Improving Cloud Security using Secure Enclaves

Jethro Beekman

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-219
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-219.html

December 22, 2016

Copyright © 2016, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Improving Cloud Security using Secure Enclaves

by

Jethro Gideon Beekman

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Chair
John Louis Manferdelli
Assistant Professor Raluca Ada Popa
Associate Professor Deirdre Kathleen Mulligan

Fall 2016

Improving Cloud Security using Secure Enclaves

Copyright 2016
by
Jethro Gideon Beekman

Abstract

Improving Cloud Security using Secure Enclaves

by

Jethro Gideon Beekman

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor David Wagner, Chair

Internet services can provide a wealth of functionality, yet their usage raises privacy, security and integrity concerns for users. This is caused by a lack of guarantees about what is happening on the server side. As a worst case scenario, the service might be subjected to an insider attack.

This dissertation describes the *unalterable secure service* concept for trustworthy cloud computing. Secure services are a powerful abstraction that enables viewing the cloud as a true extension of local computing resources. Secure services combine the security benefits one gets locally with the manageability and availability of the distributed cloud.

Secure services are implemented using secure enclaves. Remote attestation of the server is used to obtain guarantees about the programming of the service. This dissertation addresses concerns related to using secure enclaves such as providing data freshness and distributing identity information. Certificate Transparency is augmented to distribute information about which services exist and what they do. All combined, this creates a platform that allows legacy clients to obtain security guarantees about Internet services.

K. has seen for himself that the court officials, including some who are quite high up, come forward without being asked, are glad to give information which is fully open or at least easy to understand, they discuss the next stages in the proceedings, in fact in some cases they can be won over and are quite willing to adopt the other person's point of view. However, when this happens, you should never trust them too far, as however firmly they may have declared this new point of view in favour of the defendant they might well go straight back to their offices and write a report for the court that says just the opposite.

— Franz Kafka, The Trial

Contents

Contents ii

List of Figures iv

List of Tables v

1 Introduction 1

2 Background and related work 4
- 2.1 Trustworthy computing . . . 4
- 2.2 Verifiable cloud services . . . 5
- 2.3 Secure Enclaves . . . 6
- 2.4 Intel Software Guard Extensions . . . 8
- 2.5 Rollback protection . . . 9
- 2.6 Services with untrusted servers . . . 10
- 2.7 Certificate Transparency . . . 11

3 Secure services 12
- 3.1 Threat model . . . 12
- 3.2 Overview . . . 13
- 3.3 Interface . . . 13
- 3.4 High-level design . . . 16
- 3.5 Implementation . . . 18
- 3.6 Possible applications . . . 19

4 Rollback protection (Hugo) 21
- 4.1 Overview . . . 23
- 4.2 Design of Hugo . . . 25
- 4.3 Obliviousness . . . 31
- 4.4 Counter backends . . . 33
- 4.5 Evaluation . . . 35

5 Attestation Transparency 39

5.1 Overview . . . 39
5.2 Client verification of secure services . . . 42
5.3 Attestation Transparency . . . 43
5.4 Incremental deployment — logs . . . 43
5.5 Validating enclave identities . . . 45
5.6 Enclave policies . . . 46
5.7 Incremental deployment — clients . . . 48

6 Discussion and open research questions **49**
6.1 Limitations . . . 49
6.2 Adoption . . . 50

7 Conclusion **52**

Bibliography **54**

Glossary **60**

List of Figures

3.1 Secure service threat model. The shaded areas may be controlled by the adversary. 12
3.2 Secure service architecture. The shaded blocks are not trusted. 14
3.3 Key management and setup procedure for a secure service. 17

4.1 Architectural diagram showing how Hugo is incorporated into existing applications. 24
4.2 Collect-commit pipeline with two types of request shown. 29
4.3 Dependency graph between various variables in Trace 4. 33
4.4 Latency of requests for different configurations. 38
4.5 Throughput of requests for different implementations. 38

5.1 Overview of Attestation Transparency. 41

List of Tables

3.1 Secure storage types 15
3.2 Rust types of values in different memory regions 19

4.1 Comparison of Hugo's throughput and latency versus existing systems. 35

Acknowledgments

First and foremost I wish to thank John Manferdelli for *de facto* being my academic advisor. Without John, the work in this dissertation might very well have never existed.

I also wish to thank David Wagner for making this arrangement possible, and for seemingly having previously already looked at every single computer security subject I'm interested in. Thanks to Raluca Ada Popa for our brief but effective collaboration and thanks to Elad Alon for supporting me while I figured out what I wanted to do in graduate school.

Thanks to all my colleagues at U.C. Berkeley for the many interesting and insightful discussions about computer security, the Internet and the world. I also wish to thank Angie and the rest of the support staff at the Electrical Engineering and Computer Sciences department.

And finally, thanks to my parents for advising me in everything I do and to my brother and sister for being awesome.

Chapter 1

Introduction

End users increasingly perform important computing activities online in the *cloud*. This is convenient for them but the guarantees they get about those activities are significantly reduced from an ideal *local* computing model where applications are run on trusted machines, inaccessible to adversaries, using software installed and maintained by knowledgeable trusted personnel known to the end user. On well-managed local machines, users can be confident that the software they use is the version they expect, with known behavior and mechanisms to prevent unauthenticated access to their data and unauthorized modification to the software itself.

Cloud services typically do not provide similar guarantees, which raises privacy, security and integrity concerns [17, 34]. Who will have access to my data, intentionally or unintentionally? Will the service continue to work properly tomorrow? Can the service read my data and use it for purposes I didn't have in mind? Will my data still exist in the same form tomorrow? Could a malicious system administrator at the service read or modify my data? If a system administrator's credentials are breached, could an attacker gain access to my data? For current Internet services, the answers to these questions are often unsatisfying.

On the other hand, cloud services provide many benefits that local users don't get. Cloud services provide availability through redundancy and replication and they remove the burden of maintenance from the user. In addition, the sheer number of available services seems to be surpassing the number of applications available locally.

The ideas presented in this dissertation aim to combine benefits from the cloud-based service model with some of the guarantees with which local computer users are familiar. These ideas will help us move towards the ultimate goal of trustworthy cloud computing:

trustworthy cloud computing

> *being able to outsource computational needs while maintaining the privacy and integrity of both data and computation*

A core component we use to reason about and validate our ideas is the *unalterable secure service* (or just *secure service*). These services can be secure replacements for any component in a *service-oriented architecture* [48]. Such a service must provide the following three security properties:

S1. *Handle user data securely.* Data security is defined in terms of three sub-properties *privacy, integrity, and freshness.* This is a key property necessary for users to gain trust in cloud computing, for it is exactly these properties users rely on when using local computing for security reasons.

- **a.** *Privacy.* Unauthorized parties must not be able to learn bits of information that are being handled or stored by the secure service.
- **b.** *Integrity.* Unauthorized parties must not be able to tamper with information that is being handled or stored by the secure service, nor must they be able to influence the result of any computation being done by the secure service.
- **c.** *Freshness.* Unauthorized parties must not be able to present an unmodified historic set of then-actual information as current through the secure service.

S2. *Protect against insider attacks.* Even an insider must not be able to break any of the other defined security properties. For instance, the insider might be a malicious system administrator, a system operator whose credentials have been compromised, or even a government order that compels the service to provide access to user data [23]. Of course insider attacks can occur in the local computing model. However, in that scenario physical and virtual local access is controlled by the same party controlling the computation.

S3. *Service operation verifiable by clients.* A client of a secure service must be able to verify the correct operation of the service and its adherence to these security properties. If it can't, the system would be vulnerable to imposter attacks, where the service looks legitimate but in fact is an insecure version of the same service.

A more detailed threat model under which these properties must hold is described in §3.1. In addition to those security properties, a platform built to support secure services must be practical in order to be adopted by the industry. Therefore we set the following three practicality goals:

P1. *Practical performance.* The orders of magnitude of slowdown for cryptographically secure computation [71] or oblivious computation [43] are not going to lead to widespread adoption of those schemes in the near future. We set out to achieve minimal performance loss compared to an insecure baseline.

P2. *Incremental deployment with legacy client support.* Worldwide technologies that require immediate adoption by a large number of parties to be useful are unlikely to be adopted at all. We aim to design a system with immediate benefits for its users without putting

additional requirements on other users or services. Additionally, users of secure services shouldn't need to change any client software to enjoy most of the security benefits conferred by our platform.

P3. *Support for software updates.* It's an inconvenience of life that software will need to be updated. Even though we describe our services as *unalterable*, they must be able to have their software updated as well, while maintaining all security properties in the process. It shouldn't be possible to update the service software to a version that is less secure.

This dissertation describes how to use secure enclaves to build secure services that have these properties and meet these goals. Secure enclaves (§2.3) are a trusted computing primitive providing fully isolated execution, sealing and remote attestation. While we evaluate our concept with Intel SGX [46] in mind, everything can be generalized to fit the generic idea of a secure enclave.

Essentially, a secure service (§3) is a TLS server running inside a secure enclave hosted in the cloud. By terminating the TLS secure channel inside the enclave, security is guaranteed. The application protocol can be any suitable protocol for Internet services, such as HTTP REST or IMAP, and possible application are discussed in §3.6.

Persistent state stored by untrusted third parties on behalf of secure enclaves can be subject to rollback attacks. We identify shortcomings in current rollback protection mechanisms, especially when trying to combine them with long-running services. We introduce a *rollback budget* to enable a trade-off between security and performance and implement it in Hugo (§4).

Using transparency of code, we both enable the ability to update secure services as well as confer security benefits to legacy clients. Attestations can bind code identities to cryptographic keys, and making them publicly available through public logs will allow anyone to audit code identities on behalf of others such as legacy clients. Attestation Transparency (§5) also enables detection of software updates, which deters service providers from implementing insecure updates.

This dissertation would not have been possible without several recent advancements in security technologies upon which this work builds. The next chapter discusses those technologies.

Chapter 2

Background and related work

2.1 Trustworthy computing

Bill Gates defined *trustworthy computing* in 2002 [24] as "computing that is as available, reliable and secure as electricity, water services and telephony." A lot of research before and after that internal company memo has aimed to achieve that goal.

An early line of research strove to verify the integrity of a known system stack. This was important since malicious or unreliable system software would certainly prevent the secure operation of even well written applications which were themselves safe. The AEGIS system [5] proposed verifying the entire stack by having a chain of integrity checks where each step in the boot process verifies the next before proceeding. This work has been extended to protect other critical software like BIOS code which, if corrupted, presents the application developer with insurmountable barriers for safe operations. Parno et al. provide an overview [49] of the relevant techniques and research in this area.

Wobber et al. describe how in the Taos operating system they treat code (represented, say, as a cryptographic hash) as a first class security principal which can be authenticated [68]. This enabled distributed programs to establish an authentication and authorization model that was as rich and reliable as that for a single program running in batch mode on a single machine. It is not always desirable to attest directly to code principals, as software can change frequently and can exist in many different configurations. Property-based attestation [16] aimed to solve this by using properties of software, instead of the software itself, as security principals.

Since secure distributed computing relied on increasingly well studied and accepted cryptographic mechanisms, researchers sought a key management mechanism that allowed remote verification of program identity and isolation properties of program elements running on widely dispersed machines. Trusted computing primitives combining a discrete security chip [63] coupled with processor features [31] provided the necessary underlying capabilities. Brannock et al. propose a Secure Execution Environment [12] with properties similar to our *secure enclave*.

Researchers recognizing that even with the foregoing advances, large TCBs made security difficult to assure and maintain and thus attempted to minimize the footprint to help ensure that security guarantees could be credibly met. *Protected module architectures* (PMAs) have pioneered the way towards TCB minimization in trusted computing. Many of the ideas apply directly to secure enclaves.

Nizza [28] provides the ability to run “secure applications” on top of a small TCB alongside a regular OS. This is achieved through judicious use of isolation and privilege separation using an L4 microkernel. While this provides good security, there is no hardware root-of-trust and operation can not be verified.

Flicker [45] emphasized application of the Trusted Computing primitives on small services within an application isolating them from the rest of the application and the operating system, providing a PMA abstraction on x86 platforms. For example, Flicker enclaves were well suited as a virtual Hardware Security Module or as an authentication enclave that used a long term secret; the security model ensured that the OS, other applications and other portions of the same application could not get private key material. However, it was shown that Intel Trusted Execution Technology, which Flicker is based on, is not secure [69].

Various other systems were proposed to marry cloud computing with trusted computing, such as Self-service Cloud Computing [13], Cryptography-as-a-Service [10], and MyCloud [42]. These systems focused on providing trust in the cloud hypervisor to a customer of the cloud service provider, not on providing trust of Internet services to users of those services.

SGX [30] employed specialized hardware for this same purpose and also encrypted enclave memory in DRAM thus protecting from an adversary with system bus access. Several recent works employ SGX to protect cloud service components. Haven [6] employed SGX to run MS SQL Server entirely in a secure enclave. Clients of that database server could benefit from the Attestation Transparency Framework to verify the server they’re connecting to. VC3 [55] implements secure MapReduce operations for the Hadoop distributed computation platform using SGX.

2.2 Verifiable cloud services

While PMA designs have focused on TCB minimization, others have researched building systems that allow verifiable operation of services in the cloud.

Hawblitzel et al. propose a system using Ironclad Apps [26] for secure remote computations by formally verifying the entire server stack.

