LinFS [55]), provided ”greater synchronicity” between data and its provenance by caching the provenance in an in-kernel Berkeley DB [50]. In 2008, Simmhan et al. proposed Karma2 [58], a provenance framework for data-driven workflows that, unlike PASS, implemented provenance collection at the workflow and process level [67]. Moreover, instead of Berkeley DB, Karma2 was based on an XML database [67]. To address the deployment issues caused by the different storage models adopted by each provenance system (e.g., Berkeley DB, XML, etc.), Hasan et al. introduced a ”highly-configurable, platform-independent library for secure provenance” called SPROV2 [29].

The growing interest and availability of TEEs to the public, in conjuction with the advent of blockchains, has ushered in a new era for data provenance. TEEs provide a trusted and strongly isolated environment for the execution and storage of security and privacy critical code and data, which perfectly serves the requirements of secure and trustworthy provenance. Blockchains can ensure the availability and tamper-evidence of provenance by providing a witness that auditors can query to verify its integrity. Taha et al. [8] were the first to leverage the TPM to achieve trustworthy provenance collection, although this was still integrated at the OS level. Liang et al. were the first to securely store provenance metadata on a blockchain with their ProvChain [39] system. However, Kaaniche et al. were the first to secure both provenance collection and storage with Prov-Trust [33]. They achieved this by running the collection module inside an SGX enclave and storing the provenance metadata on a Hyperledger blockchain [5]. Hyperprov [64] is another provenance framework that uses Hyperledger Fabric to securely store the metadata, while storing the actual provenance data on a pluggable off-chain, storage medium.

Since blockchains can have several writers, they have to rely on consensus algorithms to reach unanimous agreement on the state of the ledger. Depending on the algorithm, it may take several seconds, if not minutes, to commit a transaction on a blockchain. Permissioned blockchains guarantee lower transaction finality times by restricting the blockchain writers to a select few authorized ones, which allows for adopting more efficient, deterministic consensus mechanisms like PBFT [12]. When we only have one writer per blockchain, there is no need for consensus at all. This observation is exactly what underpins projects like Hypercore [30], Secure Scuttlebutt (SSB) [56] and Chronicle [51], which rely on single-writer, append-only logs that, in the case of Hypercore and SSB, can be replicated by other peers over a network. Although these logs can indeed be replicated, Hypercore and SSB do not specify an incentive for replication, which is what derived projects like DatDot [17] attempt to solve by bridging the log replication protocol with a blockchain's built-in incentive model. These logs, which are part of a wider class referred to as verifiable data structures [2], are a fundamental building block of Google's Trillian [24] project, which powers certificate transparency [38], one of the most widely used production grade ledger-based ecosystems. Certificate transparency constitutes an exemplary instance of another problem that can be solved without blockchains. Certificate transparency and, more in general, verifiable data structures, provide several proofs that auditors can periodically verify to ascertain their immutability (i.e., integrity and append-only) and correct operation. However, there is no guarantee that an append-only log cannot be truncated, i.e., rolled back to a proper prefix, in between two audit operations, since there is no permanent secure communication channel between the auditors and whomever maintains the log [65]. This problem, referred to as freshness, is all the more critical if the log is maintained and operated in an untrusted environment, where attackers may want to truncate information to conceal their malicious activity or just to disrupt correct operation.

Strong rollback detection requires either that information be stored on a replay-protected medium (e.g., RPMBs) or that its state be anchored to a secure hardware monotonic counter. When the freshness of data cannot be guaranteed by hardware, trusted monotonic counter services such as ROTE [44] and ADAM-CS [43] exist that allow for allocating a virtual monotonic counter on a remote TEE. ROTE securely stores the counters in SGX enclaves and replicates them with a flexible gossiping protocol that can withstand up to 1 Byzantine enclave for every 2 processors (each enclave is run on a separate processor), instead of the usual $3f + 1$ constraint of standard Byzantine consensus protocols [44]. ADAM-CS, on the other hand, relies on TPMs to multiplex hardware counters into multiple virtual counters protected against data loss and rollback attacks, and only uses replication to increase performance [23].