CloudProxy [44] provides a layered abstraction for trusted computing primitives. While not providing the same isolation guarantees as secure enclaves it does provide sealing and attestation primitives. It provides hierarchical attestation from the hardware root-of-trust at boot, to the operating system and from there to individual applications. This allows clients to verify the proper functioning of functioning of services run on this platform.

Haven [6] was the first to propose running an entire service inside an enclave. Haven employs a Windows-based unikernel capable of running inside Intel SGX, and runs Microsoft SQL Server or Apache on top of that. TCB minimization was not a goal of Haven, and it includes hundreds of megabytes of binaries in its TCB.

2.3 Secure Enclaves

We define a *secure enclave* as (a) an isolated process, executed on a platform that provides confidentiality and integrity of code and data as well as (b) sealing and (c) attestation. In general, these technologies allow initializing an isolated and perhaps encrypted block of memory with a known program. Access to application memory is restricted by hardware and external access to the software is similarly restricted to identified entry points into the code. The software loaded in an enclave is also measured[1], allowing the hardware to attest to another party that the expected software was properly loaded and initialized and that the enclave software is isolated from other software running on the computer. The platform also provides a way to encrypt data so that the encrypted data can only be decrypted by this particular instance of the code running on this particular hardware. Different technologies provide such secure enclaves, including Intel SGX [46, 4], AMD Secure Encrypted Virtualization [35], IBM SecureBlue++ [11], TPM-based Flicker [45], and perhaps ARM Trustzone [67]. Our design builds on these general concepts and is not tied to any particular platform.

Fully isolated execution

Isolated execution of a process restricts access to a subset of memory to that particular process or *enclave*. No other process on the same processor, not even the operating system, hypervisor, or system management module, can access that memory. Additional security measures may be provided by the platform such as memory encryption or a separate memory bank used solely for secure enclaves. Generally, as part of the isolated execution, the enclaves have no or limited access to the I/O system. This isolation dramatically reduces the Trusted Computing Base (TCB) for the enclave, which comprises only the enclave code itself and the secure enclave platform. This is a break from the traditional hierarchical kernel–userspace privilege model, in which the entire operating system kernel is generally considered to be part of the TCB of a user process.

Sealing

Sealing is the authenticated encryption of data with an encryption key based on the identity of the enclave and the platform it is running on. The enclave's identity ($I_{enclave}$) could for

[1] A *measurement* is typically a cryptographic hash of the software as loaded together with any configuration information which may affect the software behavior.

example be a hash of its binary as it was loaded into memory and that of the platform could be a unique identifier embedded in the processor hardware. This provides secure enclaves with the ability to have other parties, such as the operating system, store information securely on behalf of the enclave. No process other than the exact same enclave running on the same physical processor will be able to decrypt such data.

Sealing, or more generally, encryption is important for secure enclaves since the secure enclave concept does not include secure persistent storage. This is generally solved by using an untrusted persistent store and storing data only in encrypted form. This provides a form of secure persistent storage.

Define *authenticated encryption* for a key K, message m, and ciphertext c as

$$c = E(K, m)$$

$$m = D(K, c).$$

Similarly, define sealing and unsealing for a message m and sealed text s as

$$s = E_{\text{seal}}(m)$$

$$m = D_{\text{seal}}(s).$$

Remote attestation

Remote attestation is the ability to prove to third parties that you are running a secure enclave with a particular identity securely on your hardware. The mechanism allows software to make statements that can be verified remotely by communicating parties, using *attested statements*. When talking about secure enclaves, some hardware-based root of trust, H, will attest that it is running a program with identity (measurement) I. In order for such a program to communicate with the outside world securely, it will need an encryption key K, and a way to securely announce to the outside world that it controls that key. Attestation provides such a mechanism: the hardware makes a statement of the form[2]

$$A(I, K) = \ll H \text{ says ``}H \text{ runs } I \text{ which says } [K \text{ speaks for } I]\text{''} \gg .$$

Platforms providing secure enclaves often provide ways for an entity I_1 to endorse a particular program with identity I_2. For example, I_1 might cryptographically sign I_2, and this signature can be verified as part of loading I_2. Such an attestation is of the form

$$A(I_1 : I_2, K) = \ll H \text{ says ``}H \text{ runs } I_2 \text{ which says } [K \text{ speaks for } I_2]\text{'' and ``}I_1 \text{ endorses } I_2\text{''} \gg .$$

[2]Following the Taos language [68].

If the platform can not verify the endorsement itself, a similar statement can still be formed by including the endorsement directly, as in

$$A(I_1 : I_2, K) = \ll H \text{ says "} H \text{ runs } I_2 \text{ which says } [K \text{ speaks for } I_2]\text{"} \gg \text{ and } \ll I_1 \text{ says "} I_1 \text{ endorses } I_2\text{"} \gg .$$

If an enclave depends on other enclaves for its security guarantees, attestation can be performed transitively. Take for example an enclave B that relies on an enclave A and a party C requests an attestation of B. In a prior initialization phase, B will have requested an attestation of A and verified it. Because this initialization and verification is "hard-coded" in B, it is part of B's identity. C can therefore be assured that when the attestation of B verifies correctly, A was also properly attested to.

2.4 Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) [46, 4, 27, 30] are a recent hardware technology and instruction set extension providing secure enclaves. A special set of instructions can measure and encrypt a memory region before transferring execution control to it. The trusted computing base of SGX-based secure enclaves encompasses only the processor hardware, its microcode and firmware, and the enclave image itself. In particular, a hypervisor or operating system is *not* part of the TCB. Data stored in memory regions belonging to the enclave is encrypted before it leaves the processor, so that the memory bus is also not part of the TCB. The security of this system is predicated on the correct functioning of the processor hardware and the SGX instruction set.

The processor effectively adds an additional privilege level *enclave mode*, which is both more and less privileged than other privilege levels such as *user mode* and *kernel mode*. Each enclave has its own enclave mode address space, and the processor maintains its integrity. Enclaves are associated with a particular user process's address space and additionally have access to all process user memory. The process when running in user mode does not have access to enclave memory. Also, unlike regular user memory, the kernel has no access to the enclave memory, which is also enforced by the processor.

Attestation

SGX-enabled hardware can generate *reports*: integrity-protected statements about the enclave generated by the hardware

$$\text{Report}_{\text{local}} = \text{MAC}\left(I_{\text{enclave}} \| I_{\text{signer}} \| D_{\text{user}}\right).$$

The MAC key is different for each processor and private to the enclave that requested the report—only that enclave on the same processor can verify the report. I_{enclave} is the measurement of the code of the enclave the report is generated of and I_{signer} is the public

key that was used to sign that enclave before loading it. D_{user} is an arbitrary value that can be specified by the enclave when requesting the attestation report. This can be used to bind data to the attestation.

A special secure enclave provided by Intel, called the *quoting enclave*, can replace the MAC with a signature

$$\text{Report}_{\text{remote}} = \text{Sign}\left(I_{\text{enclave}} \| I_{\text{signer}} \| D_{\text{user}}\right).$$

The signature private key is private to the processor and cannot be used improperly or for any purpose. The corresponding public key is made available by the vendor, and a third party can use it to verify that the report was created by actual Intel hardware, such that

$$A(I_{\text{signer}} : I_{\text{enclave}}, D_{\text{user}}) = \text{Report}_{\text{remote}}.$$

Sealed storage

A special instruction can generate an enclave-specific *sealing key.* The key is derived as

$$K_{\text{seal}} = H\left(I_{\text{enclave}} \| K_{\text{device}} \| \ldots\right)$$

where K_{device} is a hardware-embedded secret unique to this device. The enclave can use this key to encrypt data which can only be decrypted by the same enclave running the same code on the same hardware, such that

$$E_{\text{seal}}(m) = E(K_{\text{seal}}, m)$$

$$D_{\text{seal}}(s) = D(K_{\text{seal}}, s).$$

A different key can also be derived as $K_{\text{derived}} = H\left(I_{\text{signer}} \| K_{\text{device}} \| \ldots\right)$. This key can be used to transfer data between enclaves running on the same hardware that were signed by the same public key.

2.5 Rollback protection

None of the PMAs and service frameworks in the previous sections protect against rollback attacks. However, there are schemes designed to add rollback protection to PMAs.

Memoir [50] was the first robust rollback protection system with a small TCB based on Flicker. Memoir works by building a hash chain of every request from the initial state to the current state. Every link in the hash chain consists of the previous hash and the current request. The most recent hash is embedded in the most recent state, and the most recent hash is stored securely in TPM NVRAM. This provides rollback protection because only the state with the current hash embedded in it will be accepted. It also provides crash resilience because inputs can be replayed exactly to get to the right point in the the hash

chain. Because inputs and their resulting output states are strictly ordered, it is not possible to perform concurrent operations. Memoir requires an NVRAM write for every input, or requires putting a trusted shutdown routine and uninterruptible power supply (UPS) in the TCB.

ICE [61] obviates the need for a UPS by basically building one into the trusted hardware. The trusted hardware will store the most recent hash in NVRAM on shutdown, whether clean or due to a power failure. Unfortunately, such functionality is not currently available in commodity hardware.

Ariadne [62] implements many of the same concepts as Memoir, except it uses a monotonic counter instead of storing a hash. For every input, the input and the previous state are stored together with the next value of the counter. Then the counter is incremented and computation proceeds. The monotonic counter is provided securely by a TPM. This provides rollback protection because only the state with the current counter value in it will be accepted. It also provides crash resilience because inputs can be replayed exactly to rerun a crashed computation and obtain its output. Concurrency is not supported for the same reasons as Memoir, and a TPM counter increase is required for every input.

2.6 Services with untrusted servers

Consistency schemes without the need for trusted hardware while providing guarantees similar to, but weaker than, full rollback protection have been proposed.

Secure Untrusted Data Repository (SUNDR) [41] is specifically designed to work with data coming from multiple trusted users. SUNDR provides a guarantee called *fork consistency*. Users must either see the same state, or a fork of a previous state. A fork can never incorporated changes from another fork without being detected. A dishonest server must maintain the fork forever and this could be detected out-of-band by different clients.

Caelus [37] provides consistency guarantees for an untrusted service to at least one trusted client. It does so by keeping a history of all operations performed and periodically attesting to this history. Clients will be able to detect inconsistencies after one such attestation period by checking whether its operations have been recorded properly.

Verena [36] is a web application framework that provides end-to-end integrity guarantees against a compromised web server including rollback protection. It achieves the latter property by storing hashes securely at a separate hash server, and updating them upon every user modification. Verena assumes that the hash server and the main server on which the application runs are mutually distrustful. As such, Verena is susceptible to software attacks to the hash server and does not provide crash resilience or concurrency. Moreover, it is not clear that Verena integrates well with obliviousness.

2.7 Certificate Transparency

Attacks on PKI [1] threatened the trustworthiness of co-dependent services which can benefit from the execution flexibility of cloud computing and the vast quantity of community curated data. This prompted the development of the Certificate Transparency (CT) framework [38] to highlight misissued certificates. As the name implies, it aims to provide transparency to the issuance of TLS certificates. CT makes all legitimate TLS certificates a matter of public record, making it trivial to identify misissued certificates. The framework consists of several parts.

Public append-only logs A CT log server maintains a log of all certificates submitted to it. The log is structured as a Merkle tree which allows efficient verification of additions to the log. When submitting a certificate to the log server, the server will return a Signed Certificate Timestamp (SCT). The SCT is a promise that the server will include the certificate in the log within a certain time limit, the *maximum merge delay*. The SCT can be used as proof to other parties that a certificate is part of the public record.

Monitors A monitor watches one or more CT log servers for suspicious changes. For example, a domain owner might know that all its certificates are issued by a particular CA. If a certificate for their domain issued by a different CA appears in a log, the monitor raises an alarm. The administrator can then act upon that alarm, e.g., by demanding the revocation of the phony certificate.

Auditors An auditor watches one or more CT log servers for consistency. It checks that the Merkle tree is updated consistently and that certificates are included as promised by SCTs. If it detects any inconsistency, it raises an alarm. The CT log owner will then need to explain the discrepancy or risk being shut down.

Browsers Once the CT framework is fully operational, TLS clients such as browsers can demand proof from TLS servers that the server's certificate appears in a log. TLS servers can provide this proof in the form of SCTs. If a certificate does not appear in the logs, that is suspicious, and the client can choose to abort the connection attempt.

Chapter 3

Secure services

We want application service providers on the Internet to be able to host *secure services*. These services must be able to store and handle user data securely. By secure, we mean that the data's confidentiality and integrity is preserved, in the face of attacks within the scope of the threat model set forth below, which includes insider attacks.

This chapter presents the architecture we use to implement *unalterable secure services*. Unalterable here means that the functionality of the service cannot be changed. This allows a client of the service to view the service as an extension of the client itself and not just a third-party program subject to the whims of another entity.

Our architecture runs services inside a secure enclave as well as to encourage secure software development. To reduce the attack surface, the architecture presents a limited interface to the programmer that should be sufficient for Internet services, and the interface is implemented in a *memory-safe and type-safe language,* Rust [53].

3.1 Threat model

We assume the server hosting the service uses some secure enclave technology that prevents the adversary from accessing the code and data running in the enclave. We allow adversaries all the capabilities of an active network attacker as well as full control over non-enclave

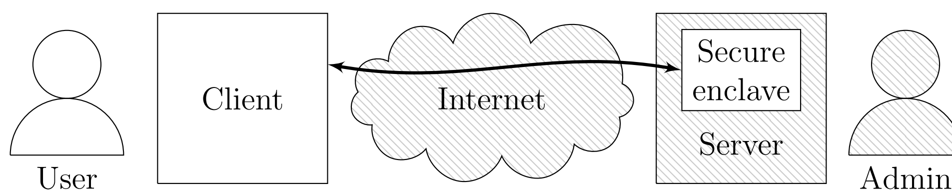


Figure 3.1: Secure service threat model. The shaded areas may be controlled by the adversary.

software running on the computer hosting the service (e.g., for SGX enclaves, this includes control over the operating system). For instance, an insider might add malicious software, or service provider personnel might accidentally misconfigure the service; these are included in the threat model. The adversary also has the ability to run its own servers mimicking real services. We assume the user's client is secure and cannot be tampered with. The threat model is depicted in Figure 3.1.

We consider availability out of scope. A cloud provider will have a strong economic incentive not to deny service. However, if a malicious insider wishes to destroy all user data or deny access to the service, they can do so.

3.2 Overview

The basic idea is to run all of the service code—including TLS session establishment, request handling, and storage—in an enclave on the server. This provides isolation and ensures that even insiders on the server cannot tamper with the running code or memory of the service. Also, we use sealed storage to prevent malicious insiders from reading or modifying data stored by the service on persistent storage: effectively, all data is encrypted before it leaves the enclave.

The user connects to the server, using TLS to establish a secure channel between the client and server. We use remote attestation to allow the user to verify what code is running in the enclave: secure hardware on the server provides a signed statement indicating what code has been loaded into the enclave. A fully attestation-aware client would then use this attestation to verify that the server is running the expected code.

Conveniently, the TLS protocol is widely supported and provides a secure channel to the server, while verifying the authenticity of that server. This means legacy clients will have no trouble connecting to the server. As is usual for TLS, the client checks that the server's TLS certificate is valid and authenticates the server using the public key found in this certificate. Our system extends the guarantees provided by this authentication step by further constraining the use of the private key.

In particular, a secure service runs inside a secure enclave and it will generate its TLS private key there. The TLS private key will never leave the enclave in unencrypted form; it is stored using sealed storage, so that only the enclave can retrieve it. Thus, even insiders cannot learn the service's TLS private key.

In our architecture (Figure 3.2), only the CPU and the code inside the secure enclave are trusted.

3.3 Interface

The secure enclave has no input/output capabilities and relies on an *untrusted driver* for (insecure) access to the outside world. The untrusted driver is part of the host operating

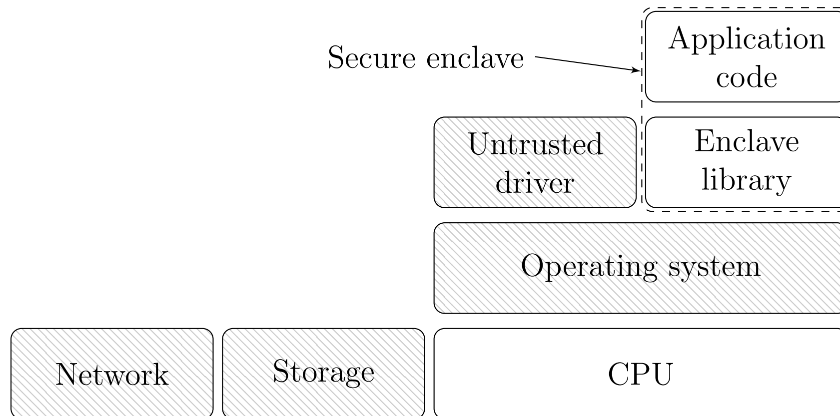


Figure 3.2: Secure service architecture. The shaded blocks are not trusted.

system and provides persistent storage (e.g., via C Standard I/O Streams), networking (e.g., via BSD sockets), and inter-process communication (e.g., via stdin/stdout/stderr). On top of this, the *secure enclave library* implements encrypted networking using TLS (e.g., using a standard TLS library), encrypted and sealed storage, attestation, and IPC. It is these features that are exposed to the *application code*. Software developers use these secure primitives to write their secure service. The secure enclave library and application code together form the secure enclave.

Secure networking

interface SecureClient

  static fn $connect(address : \mathsf{string}) \to \mathsf{stream}$

  static fn $connect_enclave(address : \mathsf{string}, attester : \mathsf{Attester}, id : \mathsf{blob}) \to \mathsf{stream}$

interface SecureServer

  static fn $listen(channel : \mathsf{int}, key : \mathsf{Key}) \to \mathsf{SecureServer}$

  fn $accept() \to \mathsf{stream}$

  fn $accept_enclave(attester : \mathsf{Attester}) \to (\mathsf{blob}, \mathsf{stream})$

Secure networking is provided using TLS. The secure client interface allows connecting to an Internet address using TLS and verifying the connection using default methods (per RFC 5280 [18] and RFC 6125 [54]). There is also an option to connect to another secure service running in an enclave and have it attest to the secure channel parameters. The client will then verify that the attestation is valid and matches the expected server enclave identity.

The secure server interface will listen on a specified port and accept TLS connections from clients. This is the main communication mechanism for a service using our architecture. There is also an option to accept connections from clients that are themselves running in an enclave and have that client identify itself and attest to the secure channel parameters.

Secure storage

interface SecureStorage

- **static fn** $sealed()$ → SecureStorage
- **static fn** $keyed(id$: string, key : blob) → SecureStorage
- **static fn** $keyed_sealed(id$: string, key : blob) → SecureStorage
- **fn** $read(name$: string) → blob
- **fn** $write(name$: string, $value$: blob)
- **fn** $delete(name$: string) → bool
- **fn** $list()$ → string[]
- **fn** $exist(name$: string) → bool

The secure storage interface allows the application to store persistent data safely. The interface provides access to different data objects identified by their name or path, while using an encrypted storage backend. There are three possible keying schemes for encrypting the data before storage: using the sealing key, a user key, or both (encrypted with a user key, then sealed). The different schemes have different benefits as shown in Table 3.1. In case of

Table 3.1: Secure storage types

Benefit	Sealed	Keyed	Both
Protect against blanket access after breach		✓	✓
Protect against offline attack versus weak user key	✓		✓
Recover data after hardware failure		✓	

a breach—e.g., due to a faulty update (see §5.5), or a code bug—using sealing only, in its simplest form, is inadequate. Further, sealing—in its simplest form—is hardware-dependent and any sealed data is lost after a hardware failure. Using a per-user key based on the user's password enables password-guessing attacks if the password is weak. This is of particular concern since in addition to online attacks via the normal service authentication mechanism that all Internet services have to deal with, in our model an adversary can perform offline attacks on the stored data.

Asymmetric cryptography

interface Key

- **static fn** $new()$ → Key
- **static fn** $deserialize(data$: blob) → Key
- **fn** $serialize()$ → blob
- **fn** $get_certificate_signing_request(subject$: string, $attester$: Attester) → blob
- **fn** $is_certificate_valid()$ → bool
- **fn** $set_certificates(cert$: blob[ ])

The key interface abstracts over a public-private key pair together with a certificate chain for that public key. A key can be constructed by generating a new one or by deserializing a byte stream (presumably obtained from sealed storage). A new certificate signing request can be generated for a particular subject, and an attestation can be included in the request as well. The purpose of this is detailed in §5.3.

Remote attestation

interface Attester

  static fn $attest(key : \mathsf{blob}) \rightarrow \mathsf{blob}$

  static fn $verify(statement : \mathsf{blob}, id : \mathsf{blob}) \rightarrow \mathsf{blob}$

The attestation interface allows a secure enclave to have the hardware attest to a key. It can also verify that attested statements match a certain identity and extract the key that was attested to.

Inter-process communication

interface IPC

  static fn $open() \rightarrow \mathsf{stream}$

Services may use inter-process communication, e.g., for inputting configuration data and logging. Since this channel is not secure, no sensitive information should be logged through this channel, and it must not be used for configuration that changes the security properties of the service. Instead, such configuration needs to be part of the enclave measurement.

3.4 High-level design

Using the primitives defined in the previous section, we can build an *unalterable secure service*. Keeping the private key K_{server}^{-1} of a TLS server in sealed storage and never exporting the key outside the enclave ensures only a particular secure enclave instance can have access to it. This means that when one establishes a TLS connection with a server that uses that K_{server} to authenticate its key exchange, the server endpoint is guaranteed to terminate inside the enclave.

Since the private key should never exist outside the enclave, it must be generated inside the enclave. The key setup procedure for the secure service enclave is shown in Figure 3.3. Input and output happens through the IPC channel.

All the service's static content (e.g., for a website, images and layout elements) must be included in the server binary that will be measured upon enclave startup. All dynamic/user content must be stored in secure storage.

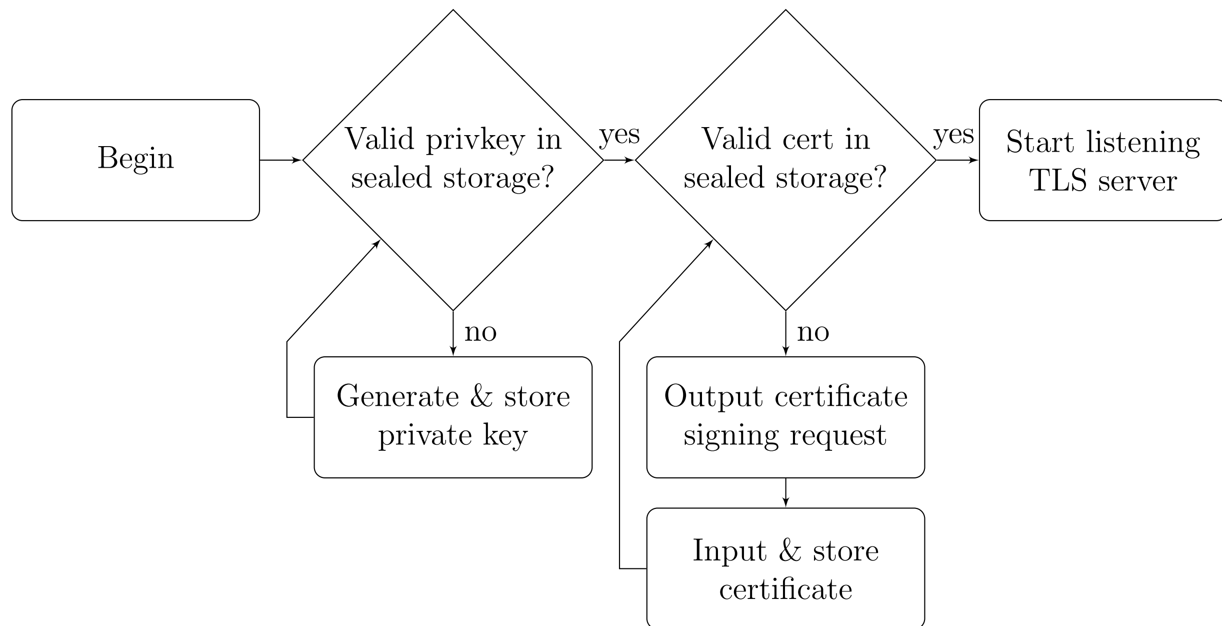


Figure 3.3: Key management and setup procedure for a secure service.

Horizontal scaling

Once one instance of a service is running, another instance can connect to it and they can both verify that they're instances of the same program. After the verification, sensitive information can be shared (over a secure channel established using a secure key-exchange protocol). Any kind of distributed service can be supported this way.

Multiple enclaves

A service might consist of multiple parts, for example a database server and an application server. The enclaves should validate each other's identity and establish a secure channel between the two. There are at least two secure ways to implement this.

Consider an enclave A that accepts connections from clients and provides controlled access to information based on the client's identity. A second enclave B wants to use enclave A's service and has fixed A's identity in its loaded and measured code. Enclaves A and B establish a secure channel and both attest to their parameters. Enclave B can verify A's attestation and see that the identity matches what is expected. Enclave A can verify B's attestation and provide B access to the information B is authorized to access.

If both enclaves must verify each other's identity using embedded identities, there is a chicken-and-egg problem. Since the identity of a program changes when including a different identity, it's not possible for both programs to have the other's identity fixed in its code.

Also, it's not secure to rely upon a system administrator to sign the identities of the two enclaves, since an insider could falsely sign the identity of a malicious enclave. One solution is to combine multiple programs into a single one with multiple operating modes. Now the same solution used for horizontal scaling can be applied.

Updates

When a service is updated, its persistent data will need to be updated too. Data encrypted with a user-dependent key can be used directly by the newer version. However, since the new service identity will be different from the previous version, all data stored in sealed storage is lost. Sealed data will need to be moved to the new version before the old version can be retired.

A secure channel will need to be established between the old and the new version, see §5.5 for details on the authentication of this channel. Once the channel is established, the old version can unseal the data in question and send it across. The new version receives the data and immediately puts it in sealed storage. If there is too much data to be transferred over the secure channel, instead that data should be encrypted with an enclave-generated secret key. The key itself can then be stored in sealed storage and transferred for updates.

3.5 Implementation

Since the secure service is fully in control of all sensitive data handled by it, care must be taken to ensure data security. The sensitive data includes at least user data and the TLS private key. Memory safety bugs could easily lead to remote code execution inside the enclave, thereby allowing an attacker full control over the data. Similary, concurrency bugs can easily lead to corruption of the sensitive data. Logic bugs could lead to both information disclosure or corruption.

To help prevent such errors, secure enclaves must be implemented using state-of-the-art secure coding techniques. The most secure solution would be to formally verify the entire enclave code. However, the verification of large arbitrary codebases is squarely within the realm of academic research [22]. Secure enclave specific techniques have been developed but verify or enforce only a limited set of properties [57, 56].