Chapter 9

Conclusion

In this work, we have designed QUICKPROV, a framework for fast, trustworthy data provenance for enclaves in heterogeneous distributed systems. Instead of relying on slow and expensive consensus algorithms to achieve trustworthiness, we leveraged tamper-evident logs and the strong security and isolation guarantees of TEEs to guarantee the confidentiality, integrity and authenticity of provenance data during its collection and storage phases. We also designed a remote attestation mechanism that TEEs can utilize to convince remote parties that the provenance data is trustworthy. This can be used when migrating enclaves and their provenance data between TEEs. Hence, we developed a PoC implementation of the QUICKPROV provenance collection and storage modules and an enclave platform for WebAssembly applications on top of which QUICKPROV can track enclave migrations and IO operations. We showed that the PoC fulfills all the security requirements except attestability, which is left as future implementation work. Finally, we argued that further optimizations can be implemented to drastically reduce storage overhead of the PoC, thereby also achieving an overall higher responsiveness.

To conclude, we envision two interesting directions for future work. The first direction is concerned with researching how TEE capabilities can be leveraged to prove the secure erasure of signing keys, which is what we earlier referred to as PoSE. This can be used by a proving TEE, for instance, to convince a relying TEE that a provenance signing key has been completely erased from TEE memory and secure storage. This ensures that the key cannot leak and be reused to forge provenance data. The second direction consists in researching optimized data structures for provenance data that will enable more efficient querying and analysis, which is extremely important for large provenance chains. This will enable provenance auditors, such as a client TEE, to more efficiently retrieve records from a chain that match a specific query, instead of traversing the entire chain to filter the records they are interested in.

Bibliography

[1] Rustls - crates.io: Rust Package Registry. `https://crates.io/crates/rustls`.

[2] Adam Eijdenberg, B. L., and Cutter, A. Verifiable Data Structures, 2015. `https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf`.

[3] Alam, M., Ali, T., Khan, S., Ali, M., Nauman, M., Hayat, A., Khan, K., and Alghathbar, K. Analysis of existing remote attestation techniques. *Security and Communication Networks 5* (09 2012).

[4] Alqahtani, S., and Demirbas, M. Bottlenecks in Blockchain Consensus Protocols. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)* (aug 2021), IEEE.

[5] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., and Yellick, J. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[6] Apache Foundation. Teaclave TrustZone SDK. `https://github.com/apache/incubator-teaclave-trustzone-sdk`.

[7] ARM Limited. Building a Secure System using TrustZone®Technology. `https://www.arm.com/technologies/trustzone-for-cortex-a/tee-reference-documentation`.

[8] BANY TAHA, M. M., CHAISIRI, S., AND KO, R. K. L. Trusted tamper-evident data provenance. In *2015 IEEE Trustcom/BigDataSE/ISPA* (2015), vol. 1, pp. 646–653.

[9] BUCHMAN, E., KWON, J., AND MILOSEVIC, Z. The latest gossip on BFT consensus, 2018.

[10] BUKASA, S. K., LASHERMES, R., BOUDER, H. L., LANET, J.-L., AND LEGAY, A. How TrustZone Could Be Bypassed: Side-Channel Attacks on a Modern System-on-Chip. In *WISTP* (2017).

[11] BYTECODE ALLIANCE. wasmtime - A standalone runtime for WebAssembly. `https://github.com/bytecodealliance/wasmtime`.

[12] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (USA, 1999), OSDI '99, USENIX Association, p. 173–186.

[13] CERDEIRA, D., SANTOS, N., FONSECA, P., AND PINTO, S. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 1416–1432.

[14] CHAN, B. Y., AND SHI, E. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (New York, NY, USA, 2020), AFT '20, Association for Computing Machinery, p. 1–11.

[15] COKER, G., GUTTMAN, J., LOSCOCCO, P., HERZOG, A., MILLEN, J., O'HANLON, B., RAMSDELL, J., SEGALL, A., SHEEHY, J., AND SNIFFEN, B. Principles of remote attestation. *Int. J. Inf. Secur. 10*, 2 (jun 2011), 63–81.