If full formal verification is not possible, using a memory-safe language is probably the next best thing. Memory-safe languages can prevent entire classes of bugs. In addition, such languages might prevent the creation of references to non-enclave memory on platforms where such a thing would normally be possible. This would be an additional protection against accidental information disclosure.

Unfortunately, most memory-safe languages require some sort of *runtime*. The use of runtimes is hampered by the limited system interaction of most enclave implementations. This makes porting most languages to run inside secure enclaves difficult.

Rust & SGX

There is one language however which boasts memory-safety without a runtime: Rust [53]. The Rust language and compiler enforce memory-safety at compile time through static analysis. It has many other safety features as well, such as safe concurrency (preventing data races) and first-class error handling (it's not possible to misuse an error return value). We built a set of tools and libraries [7, 8] to use Rust with Intel's secure enclave implementation, SGX. This encompasses the "enclave library" mentioned in Figure 3.2 on page 14.

Rust makes it almost trivial to implement "Information Release Confinement" [56]. By limiting all standard memory allocations to the enclave stack and enclave heap, it isn't possible for a programmer to write sensitive data outside enclave memory. To communicate data with userspace, a second heap in user memory needs to be used. Data stored on the user heap is of a different type than that stored in enclave memory, see Table 3.2, and therefore data can't be accidentally used inappropriately. The developer must explicitly indicate that they want to move data from enclave memory to the user heap or vice versa.

Table 3.2: Rust types of values in different memory regions

	scalar value	multiple scalars
Enclave stack	`T`	`[T;n]`
Enclave heap	`Box<T>`	`Box<[T]>`
Enclave reference	`&T`	`&[T]`
User heap	`UserBox<T>`	`UserSlice<T>`
User reference	N/A	N/A

Side-channel attacks and oblivious computation

Intel SGX explicitly does not aim to protect against most side-channel attacks [33]. Previous work has shown that much information can be inferred from observing the behavior of an SGX process from the Operating System [70]. Developers wishing to defend against such attacks to maintain confidentiality should employ oblivious algorithms [43] or use a secure enclave architecture that has better protections [20]. However, our library must preserve the obliviousness of applications written using the library. This is accomplished by avoiding secret-dependent branches in our code and using constant-time implementations of cryptography algorithms [25, 9].

3.6 Possible applications

Browser-based cryptographic systems

One of the arguments against doing in-browser cryptography using JavaScript is its poor auditability [39, 51]. Even if a user assures themselves of the quality of the cryptographic

system by carefully inspecting a page's DOM tree, there is no guarantee the server will send you the exact same page the next time you visit it. With a secure service, a user does get that guarantee. Because the logic for sending HTTP responses is fixed within the unalterable secure service's identity, a client will receive the same script every time. This in combination with the Web Crypto API [58] brings us closer to being able to do browser-based crypto properly and securely.

Bootstrapping secure web applications

In the web application world, many production updates are pushed out every day. Having to go through the update process and requesting a new TLS certificate every time might not be practical. It is not necessary, however, to include an entire website within the secure enclave.

Instead, one can create a small *core web page* at a well-known URL (e.g. example.com) that will load further web content. Even untrusted content can be included (e.g. from not-audited.example.com) when using a technique such as HTML5 privilege separation [3]. The small core is secure and verified and provides only security functionality to the web application, which should require infrequent changes. The untrusted part of the website can be developed and updated frequently as normal, while not being able to cause harm because of the privilege separation.

Including static external content, e.g. from Content Delivery Networks, is supported securely through the recent Subresource Integrity draft [2]. Websites can include a hash with a URL on an external resource which will be checked by the browser.

Including dynamic external content is trickier. If an external site is known to be a secure service defined in this paper, verifying its known public key should be sufficient to ensure the safety of loading its contents. The Subresource Integrity mechanism could be extended to allow public key pinning on an external resource.

Encrypted e-mail storage server

An e-mail provider could run their SMTP/IMAP stack as two separate secure services. The IMAP server, storing the user's e-mails, will maintain an internal directory of users and corresponding encryption keys. Only the user will have access to their e-mails which are encrypted at rest. The SMTP server, when receiving mail for a local user, will obtain the local public key for that user from the IMAP server and encrypt the received message before handing it to the IMAP server for storage.

This setup provides secure encrypted e-mail storage for legacy IMAP clients including the inability of an insider to obtain the user's e-mails or credentials. Additionally, an SMTP client could verify the server's identity before submitting mail, making sure that the e-mail will get delivered to a secure mailbox.

Chapter 4

Rollback protection (Hugo)

This dissertation proposed secure services which are similar to—but not quite the same as—Protected Module Architectures (PMAs), a concept analyzed extensively in the literature [14, 28, 45]. PMAs such as Flicker are typically described as securing small part of a bigger application. This protected module provides a request-oriented interface, taking some input and a current state, then computing on it to produce an output and an updated state. A PMA could for example be used to securely store a Certification Authority's (CA) private key for certificate signing. Secure services do not fit this model well because they are servicing multiple requests at a time and are using and updating shared state simultaneously.

An important shortcoming of both the secure enclave and PMA models is the inability to store state persistently within the trust boundary. A common way to solve this problem is to use a special enclave-specific sealing key to encrypt state and have an untrusted party store the encrypted blob. While this preserves the privacy and integrity of the state, it does not provide *freshness*. The untrusted party will over time collect many different encrypted states. When the untrusted party presents the enclave with anything but the latest state, this is known as a *rollback attack*. In the example of our versioned file storage, this could emerge as a user not seeing the latest version of a file they uploaded. Current secure enclave-based services such as the Haven SQL Server [6] are susceptible to these attacks.

Rollback attacks have been studied in the context of PMAs [50, 61, 62], but the solutions developed for PMAs—while secure—are not practical for long-running services. In particular, the *latency* involved with updating the secure state for each request is too high and the *throughput* in terms of the number of requests serviced is too low.

We present **Hugo**,[1] a framework and programming paradigm for protecting the freshness of persistent state for long-running services. In what follows, we discuss challenges Hugo faces in achieving these properties and how it addresses them.

Not all inputs modify state: Existing systems [50, 62] implement strict *state continuity* by persistently storing every input to the PMA in a rollback-protected manner before doing any computation. With these mechanisms you're paying an upfront cost for every

[1] From the Dutch word "geheugen" (memory).

single computation. Also, if the input of the computation is large but the output is small, this requires additional storage capacity. Storing every input is both inconvenient and unnecessary.

It is inconvenient because TLS, the most widely used cryptographic protocol for establishing secure channels necessary to communicate with secure services, requires several round-trips to setup such a channel. Each round-trip is a separate input to a PMA, and the rollback protection cost has to be paid each time. Hugo considers rollback protection for application-level requests, not low-level inputs.

It is unnecessary because not all inputs change state. For example, in a web service, many requests are likely to just retrieve information without changing state. In Hugo, rollback protection is applied only to state, not input.

Not all state is rollback-sensitive: Not all of the persistent state *necessarily* needs to be protected against rollback attacks. We conjecture that state rollbacks are only bad when exhausting some *limited resource*. Examples of such limited resources are a privacy budget for a differentially-private database, or a number of tries to enter a password. Consider the latter case in which a particular user has n tries left. If an adversary could try a password, learn the outcome of the password verification, and rollback the state to when there were n tries left, this would allow the adversary to try a virtually infinite number of passwords. Password tries are therefore a limited resource in need of rollback protection. Examples of things that don't need rollback protection are for example the exact number of page views or a login session cache. Hugo lets the developer specify which state is in need of rollback protection, but applies rollback protection opportunistically to all state.

Opportunistic rollback protection: There is a certain overhead to rollback protection. To achieve the best performance, state should only be protected against rollbacks when strictly necessary. Nonetheless, it's desirable to protect all state against rollback when this doesn't hurt performance. To ensure rollback protection is always applied when necessary but opportunistically applied otherwise, we introduce the notion of a *rollback budget*. This budget specifies the amount of state that could potentially be rolled back by an adversary when feigning a crash. The number password tries will have a rollback budget of 0, meaning no state will ever be able to be rolled back.

Crash resilience: It is possible that the latest state was lost, for example, due to a system crash. Rollback protection mechanisms must ensure *crash resilience*, meaning that a crash will not leave the system in an unusable state. On the other hand, an adversary must not be able to game the rollback protection system by feigning a crash.

Existing systems based on *state continuity* [50, 62] can restart computation based on the stored input if there was a crash. Since Hugo does not store input, a different solution is needed.

We surmise that there is enough of the previously-discussed limited resources that exhausting some of the resource while having to redo the computation is not a problem in the rare occurrence of a system crash. For example, if a user tries to login to a system, and the system crashes just as the password is being verified, it is OK if the user has one less try for their password without learning the outcome of the password verification.

When updating state due to a request, Hugo updates the state before outputting the result of the request. This means that in the event of a crash, the historical input can't be replayed to learn the result of the crashed computation. As discussed, this is acceptable. If necessary, the same request can be submitted again when the system is back up.

Obliviousness: Secure enclaves are susceptible to various side-channel attacks [33, 70]. A common defense against such attacks is *oblivious* computation. Hence, it is important that the rollback protection system neither makes unoblivious accesses nor leaks information that the original application did not. We show that Hugo preserves this obliviousness by making accesses that mimic the obliviousness of the application.

Concurrency: Existing systems require a strict ordering to their input and do not support servicing multiple requests at the same time. Hugo supports concurrency trivially for requests that do not change state. For requests that do change state, concurrency is supported by having multiple requests coordinate the rollback budget and state changes.

Keeping trusted state: Rollback protections for PMAs rely on trusted hardware to provide a limited amount of trusted persistent state, e.g. to store a single hash or counter. Hugo also relies on a trusted party to keep a secure monotonic counter, but is flexible in the choice of this party. It could for example be a trusted network service, or in some cases the client of a service built on top of Hugo, or existing commodity hardware (TPM). However, not all secure enclave implementations have direct access to hardware (e.g. Intel SGX). We describe how to set up a secure channel with existing trusted hardware to ensure the proper and secure functioning of the counter.

To summarize, we make the following three key contributions in this chapter:

- We present Hugo, a performant and concurrent rollback protection scheme designed for long-running services. We demonstrate a versioned web-based file storage service built using a Hugo prototype written in Rust on top of Intel SGX, achieving 2–13× the throughput of serialized continuity schemes.
- We show how a *rollback budget* can provide great performance combined with opportunistic rollback protection, achieving 8–13× the throughput and 50% of the latency of serialized continuity schemes.
- We present a flexible trusted counter backend supporting various use cases and hardware, describing how to setup a secure channel with a TPM and how to leverage trust in a client of a secure service.

4.1 Overview

Hugo provides a rollback-protected key-value store abstraction for secure enclaves. It builds this on top of a secure monotonic counter interface and a regular revertible (untrusted) key-value store. This is done by keeping track of the state in the entire key-value store and linking that state to value of the monotonic counter.

Threat model

The threat model for Hugo is the same as in §3.1 on page 12, except that the secure enclave also has access to a trusted counter. The communication channel between the enclave and the counter need not be trusted as we describe how to setup a secure channel between the two. It is within the adversary's capabilities to inspect, modify, and delete any and all data stored in untrusted storage, as well as intercept, modify and block requests made to and responses coming from storage. Likewise, it can intercept, modify and block requests made to and responses coming from the counter. The adversary can interrupt, resume, and terminate the execution of the secure enclave at will. The adversary can run multiple of the same secure enclave simultaneously and replay requests and responses from each to others.

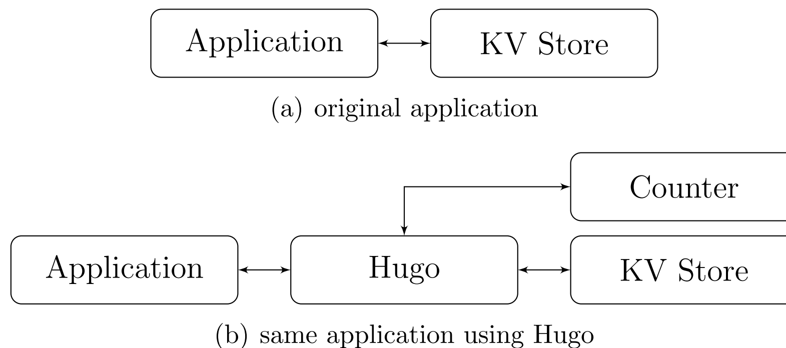
Architecture

Hugo is designed as a small program library that can be interposed in a 'plug-in' manner into an existing secure enclave application, if the application was already using a key-value store for storage. Figure 4.1 shows how Hugo connects to other components in the system.

Whenever the application writes a value to a key, Hugo updates its internal state with the key and the hash of the value. The internal state is serialized and sealed together with the next counter value. Then, the original key and value are written to the backing store, as well as a special state key and the sealed state. Finally the counter is incremented. It is important that an application does not produce any output relating to the state change until *after* the counter has been successfully incremented.

When the application reads a key, Hugo passes the read request through to the backing store. The returned valued is hashed and compared to Hugo's internal state. If the hash is a match, the value is returned to the application. Otherwise, an integrity error is signaled.

To restore the state at start-up time, Hugo reads the special state key and unseals the returned value. The unsealed counter value is compared against the current counter value. If



(a) original application

(b) same application using Hugo

Figure 4.1: Architectural diagram showing how Hugo is incorporated into existing applications.

the unsealed counter is lower than the current counter, a rollback error is signaled. Otherwise, application start-up can continue normally.