[16] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086, 2016. `https://eprint.iacr.org/2016/086`.

[17] DATDOT. DatDot. `https://datdot.org/`.

[18] DATRS. Hypercore. `https://github.com/datrs/hypercore`.

[19] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (1983), 198–208.

[20] DOUCEUR, J. R. The Sybil Attack. In *Peer-to-Peer Systems* (Berlin, Heidelberg, 2002), P. Druschel, F. Kaashoek, and A. Rowstron, Eds., Springer Berlin Heidelberg, pp. 251–260.

[21] GLOBALPLATFORM, INC. TEE Protection Profile Version 1.3, 2020.

[22] GLOBALPLATFORM, INC. TEE Internal Core API Specification v1.3.1, Jul 2021. `https://globalplatform.org/specs-library/tee-internal-core-api-specification/`.

[23] GOLTZSCHE, D., NIEKE, M., KNAUTH, T., AND KAPITZA, R. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference* (New York, NY, USA, 2019), Middleware '19, Association for Computing Machinery, p. 123–135.

[24] GOOGLE. Trillian. `https://transparency.dev/`.

[25] GU, J., HUA, Z., XIA, Y., CHEN, H., ZANG, B., GUAN, H., AND LI, J. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017), pp. 225–236.

[26] GUNN, L. J., ASOKAN, N., EKBERG, J.-E., LILJESTRAND, H., NAYANI, V., AND NYMAN, T. Hardware Platform Security for Mobile Devices. *Foundations and Trends in Privacy and Security 3*, 3-4 (2022), 214–394.

[27] HAO, F., CLARKE, D., AND ZORZO, A. F. Deleting Secret Data with Public Verifiability. *IEEE Transactions on Dependable and Secure Computing 13*, 6 (2016), 617–629.

[28] HARDIN, T., AND KOTZ, D. Amanuensis: Information provenance for health-data systems. *Information Processing & Management 58*, 2 (2021), 102460.

[29] HASAN, R., SION, R., AND WINSLETT, M. SPROV 2.0: A Highly-Configurable Platform-Independent Library for Secure Provenance.

[30] HYPERCORE. Hypercore Protocol. `https://hypercore-protocol.org/`.

[31] IMRAN, A., AND AGRAWAL, R. *Data Provenance.* 01 2017.

[32] INTERNATIONAL TELECOMMUNICATIONS UNION. ITU-T Recommendation X.509. Tech. rep., International Telecommunication Union, 2019.

[33] KAANICHE, N., BELGUITH, S., LAURENT, M., GEHANI, A., AND RUSSELLO, G. Prov-Trust : towards a trustworthy SGX-based data provenance system. In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications - Volume 3: SECRYPT* (July 2020), P. Samarati, S. D. C. di Vimercati, M. Obaidat, and J. Ben-Othman, Eds., SCITEPRESS, pp. 225–237.

[34] KARVELAS, N., AND KIAYIAS, A. Efficient proofs of secure erasure. pp. 520–537.

[35] KERMARREC, A.-M., LAVOIE, E., AND TSCHUDIN, C. Gossiping with Append-Only Logs in Secure-Scuttlebutt. In *Proceedings of the 1st International Workshop on Distributed Infrastructure for Common Good* (New York, NY, USA, 2020), DICG'20, Association for Computing Machinery, p. 19–24.

[36] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 1–19.

[37] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. pp. 75–88.

[38] LAURIE, B. Certificate Transparency. *Queue 12* (08 2014), 10–19.

[39] LIANG, X., SHETTY, S., TOSH, D., KAMHOUA, C., KWIAT, K., AND NJILLA, L. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2017), IEEE Press, pp. 468–477.

[40] LINARO. OP-TEE (Open Portable Trusted Execution Environment). `https://www.op-tee.org/`.

[41] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory

from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium* (USA, 2018), SEC'18, USENIX Association, p. 973–990.

[42] MAO, Y., ZHANG, J., AND LETAIEF, K. B. Dynamic Computation Offloading for Mobile-Edge Computing With Energy Harvesting Devices. *IEEE Journal on Selected Areas in Communications 34*, 12 (2016), 3590–3605.

[43] MARTIN, A., LIAN, C., GREGOR, F., KRAHN, R., SCHIAVONI, V., FELBER, P., AND FETZER, C. ADAM-CS: Advanced Asynchronous Monotonic Counter Service. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2021), pp. 426–437.

[44] MATETIC, S., AHMED, M., KOSTIAINEN, K., DHAR, A., SOMMER, D., GERVAIS, A., JUELS, A., AND CAPKUN, S. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, Aug. 2017), USENIX Association, pp. 1289–1306.

[45] MOREAU, L., FREIRE, J., FUTRELLE, J., MCGRATH, R., MYERS, J., AND PAULSON, P. The open provenance model: An overview. vol. 5272, pp. 323–326.

[46] MUNISWAMY-REDDY, K.-K., HOLLAND, D., BRAUN, U., AND SELTZER, M. Provenance-Aware Storage Systems. pp. 43–56.

[47] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2009. `http://www.bitcoin.org/bitcoin.pdf`.

[48] OGDEN, M. Dat - Distributed Dataset Synchronization And Versioning, 01 2017.

[49] OP-TEE. OP-TEE Secure Storage. `https://optee.readthedocs.io/en/latest/architecture/secure_storage.html`.

[50] ORACLE. Berkeley DB. `https://www.oracle.com/database/technologies/related/berkeleydb.html`.

[51] PARAGON INITIATIVE ENTERPRISES. Chronicle. `https://github.com/paragonie/chronicle`.

[52] PARITY TECHNOLOGIES. wasmi - WebAssembly (Wasm) Interpreter. `https://github.com/paritytech/wasmi`.

[53] PINTO, S., AND SANTOS, N. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv. 51*, 6 (jan 2019).

[54] QEMU. QEMU - A generic and open source machine emulator and virtualizer. `https://www.qemu.org/`.

[55] SAR, C., AND CAO, P. Lineage File System, 2005. `http://theory.stanford.edu/~cao/lineage`.

[56] SECURE SCUTTLEBUTT CONSORTIUM. Secure Scuttlebutt. `https://scuttlebutt.nz/`.

[57] SERDE-RS. serde. `https://serde.rs/`.

[58] SIMMHAN, Y., PLALE, B., AND GANNON, D. Karma2: Provenance Management for Data-Driven Workflows. *Int. J. Web Service Res. 5* (04 2008), 1–22.

[59] SUBSTRATE.IO. substrate - The Blockchain Framework for a Multichain Future. `https://substrate.io/`.

[60] TAMRAKAR, S. *Applications of Trusted Execution Environments (TEEs)*. Doctoral thesis, School of Science, 2017.

[61] THE OPENSSL PROJECT AUTHORS. OpenSSL. `https://www.openssl.org/`.

[62] TRUSTED COMPUTING GROUP. Trusted Platform Module Library Specification - Part 1: Architecture, Nov 2019. `https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf`.

[63] TRUSTED LABS. Security Evaluation of Trusted Execution Environments: Why and How?, 2013.

[64] TUNSTAD, P., KHAN, A. M., AND HA, P. H. Hyperprov: Decentralized resilient data provenance at the edge with blockchains. *CoRR abs/1910.05779* (2019).

[65] VAN DIJK, M., SARMENTA, L. F. G., O'DONNELL, C. W., AND DEVADAS, S. Proof of Freshness : How to efficiently use an online single secure clock to secure shared untrusted memory.

[66] WASMER.IO. wasmer - The Universal WebAssembly Runtime. `https://github.com/wasmerio/wasmer`.

[67] ZAFAR, F., KHAN, A., SUHAIL, S., AHMED, I., HAMEED, K., KHAN, H. M., JABEEN, F., AND ANJUM, A. Trustworthy data: A survey, taxonomy and future trends of secure provenance schemes. *Journal of Network and Computer Applications 94* (2017), 50–68.

[68] ZHANG, N., SUN, K., SHANDS, D., LOU, W., AND HOU, Y. T. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016.