4.2 Design of Hugo

This section will go into more detail about the design of the different components and algorithms in Hugo.

Counter interface

Many different secure counters are supported, see §4.4. Each counter backend exposes a straightforward interface as follows:

- AllocateCounter$\rightarrow I$
- ReadCount($I_{counter}$) $\rightarrow i$
- BumpCount($I_{counter}$)

AllocateCounter allocates a new counter for this enclave and returns a globally unique identifier for this particular counter. This identifier will later be used to securely associate a state with this particular counter. If the same identifier was previously for another counter, the new counter must be initialized with at least the maximum value of the old counter. In the case of a TPM, the identifier might for example be the combination of this TPM's public endorsement key and the index handle.

ReadCount simply reads the current value of the counter identified by the argument. BumpCount increments the counter value by one and blocks until the counter is actually incremented.

Key-value store interface

The revertible key-value storage backend provides the following standard interface:

- Read(k) $\rightarrow v$
- Write(k, v)
- ClearAll

Write will write the value v at location k, replacing an old value at the same location, if any. The special value $\emptyset$ can be written to erase a location. Read will read the value stored at location k, or the special value $\emptyset$ if the location is empty. A correctly functioning key value store will return the value that was stored there by the most recent Write call, but the security of our scheme does not depend on this. ClearAll will erase the entire store as if $\emptyset$ was written to every location.

Strawman implementation

Using the building blocks introduced in the previous section we now describe a simplified version of the main Hugo algorithm, as described in §4.1. This simplified version, Algorithm 1, is completely secure, but does not support efficient concurrency or the rollback budget. For clarity, the case of $\emptyset$ reads and writes has been omitted. An **atomic** block means that in case of an interruption of service (crash, power off, etc.) and an honest revertible key-value store, the whole block completed successfully, or nothing changed. The security of our scheme does not depend on the atomicity.

The algorithm works as follows. A map S of all the keys in the revertible key-value store and the hash of the values is kept in enclave memory. When a value is written to the store using WriteFresh, its hash is computed and stored or updated in the map. When a value is retrieved from the store using ReadFresh, its hash is checked against the one in the map. If the hashes don't match, an error is signaled. This provides partial rollback protection: the untrusted key-value store cannot compose an inconsistent state by piecing together various writes.

Additionally, when a value is written, after updating the map, the map is serialized and sealed to the enclave identity, counter identity and the next counter value. The serialized and sealed map c along with the counter identity and the next counter value is stored in the revertible key-value store at the special key Σ. This will allow us to recover S after a failure in the future. Then, the counter is incremented.

Restoring S follows a similar procedure in reverse. First, the expected counter identity and counter value, and c are read from key Σ. If the expected counter value is lower than the actual value of that counter, an error is signaled. Otherwise, c is unsealed (failure to unseal will also signal an error) and stored in S. Further ReadFresh and WriteFresh calls can proceed as normal.

We already established that partial rollbacks are not possible. Also note that a violation of the atomicity by the untrusted key-value store is treated the same as a partial rollback.

Full state rollbacks are prevented by the counter. If the secure service is interrupted before line 25, no state has changed and there is therefore no rollback. If the secure service is interrupted during the atomic operation, the system must revert to the state as it was before the operation was started. If it doesn't, the state will be inconsistent and some future invocations of ReadFresh will fail. If the secure service is interrupted just after the atomic operation has succeeded or before the counter is physically incremented, Restore will accept both the current and previous states. An adversary is able to generate many new states with the same next counter value by crashing the application just before line 31. This is acceptable since any output associated with the new states has not yet been outputted. Once the counter increment succeeds, Restore will accept only the current state.

S can be efficiently implemented using a Merkle tree so that each WriteFresh need only output $O(\log n)$ data each state update instead of $O(n)$.

Algorithm 1 Simplified rollback protection

```
1: global I_counter
2: global S
3: procedure Reset
4:     I_counter ←AllocateCounter
5:     i ←GetCount
6:     S ← ∅
7:     c ←Seal(I_enclave||I_counter||i, S)
8:     begin atomic
9:         ClearAll
10:        Write(Σ, (c, I_counter, i))
11:    end atomic
12: procedure Restore
13:    c, I_counter, i ←Read(Σ)
14:    i_0 ←ReadCount(I_counter)
15:    assert i ≥ i_0
16:    S ←Unseal(I_enclave||I_counter||i, c)
17:    for 1 . . . (i − i_0) do
18:        BumpCount
19: procedure ReadFresh(k) → v
20:    v ←Read(k)
21:    assert H(v) = S[k]
22:    return v
23: procedure WriteFresh(k, v)
24:    i ←ReadCount(I_counter)+1
25:    S[k] ← H(v)
26:    c ←Seal(I_enclave||I_counter||i, S)
27:    begin atomic
28:        Write(k, v)
29:        Write(Σ, (c, I_counter, i))
30:    end atomic
31:    BumpCount(I_counter)
```

Algorithm 2 Rollback budget implementation

*changes compared to Algorithm 1 indicated with **

```
 1: global actual                                         *
 2: procedure ResetBudget
 3:     Reset
 4:     actual ← 0                                        *
 5: procedure RestoreBudget
 6:     Restore
 7:     actual ← 0                                        *
 8: procedure WriteBudget(k, v, budget)
 9:     i ← ReadCount(I_counter)+1
10:     S[k] ← H(v)
11:     c ← Seal(I_enclave||I_counter||i, S)
12:     begin atomic
13:         Write(k, v)
14:         Write(Σ, (c, I_counter, i))
15:     end atomic
16:     actual ← actual + Cost(v)                         *
17:     if budget ≤ actual then                           *
18:         actual ← 0                                    *
19:         BumpCount(I_counter)                          *
```

Rollback budget

The rollback budget allows limited full rollback of the whole state to a previous self-consistent state. Each write has a certain cost associated with it. As long as the cumulative cost has not exceeded the budget, the counter will not be increased during a write. The augmented algorithm is shown in Algorithm 2.

It is possible to obtain increased rollback protection by increasing the counter opportunistically whenever new data is written. By calling BumpCount asynchronously, execution can continue while the state is being rollback protected as fast as possible. During this time more data can be collected which will be linked to the future counter value. Only once a write would exceed the budget will the write block until the counter increment is complete.

This results in a pipeline where state gets rollback protected as soon as possible, see Figure 4.2. As soon as a counter increase is initiated ("commit"), writes for the following counter state will be collected ("collect") while the increase is pending. Once the increase is complete, another increase is immediately issued and the cycles continues.

Note that the maximum time a request might block is now $2\Delta t$ (where Δt is the time it takes from issuing the counter increase request to its confirmation), whereas it is Δt without the pipeline. The throughput is higher though, depending on the budget. As such, different workloads will require different solutions.

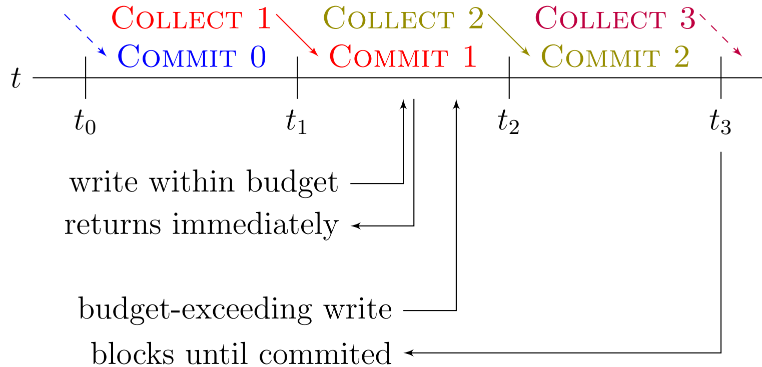


Figure 4.2: Collect-commit pipeline with two types of request shown.

Crash-evidence

Because using the rollback budget provides weaker security guarantees, it's important to at least detect when the secure service was interrupted during a state update. This is performed simply and reliably by keeping track of clean shutdowns.

Upon RESET, $S[\Gamma]$ is set to UNCLEAN. A new function SHUTDOWN will set $S[\Gamma] \leftarrow$ CLEAN, and store S with no rollback budget, after which the program must exit immediately. Upon RESTORE, after loading S, if $i > i_0$, $S[\Gamma]$ is set to UNCLEAN. Then $S[\Gamma]$ is set to UNCLEAN and the old value is returned as a clean shutdown indicator.

A secure service may rely on an external trusted party to implement rate-limiting of unclean shutdowns.

Concurrency

Lastly, we add concurrency to our scheme as shown in Algorithm 3. Writes of Σ need to happen in a serialized fashion to ensure the state is consistent and can be recovered fully. We augment our definition of atomicity with three additional properties. An **atomic** block means that

1. in case of an interruption of service (crash, power off, etc.), the whole block completed successfully, or nothing changed;
2. all computations in the current thread work on the global state as it was at the beginning of the block;
3. concurrent threads either see all changes or none;
4. consecutive atomic blocks observe strict ordering (they are pipelined).

We require that each k is either read in a shared fashion or written exclusively. This can for example easily be accomplished with a read-write lock per k. Writes and reads to different keys can happen concurrently.

Algorithm 3 Concurrent rollback protection

*changes compared to Algorithm 2 indicated with **

```
 1: global current                                              *
 2: procedure ResetConcurrent
 3:     ResetBudget
 4:     current ← ReadCount(I_counter)                          *
 5: procedure RestoreBudget
 6:     RestoreBudget
 7:     current ← ReadCount(I_counter)                          *
Require: No concurrent writes of k                              *
 8: procedure ReadConcurrent(k)
 9:     return ReadFresh(k)
Require: No concurrent reads or writes of k                     *
10: procedure WriteConcurrent(k, v,budget)
11:     bump ← false                                            *
12:     begin atomic                                            *
13:         actual ← actual + Cost(v)                           *
14:         i ← current +1                                      *
15:         if budget ≤ actual then                             *
16:             actual ← 0                                      *
17:             current ← i                                     *
18:             bump ← true                                     *
19:     end atomic                                              *
20:     begin atomic                                            *
21:         S[k] ← H(v)
22:         c ←Seal(I_enclave||I_counter||i, S)
23:     end atomic                                              *
24:     begin atomic                                            *
25:         Write(k, v)
26:         Write(Σ, (c, I_counter, i))
27:     end atomic                                              *
28:     begin atomic                                            *
29:         if bump then                                        *
30:             BumpCount(I_counter)                            *
31:     end atomic                                              *
```

4.3 Obliviousness

Existing secure enclave implementations such as Intel SGX do not protect against side-channel attacks [33]. Previous work has shown that much information can be inferred from observing the behavior of an SGX process from the Operating System [70]. Developers wishing to defend against such attacks to maintain confidentiality should employ oblivious algorithms [43]. If an application uses a key-value store in an oblivious manner, Hugo can be used to provide rollback protection for that application while remaining oblivious.

To prove our point, we analyze Hugo under a variant of the program-counter model [47]. Under the PC model, it is assumed that an adversary can obtain a *trace* of every instruction the program executes. In our model, the trace includes both instructions and data, and the full program is known to the adversary. In other words, the adversary can observe all data Hugo operates on. In our analysis we show that an adversary will not learn more information than they would learn from the oblivious program without Hugo.

The information "leaked" by any program using a key-value storage backend without Hugo is shown in Traces 1 and 2. On the left is the code and on the right is the trace

Trace 1 Unprotected read

	Code	Trace
1:	**procedure** READ(k) $\to v$	"READ", k, v

Trace 2 Unprotected read

	Code	Trace
1:	**procedure** WRITE(k, v)	"WRITE", k, v

information for that code. An oblivious program will need to make sure that any trace consisting of a sequence of these traces is oblivious, meaning no information can be learned from any k, v or the ordering of those calls.

In Hugo, the trace for READ is replaced by the trace for READCONCURRENT in Trace 3. As shown, the sequence of operations in the trace is dependent only on values that were

Trace 3 Hugo's read

	Code	Trace
1:	**procedure** READCONCURRENT(k) $\to v$	
2:	$v \leftarrow$READ(k)	"READ", k, v
3:	**assert** $H(v) = S[k]$	"**assert**", k, v, S
4:	**return** v	v

already known to the adversary per Trace 1. The only additional information being made known to the adversary is S. Since S is determined solely by the sequence of writes in the past, the adversary has full knowledge of the information contained in S already.

Hugo replaces the trace for WRITE by the trace for WRITECONCURRENT in Trace 4. Conditional parts of the trace are denoted by '?'. Information that was not present in Trace 2

Trace 4 Hugo's write

```
 1: procedure WriteConcurrent(k, v, budget)
 2:     bump ← false                                              bump
 3:     begin atomic
 4:         actual ← actual + Cost(v)                             actual, Cost(v)
 5:         i ← current +1                                        i, current
 6:         if budget ≤ actual then                               budget, actual
 7:             actual ← 0                                        ?actual
 8:             current ← i                                       ?current, i
 9:             bump ← true                                       ?bump
10:     end atomic
11:     begin atomic
12:         S[k] ← H(v)                                           k, v, S
13:         c ←Seal(I_enclave||I_counter||i, S)
                                                                  c, I_enclave, I_counter, i, S
14:     end atomic
15:     begin atomic
16:         Write(k, v)                                           "Write", k, v
17:         Write(Σ, (c, I_counter, i))
                                                                  "Write", "Σ", c, I_counter, i
18:     end atomic
19:     begin atomic
20:         if bump then                                          bump
21:             BumpCount(I_counter)                              ?"BumpCount"
22:     end atomic
```

is 'bump', 'actual', 'current', 'budget', COST(v), i, c, $I_{enclave}$, $I_{counter}$, "Σ", "BUMPCOUNT" and a second "WRITE":

- $I_{enclave}$, $I_{counter}$, "Σ" are all public information.
- c depends only on public information, i, and S.
- S is public, as described for READCONCURRENT.
- "BUMPCOUNT", 'bump', 'actual, 'current', and i depend only on 'budget', v (known), and READCOUNT, per Figure 4.3. That Figure also covers the conditional parts of the trace.
- READCOUNT is the value of the counter upon initialization and is publicly known.
- The second "WRITE" always happens after the first "WRITE", regardless of any other potentially unknown information.

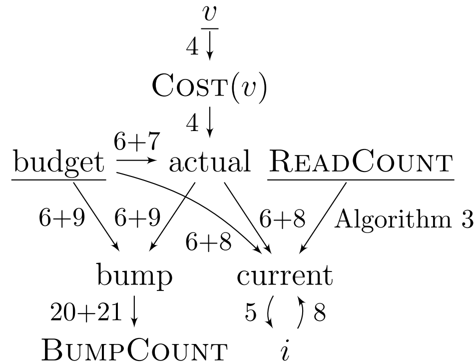


Figure 4.3: Dependency graph between various variables in Trace 4. Each arrow indicates a dependency and is annotated with the line number where the dependency comes from. Inputs are underlined. Dependencies on constants are not shown, since no information can be revealed through such a dependency.

This leaves 'budget' and $\textsc{Cost}(v)$, which to maintain obliviousness, must not reveal any additional information. A useful Cost function that does not reveal additional information is $\textsc{Cost} : v \mapsto |v|$. The developer must choose 'budget' independently from k and v or risk disclosing additional information.

4.4 Counter backends

TPM 2.0

A common secure counter available on many platforms are the non-volatile counters in Trusted Platform Modules (TPMs). However, unlike in TXT-based PMA designs, the communication channel between the TPM and a secure enclave is not trusted. This presents a problem, since a malicious operating system must not be able to present a fake TPM counter to the secure enclave.

The TPM specification version 2.0 includes the ability to setup a secure channel with a TPM [65], similar to TLS. Embedded in the TPM is an endorsement key, a private key that was generated at manufacturing time. The manufacturer is also a Certification Authority and signs the endorsement public key (EK), this is called the endorsement certificate. Consider a client C communicating with a TPM. C can request the EK and certificate, and verify that the certificate is signed by a well-known and trusted TPM manufacturer. Compare this process to how TLS certificates are used. C can then proceed to use the EK in a key-exchange protocol. For an RSA EK, this means encrypting some key with the EK, for an EC EK, this means performing EC Diffie-Hellman. The exchanged symmetric key is then used to encrypt and MAC TPM commands. Again, compare this process to how TLS works.

Using this secure channel, a secure enclave using a TPM counter can be assured that it is talking to a genuine TPM. The TPM EK and counter handle together are the *counter identity* as specified in §4.2. The TPM certificate validation code and the list of valid CAs should be included in the enclave measurement for remote attestation, so that trust in the TPM is conveyed transitively. This way, third parties can also be assured of the proper and secure functioning of the counter.

TPM multiplexing

TPMs may support only 8 distinct counters [64]. In order to support many secure enclaves, it is possible to write a separate counter enclave. The sole responsibility of this enclave is to provide a consistent counter interface to other (client) enclaves, backed by a single counter on the TPM. The counter enclave will keep a separate counter per client enclave and store the state of all counters persistently using the TPM storage backend. The rollback budgets specified by the clients could be consolidated by the counter enclave as well. The client enclaves will require attestation of the counter enclave. This approach is similar to the *memory management module* proposed in Memoir [50].

Trusted counter service

Instead of using the TPM backend with the counter enclave, the counter enclave could instead run on trusted hardware directly (e.g. TrInc [40]) and communicate with client enclaves over the network. The amount of network traffic would be very small compared to a solution where all state would be stored on a trusted network appliance. Again, the client enclaves will require some sort of attestation of the remote counter enclave.

Client cookie

In a cloud computing setting, a secure enclave service may be running in an untrusted environment. However, the users of such a service might be trusted since they have a vested interest in the correct and continued operation of the service. In such cases, the counter can be stored on the client in a “cookie”.

When the service sends a response to a client, it includes a sealed counter value. The client stores this value, the cookie, indefinitely. The next time the client makes a request, it includes the cookie. Now the service can unseal the counter value and verify that the state it’s processing on is no older than the last time a request for this client was completed.

Note that this presents a weaker guarantee than using a continuously available counter such as a TPM. After a user A finishes a request, a user B might perform several requests that change the service state. User A will not be able to determine if any of the changes made by user B were rolled back. This guarantee may nonetheless be sufficient for certain scenarios. Cookies may also be shared out-of-band with other clients of the same service to convey state information for shared resources.

4.5 Evaluation

Performance modeling

Three metrics determine the performance of parallel request handling. ρ is the time it takes to process a single request, including network and TLS handshakes and data en- and decryption. It is expected that this parameter is CPU-limited due to the amount of cryptography involved. τ is the time it takes to increment the counter. N is the *parallel factor*, the number of requests that can be handled in parallel. For CPU-bound processes, N should be between 1 and 2 times the number of hyper-threaded cores on a system.

Throughput is a measure of the volume of requests that can be handled. If there's no rollback protection, this is just a matter of utilizing the available resources fully, handling N requests taking ρ each in parallel. With serialized continuity schemes, each request must be completed before the next one can be handled, so there is no opportunity for parallelism. In addition to that, each request must also bear the cost τ of increasing the counter, even though no processing is happening at that time. Hugo *can* handle multiple requests in parallel. Even though individual budget-exceeding requests must wait for counter increments, other requests can be serviced at that time, and throughput is not limited by τ.

Latency is the time it takes to handle a single request. If there's no rollback protection, this is simply the baseline ρ. Serialized continuity schemes incur a cost of *at least* τ. In practice, serialized schemes incur τ for *every input*, not just every request. An interactive session-establishment protocol such as TLS might require multiple inputs per request. Therefore the $\rho + \tau$ is a very optimistic lower bound on the total latency of serialized schemes. If a request does not exceed Hugo's rollback budget, no extra time is required beyond just servicing the request. If a request does exceed the budget, it will have to wait up to τ until the current collect cycle is complete,[2] and then another τ for the commit cycle.

Table 4.1 summarizes the idealized throughput and latency of no rollback protection, existing serialized schemes, and Hugo. Hugo does not decrease throughput at all compared

Table 4.1: Comparison of Hugo's throughput and latency versus existing systems.

	E(Throughput)	E(Latency)
No rollback protection	$\frac{N}{\rho}$	ρ
Serialized continuity	$\frac{1}{\rho+\tau}$	$\rho + \tau$
Hugo (within budget)	$\frac{N}{\rho}$	ρ
Hugo (exceeding budget)	$\frac{N}{\rho}$	$\rho + 1\frac{1}{2}\tau$

[2] If the request rate is sufficiently high that collects/commits are happening continuously, the arrival time of a request and the windowing of the collect cycle are independent. Therefore, the amount of time left in a collect cycle when a request arrives is uniformly distributed within the cycle, and the expected time to wait until the end of the collect cycle is $\frac{1}{2}\tau$.

to a system with no rollback protection, except due to increased synchronization overhead affecting N. If requests never exceed the rollback budget, Hugo similarly does not increase latency. Even with budget-exceeding requests, a carefully adjusted budget likely results in an average latency below that of serialized schemes.

HTTPS file storage service

To evaluate Hugo, we developed a web file storage service that runs on Intel SGX. The service is exposed to the outside world as a TLS server endpoint that is terminated inside the secure enclave. The storage service implements a REST API for uploading files (PUT), retrieving files (GET), listing files and deleting files (DELETE). Files users upload are encrypted with a key derived from their password. The secure enclave design guarantees that the service can not be surreptitiously modified to log user credentials. Users can therefore be assured of the confidentiality and integrity of the data they store with this service.

We implemented 4 versions of this service:

No rollback protection This version has the security properties described above, but does not provide freshness. A user is guaranteed to see the consistent state of their files as it was at some point in the past, but not necessarily the latest state.

Serialized continuity This version adds simulated PMA-like strict state continuity by incrementing a counter for every request (GET or PUT). As mentioned above, this is an optimistic approximation since a real PMA would treat a single HTTPS request as many inputs.

Hugo (high budget) This is an implementation of a concurrent system with a rollback budget and a collect-commit pipeline as described in §4.2. All requests are considered to be within the rollback budget.

Hugo (low budget) This version considers all PUT and DELETE requests to exceed the rollback budget. Other requests are within the budget.

The service is written mostly in memory- and type-safe Rust. Only the off-the-shelf mbedTLS library is written in C. The storage backend used is Redis, while the counter is provided by a TPM 2.0. A small untrusted driver program provides the secure enclave with TCP and Redis connectivity, and relays TPM commands.

Our test setup hosts the service on a dual-core Intel Core i5-6200U with hyper-threading and a 128MB EPC size.[3] All requests where performed over a 1Gbit network connection. The TPM used is an Intel Platform Trust Technology TPM.

We evaluated the no rollback protection version and both configurations of Hugo by performing various mixes of PUT and GET requests with between 1 and 8 concurrent requests. Since the serialized continuity version does not support concurrency and every request is

[3]This is the maximum EPC size supported by current hardware.

handled similarly, we only tested it with 1 concurrent PUT request. One test consists of 30 seconds of repeatedly requesting the appropriate ratio of request types for a single configuration, while measuring the time each request takes.

We first consider latency. The single concurrent request, 100% GET test of Hugo (high budget) has an average latency of $\rho = 60$ms. The single concurrent request, 100% PUT test of serialized continuity has an average latency of 190ms, therefore $\tau = 130$ms. Figure 4.4 shows the latency for all of our tests. Since the ratio between GET and PUT requests has limited impact on the latency for the no rollback protection version and Hugo (high budget), those measurements are shown as a range instead of individual points. Of note is the high latency and high variance in latency for low-budget high-concurrency mostly-PUT benchmarks. Our performance model predicted slightly higher latency for the low-budget benchmarks. We think the excess latency is caused by contention issues and is not an inherent problem with our design: (a) the benchmarks are run on a 2-core CPU, which obviously severely limits concurrency, and (b) because SGX does not have special synchronization primitives, all synchronization is performed using busy-loops. Both these issues can easily be resolved. Note that at 50/50 PUT/GET, 3 concurrent threads are still as fast as a single serialized continuity thread. The same holds for 5 concurrent threads at 25/75 PUT/GET.

In our setup, we obtain maximum throughput at 5 concurrent requests. Figure 4.5 shows the throughput for each of our implementations for 5 threads. The throughput of high-budget Hugo is within 4% of a system with no rollback protection. As predicted by the latency figures, the throughput for low-budget Hugo is significantly lower for high PUT/GET ratios. There are two reasons real systems based on Hugo will perform better: (a) the latency issues can be resolved as discussed earlier, and (b) the rollback budget can be tuned and not every request will exceed the budget in real systems. Every configuration of Hugo always outperforms serialized continuity in terms of throughput.

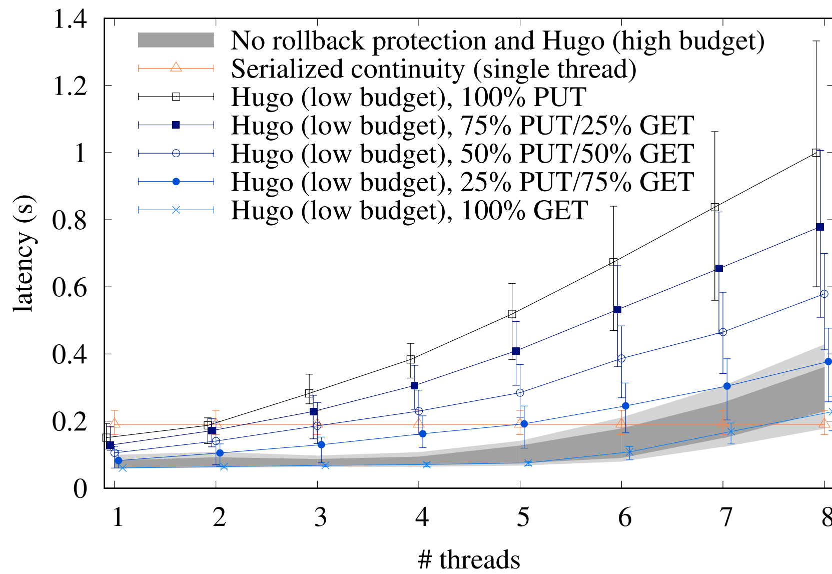


Figure 4.4: Latency of requests for different configurations. The points indicate the average latency, and the error bars 25th and 75th percentiles. The X-values are slightly offset from their integer value to make the error bars easier to read. The dark shaded area is the range of average request latencies of all configurations of the no rollback protection version and Hugo (high budget). The light shaded area the is 25th through 75th percentile range of the same.

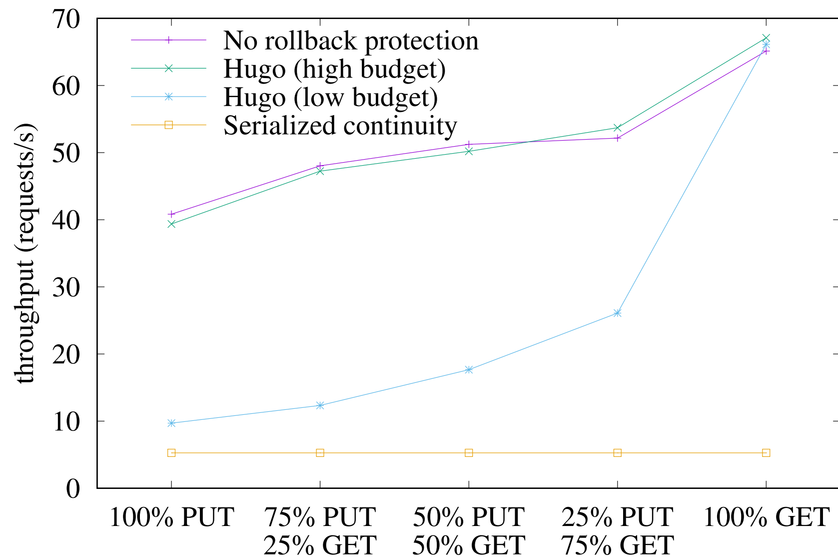


Figure 4.5: Throughput of requests for different implementations. The number of concurrent requests is 5 (but serialized continuity is limited at 1).

Chapter 5

Attestation Transparency

This chapter details our approach to use transparency to deter unauthorized access. Services store a hash of the software that they are running in a public audit log, and (conceptually) clients verify that the server is running code that matches a hash in the public audit log. This means that insiders cannot make undetectable changes to the server-side code; at least in principle, any such changes can be detected through examination of the public audit log. One challenge is how to ensure that these servers can be used from legacy clients, such as existing web browsers. We show how to achieve this goal by building on Certificate Transparency.

This chapter makes the following contributions:

Policy and update mechanism Although we describe services as unalterable, in fact, updates will often be desirable to fix bugs, improve usability or security or add functionality. Also, instances of a secure service should not be locked to a single platform. We demonstrate how these goals can be achieved, without user-visible downtime for updates, and without compromising security or privacy. Finally, secure services may—under policy control specified by a user—be authorized to collaborate with other user secure services while preserving the security promises; we show how this can be done as well.

Legacy client support In a manner similar to the way Certificate Transparency protects against CAs secretly issuing bad certificates, Attestation Transparency protects against service providers secretly changing the services they provide. This transparency provides a public record linking domain names to service implementations. Today's TLS clients immediately reap the benefits of the transparency framework, except in some cases when they are the victim of a targeted attack involving misissued certificates.

5.1 Overview

While we cannot absolutely prevent insiders from violating security, the transparency mechanism guarantees that such violations will be *publicly* detectable.

Users of a secure service must be able to verify that their client is connected to a specific service that is known to provide those security properties. Legacy clients must be supported: users must be able to obtain most of the security benefits without installing special software. Beyond legacy clients there must be an incremental deployment path. Performance loss compared to insecure services should be minimal. Services must also be updateable, and the security properties must be maintained in the update process.

When the service is first created, it publishes its TLS public key in an attested statement proving that the enclave was launched with a certain code and that that code generated the key. Legacy clients not built with Attestation Transparency in mind won't be able to verify these attestations, but the key idea of our system is that another party can do so on their behalf. Because the attestations are public, anyone can check what code the service will run, that the code is secure, that it will never reveal its TLS private key, and that it protects itself adequately from malicious insiders. This allows word to spread through out-of-band channels that the service is trustworthy. For instance, an expert might inspect the code published by `good.com` and determine that it is trustworthy and will never leak its TLS private key; inspect the attestation and TLS certificate and determine that the TLS keypair was generated by this enclave, and the public part is in the TLS certificate; and then spread the word that `good.com` can be trusted.

Of course, a malicious insider at `good.com` could always take down the secure service and replace it with malicious code, running outside an enclave. An attestation-aware client could detect this (because the attestation will change), but a legacy client could not. However, this attack is detectable. To mount such an attack, the insider would need to generate a TLS keypair and get a new certificate issued for it (because legacy clients expect to connect over TLS), and hand the new private key to the malicious code. This is detectable because it triggers issuance of a new certificate for `good.com`. In particular, we use Certificate Transparency to detect issuance of new certificates. In our design, secure services publicly commit to always publish a new attestation any time they update the service or obtain a new certificate. Thus, issuance of a new certificate without a corresponding published attestation indicates an attack or error. Crucially, because all of this information is public, anyone can monitor the published information and detect these situations, providing transparency.

Because our design focuses on transparency about what code the service will run, we call it *Attestation Transparency*. It extends Certificate Transparency to allow publishing these independently-auditable attestations. Legacy clients can rely on Certificate Transparency to ensure that attacks will be publicly detectable, while future attestation-aware clients can verify attestations themselves.

A diagram of the entire system is shown in Figure 5.1.

Policy model

To verify that a secure service will act "as promised", the user must verify that (a) the service code correctly implements the intended behavior and (b) no other program on the same computer will be able to interfere with the operation of the service code or observe its

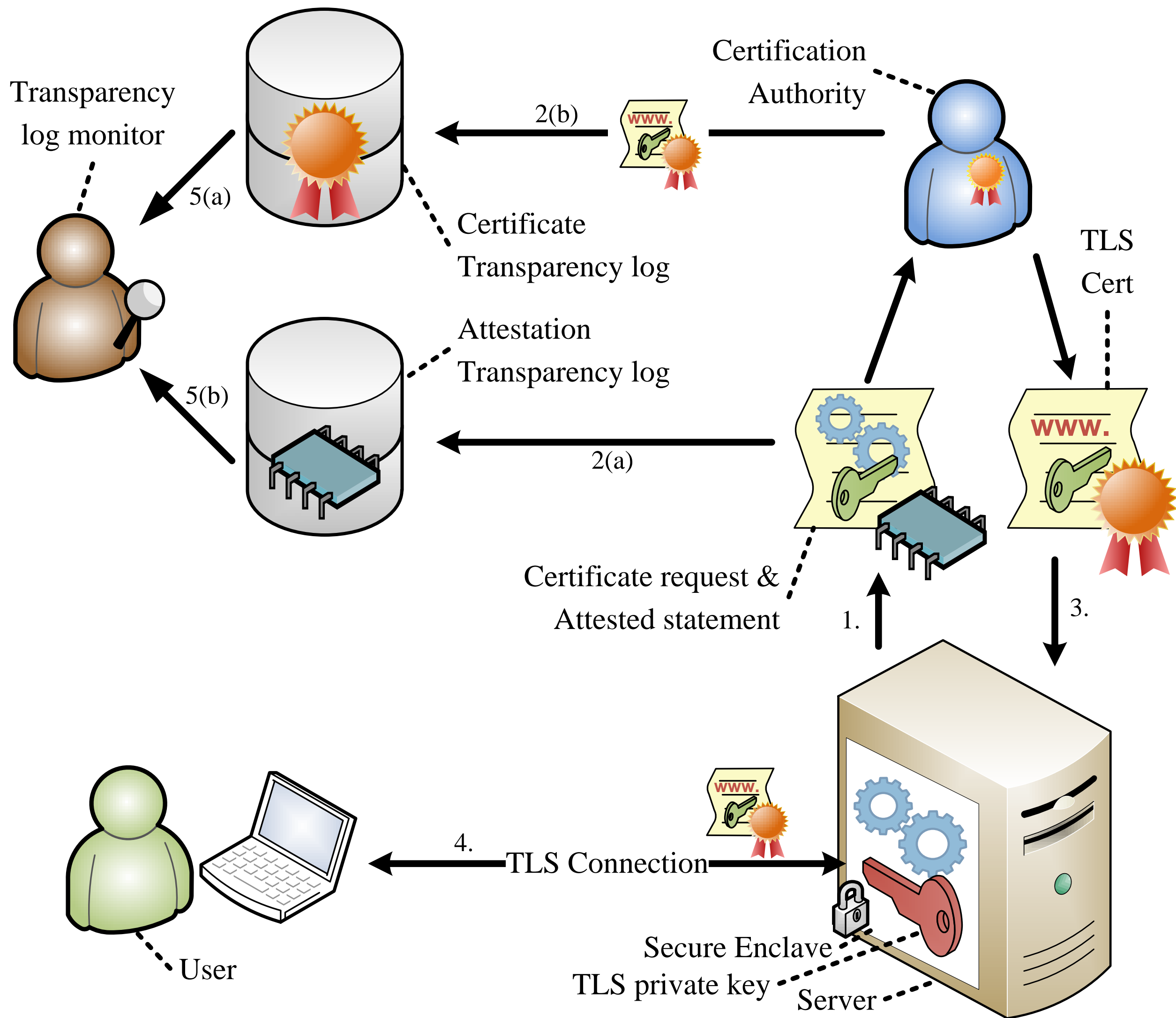


Figure 5.1: Overview of Attestation Transparency. (1) The *secure service* emits the certificate request and attested statement. (2) The attested statement and certificate are submitted to the *Attestation and Certificate Transparency logs*. (3) The secure service receives the certificate produced by the *CA*. (4) The *user* can now establish a regular TLS connection with the secure service. (5) The *transparency log monitor* independently monitors the transparency logs for possible violations.

memory. The mechanisms described below allow a service to prove to a user what service code will execute on the server enclave. Thus, in principle, a user could examine all of the code and convince themselves that it will act "as expected" and will provide all the desired security guarantees. However, in practice few users will be able to do this: code analysis is expensive and beyond the capabilities of most end users.

To address this challenge, we provide several flexible mechanisms to enable users to verify that the service code will meet their needs (§5.5). One option is that the user may rely on a cryptographically signed statement from the secure service developer naming both the service identity and the (user comprehensible) promised behavior. Since the developer can produce the implementation corresponding to the identity, it can be verified by third parties or used as a basis for legal recourse in the case the service does not, in fact, conform to the promised behavior. Alternatively, a user can rely on either an authority (for example, in the case of an enterprise service, the enterprise "IT" department) or an auditor to decide which services are trustworthy. The authority cryptographically signs a statement representing that the service code conforms with expected behavior in all respects. These can be securely and automatically checked if so designated. Alternatively, a user may rely on a set of reviewers of the secure service code who cryptographically sign such statements. Reviewers might have different motivations, some altruistic and some self-serving. Further, a user may employ policy rules to automatically determine if the specified behaviors are adequate. For example, the user may insist that the EFF examine the service and certify that it meets a designated privacy policy, *and* that a consumer agency or a product reviewer also sign a conformance statement, *and* that the developer be one of a named set of developers that the user feels comfortable with. There are other alternatives that ensure compliance with the user's needs and relieve the user of the need to conduct extensive reviews themselves.

Usually, a user will use the same policy model to decide whether an update meets those same specified needs and, if it does, whether user data accessible to previous versions can be made available to subsequent versions.

5.2 Client verification of secure services

The previous section described how to construct a secure service. This section will explain how a client connecting to such a service can verify that service.

Consider the simple scenario in which the server sends an attestation $A(I_{\text{enclave}})$ to the client as part of establishing a secure channel. The client will need to verify both the attestation and the identity. As a straw-man proposal, envision a service provider distributing a client program that includes a fixed identity and can verify attestations expected for a particular service. This would require users to install a different client per service they want to use. Additionally, since the identity is fixed in the client, service updates would also require a client update.

A more general client could contain the logic to be able to verify all possible attestation mechanisms, as well as maintain a list of all acceptable identities. Done naively, this would

be worse logistically, since now a new client needs to be distributed for every service update.

5.3 Attestation Transparency

Instead of creating this new verification mechanism that clients would need to implement, we build a verification mechanism on top of an existing mechanism that clients already know how to use: Public-Key Infrastructure. Under our proposed scheme, all the client needs to do to trust the secure service is verify the TLS server certificate using standard existing methods.

Our scheme, called *Attestation Transparency*, is an extension of the Certificate Transparency framework [38]. Remember that in our unalterable secure service model, demonstrating posession of K_{server}^{-1} by an entity implies that it is a secure service instance. The core idea of *Attestation Transparency* is that a secure service provider publishes (once) in the Attestation Transparency log an attested statement $A(I_{\text{enclave}}, K_{\text{server}})$. With this, they announce to the world that an entity possessing K_{server}^{-1} is an instance of I_{enclave}. The secure service provider also obtains a valid TLS server certificate for K_{server} through normal means and inputs it into the enclave. The certificate binds a Common Name (CN) to the key K_{server}, and the published attested statement binds that to the identity I_{enclave}. When a client establishes a TLS connection with the enclave, it verifies the certificate and the enclave uses its K_{server}^{-1} to sign the key exchange, after which the client has established a secure channel with the enclave. The whole process is depicted in Figure 5.1 on page 41.

An *Attestation Transparency monitor* performs a similar function to a Certificate Transparency monitor. The AT monitor continuously watches the CT logs for certificates issued to the CN identifying the secure service it is interested in. Once a new certificate with public key K is logged, the monitor checks the AT logs to see if any attested statements $A(I, K)$ with that same key K exist. If such an attested statement does exist, the monitor checks whether the identity I is valid for that service. If the identity is invalid, or no attested statement was found in the log, the monitor raises the alarm.

To prevent spamming, an AT log might require proof of existence of a valid certificate in the CT logs before accepting statements for submission. As such, there can be a short period of time where a certificate will exist in the CT logs without a corresponding statement in the AT logs. Monitors will need to take this into account and choose an appropriate wait time (e.g. two maximum merge delays) before raising the alarm. This wait time is the maximum time during which clients could be vulnerable to attack, before it could be noticed.

5.4 Incremental deployment — logs

While from the previous description it sounds like the AT log is separate from the CT log, this is not necessarily the case. Instead, attested statements can be included in a certificate

as a X.509 Certificate extension.[1] The secure service can output the attested statement in the requested extensions section of its certificate signing request. As Certificate Transparency is already being deployed, this means Attestation Transparency does not require any new infrastructure. We propose minor changes to CT to support AT, along with an incremental deployment path towards a smoother process in the future.

Fake attestation certificates

Currently the only data that can be included in the CT logs are certificates and precertificates.[2] To prevent spam, the only certificates accepted in the logs are those signed by a known CA.

In order to publish attested statements in the CT logs, we propose that CT logs also accept (pre)certificates from an 'Attestation Authority' (AA). This is a fake Certification Authority that only issues pre-certificates and is not trusted by regular TLS clients. The AA follows a simple procedure: it takes as input a certificate, a Signed Certificate Timestamp and a certificate signing request that includes a statement as an extension. The AA verifies the certificate and SCT and it verifies the CSR includes the same public key. It will then issue a precertificate with the same Subject Name and public key, including the statement extension and a pointer to the real certificate. The AA will only issue one precertificate per real certificate.

Attested statement log entries

An alternate first step in the deployment process is to move the Attestation Authority's responsibilities into the CT log server. This requires a change in the CT specification to add a new entry type for attested statements. The inputs for the submission procedure will be the same, the verification and spam protection measures will be the same, only the output will be an attested statement-type entry in the CT log as well as an SCT for this entry.

While this setup increases the functionality and complexity of the CT log, it reduces the logistical complexity compared to using an Attestation Authority.

Certificate extensions

It would be much more convenient to just include the attested statement as an extension in the actual valid end-entity TLS certificate. This would eliminate the need for any changes to the current CT system. It would also solve the issue of a potential delay between the appearance of the certificate and the attested statement in the logs.

It is currently practically infeasible to obtain certificates with such an extension. We contacted a total of 9 subsidiaries of the largest 6 Certification Authorities (Comodo, Symantec

[1] We have allocated OID arc 1.3.6.1.4.1.4995.1000.4.1 for this purpose.

[2] Precertificates are similar to regular certificates, conveying the same information. However, they are constructed in such a way that they can't be used in place of a regular certificate.

Group, Go Daddy Group, GlobalSign, DigiCert, StartCom) to see if they would issue certificates with this extension. Of the CAs we contacted, 5 did not respond to our inquiry or did not understand the request, 3 were unable to provide such certificates, and 1 was unsure whether it would be possible, but if it was, it would cost an additional US$5,000. We considered (ab)using an existing extension, but were unable to find a suitable one for the type of data we'd want to include.

We encourage CAs to support Attestation Transparency extensions in the future.

5.5 Validating enclave identities

The previous discussion depends on being able to determine what is a valid enclave identity. This is mostly a matter of policy, and as such we present a mechanism that supports different policies. For each service, some entity or a group of entities—known as the *policy administrator*—is in charge of verifying the policy for an enclave identity. The policy administrator maintains a private key for each service policy. After verification, the policy administrator signs the enclave code indicating that the policy was met. For example, the EFF could establish a service that audits code for privacy violations and certify complying code by signing it. These compliance certificates can be used as an automated or semi-automated mechanism by client software to determine whether it trusts the code.

When a system runs such a signed enclave, it will issue attestation statements of the form $A(I_{\text{signer}} : I_{\text{enclave}}, K_{\text{server}})$. An AT monitor will maintain a list of policy administrators it trusts for a specific CN. Now, the monitor need not itself verify the enclave identity in an AT log entry, it can instead rely on a valid signature from the policy administrator.

Handling updates

This mechanism also enables code updates to services by having an old version of the service check the policy for the new version. This check is embedded in the code for the old service, and has undergone the same vetting process as the rest of the code. The whole process for updating from an old service S_1 to a new service S_2 is as follows.

$$D \xrightarrow{S_2} P \tag{5.1}$$

When a developer D is ready to update their service, they will send their binary S_2 and optionally documenting materials to the policy administrator P.

$$P \xrightarrow{\text{Sig}_P(I_2)} D \tag{5.2}$$

The administrator will verify that the new code meets the policy and sign it.

$$SP : S_2 \xrightarrow{\text{CSR}(K), A(P:I_2,K)} CA, AT \tag{5.3}$$

The service provider SP will launch the signed enclave S_2, which will output a certificate signing request including the attested statement. The service provider submits the CSR to a CA and the attested statement will be published to the AT logs.

$$CA \xrightarrow{\text{Cert}_{CA}(K)} SP : S_2, CT \tag{5.4}$$

CA will sign a certificate. The certificate will be submitted to the CT logs. With this publication, the policy administrator has announced to the world that I_{enclave} conforms to the policy established by I_{signer}. The service provider inputs the certificate into the signed enclave and launches the service. AT monitors will see the new service with the new certificate and can use the CT/AT log to verify that everything is in order.

$$S_2 \xleftrightarrow{\text{secure channel}} S_1 \tag{5.5}$$

S_2 establishes a mutually authenticated secure channel with S_1. Both sides must verify the code identity of the other side through attestation, as well as check the Certificate Transparency proofs for their keys. Doing both validates the service code *and* ensures that there is a public record for this particular service instance.

$$S_2 \xrightarrow{\text{Proof}(A(P:I_2,K)\in AT)} S_1 \tag{5.6}$$

S_2 provides S_1—which is configured to accept policy statements from policy administrator P—with proof that an attestation $A(P : I_2, K)$ appears in the AT logs.

$$S_2 \xleftarrow{\text{sealed data}} S_1 \tag{5.7}$$

S_1 will subsequently transfer its sealed data to S_2.

The update process can be performed without downtime for the users. Users can keep using the old version of the service as long as its certificate is still valid. Once the new certificate has been obtained and the required publications in the Transparency logs have been made for the updated service, it can start accepting connections. From then on, clients will see the new certificate and Transparency log proofs, indicating that they are now using the updated service.

5.6 Enclave policies

A policy shall at least require that the TLS private key will not be leaked and that updates shall be considered valid only when accompanied with a proof that they appear in the Attestation Transparency logs. While a policy administrator may issue a signed policy statement erroneously, the statement will be ineffective until published. Once published, others can hold the policy administrator accountable. It is also important that an entity controlling a Certificate Transparency log signing key must not also be an entity controlling

a policy signing key. Such an entity would be able to issue signed policy statements and obtain a signed 'proof of inclusion' from the log without actually publishing the statement.

Policies can cover a variety of use cases from most transparent to not transparent. Care must be taken when a single party has a fair amount of control over what would be considered a secure service under their policy. Such constructions should always be paired with the ability for independent entities to verify their claims *post facto* using transparency. We propose the following policies:

Open-source reproducible builds

The software developer publishes their source code publicly with a mechanism for reproducible builds. The same developer doubles as the policy administrator and builds the binary and signs it before handing it off to the service provider.

Independent audit

The software developer hands their source code to an independent auditor. The auditor vets the secure service and describes the security properties the service has in a policy. It will sign the binary and publish the policy. When, later, the developer submits an updated version of the software, the auditor checks whether it meets the security requirements per the established policy.

As an extension to this scheme, an independent auditor could maintain several standard policies. For example they might have a 'secure IMAP service' policy. Anyone will be able to write software that adheres to the policy, and the auditor can verify and sign all such software. This effectively creates an interoperable system of secure IMAP services, where data can be transferred from one service to the other while maintaining the security properties.

Self-published with legal obligations

The software developer hands their source code to a publisher. The publisher builds the binary and signs it before handing it off to the service provider. The publisher also promises (e.g. by incorporating a clause in their terms and conditions) that enclaves they sign exhibit certain properties.

Enterprise-local audit

An enterprise might maintain a set of policies for secure services it runs internally. They can have an internal audit team that vets service updates. This way they can have the benefits of protection from insider attacks as well as local control.

5.7 Incremental deployment — clients

We present an incremental deployment path for Attestation Transparency that makes it immediately useful for today's clients while improving security guarantees for future clients.

Current clients

Initially, clients without Certificate Transparency support will benefit from the existence of the CT/AT ecosystem, as independent entities can monitor the published certificates and statements. However, there are no guarantees for such clients and targeted attacks are possible. While the CT logs might include a valid certificate for some domain, a client without CT support can be presented with a valid certificate that does not appear in the logs, and the client would be none the wiser.

Clients supporting Certificate Transparency

Once clients support Certificate Transparency, a process which is already in motion, they will get most of the benefits of Attestation Transparency as well. Suppose a service has subscribed to our secure service paradigm, promising to publish in the AT log in conjunction with the CT log. Then, by checking the Signed Certificate Timestamps when setting up a connection, a client can be sure that the server published its attestation if it has one. A user still needs to rely on manual verification or word-of-mouth to know whether a particular service at a particular domain is in fact a *secure service*.

Clients supporting Attestation Transparency

A client that can check the attested statements will be able to indicate to the user that it is in fact connected to a *secure service*.

Clients supporting remote attestation

Clients supporting remote attestation get even stronger guarantees than those just supporting Certificate Transparency. With remote attestation, a client can verify that the server they are connected to is actually running inside a secure enclave. This is helpful in case a server's TLS private key got leaked inadvertently. A third party could run a modified service with the TLS private key thus impersonating the secure service under the previous three mechanisms. When using remote attestation directly, this third party could not produce a correct attestation if the service were modified.

Chapter 6

Discussion and open research questions

6.1 Limitations

The research in this dissertation does not address availability questions at all. Denial of service is a valid attack that an adversary might perform. Worse, destruction of user data is also possible. In order to get the cloud environment closer to the desktop model, these issues need to be resolved.

In order for a user to be able to fully trust a 'secure web application' as defined in the previous section, they need to know that the data they see or the input they provide is handled securely. The current web hardly provides such mechanisms. JavaScript and CSS on a page can arbitrarily change page elements to re-order information, capture user input, or even read cross-origin data [60]. More research effort is needed to provide the user with a secure and trustworthy user interface on the web.

The security of our system relies on an adversary not being able to break in to the secure enclave. Even if the hardware is infallible—which it isn't—a simple software bug could leak sensitive information or provide an attacker with code execution capabilities. Bugs are exacerbated by being completely transparent about the code running on a machine. The transparency makes it much easier for an attacker to automate exploitation of known vulnerabilities. We propose using only safe languages such as Rust to write secure services, but even then there is no guarantee against compiler bugs or developer errors. Further guarantees could be obtained by using formal methods.

While SGX in theory provides good isolation, in practice it might have security flaws. In addition, SGX does not aim to protect against side-channel attacks [32]. The operating system is in an excellent position to mount side-channel attacks as well as Iago attacks [15]. SGX also has software components that are critical to its security, which might be more easily compromised than the hardware [19]. Compromise of SGX on the system running the secure service provides an attacker with access similar to that of directly compromising

the secure service. But, even if SGX is broken, future systems might provide better secure enclave functionality that can be used for the secure service design in this paper.

6.2 Adoption

The growth of remote attestation technology usage is hampered by a fractured ecosystem. Vendors need to come together to develop standards around remote attestation. In §5.4 we proposed standardizing the Attestation Transparency certificate extension. Additionally, the validation of remote attestations needs to be standardized. At the very least, there needs to be a registry of algorithms and a directory of vendor public key material. There should also be a standard way to open a secure channel with a third party authenticated by their remote attestation.

This dissertation proposes writing *new* applications as secure services. While the secure enclave interface abstraction and the Rust language should largely prevent application developers from doing insecure things, it is not clear that those means are sufficient. It is not clear what skills are required of an application developer to keep their application secure, and similarly what toolsets and engineering aids a developer could benefit from. This could be investigated.

Previous Trusted Computing approaches have not seen much practical use. This could be due to substantial security issues with the required hardware [69]. Our scheme differs from most approaches in that the required hardware and software support is only needed on the server side. A single entity can decide to adopt our approach and make it happen without being dependent on their customer's hardware or software choices.

Some service providers may be reluctant to adopt our scheme. However, we believe there is motivation to strongly consider it. The ITIF has predicted that the U.S. cloud computing industry could lose up to $35 billion by 2016 due to loss of trust in the post-Snowden era [29]. Forrester Research has predicted that losses could eventually be up to $180 billion [59]. Our scheme provides a mechanism that partially addresses these trust concerns.

Even so, people—enterprises, developers, individuals—need to be aware of the security and privacy risks of cloud computing, before they will even contemplate mitigating those risks. It is the opinion of this author that there is a lack of this awareness, and especially so among individuals and small-business owners. Even developers might not be cognizant of the risks or they might be too complacent towards their employers to fully realize them. Anno 2016, people put their faith in systems that are completely opaque, but unlike Kafka's K. they lack the mistrust to ask critical questions about those systems. The only benefactors are large corporations who are able to collect massive amounts of information about everything and everyone with little to no oversight. This problem will only be exacerbated by the rise of the "Internet of Things."

The people or their governments must act to demand their digital freedom and privacy. However, I doubt many people will be enticed to act from reading this dissertation. The best I can hope for is to impart some wisdom on a stray academic or developer, and I will do so

by quoting the wisdom of others. Philip Rogaway wrote an excellent piece on cryptographic science [52] which you should read in its entirety if you work in computer security. I think the main takeaway of the essay is that you should "regard ordinary people as those whose needs you ultimately aim to satisfy." In your work, are you empowering the people or taking away their control? Taking away control can be very subtle. If you take something that people can do very well on their own computer but are instead positing it as a centralized cloud service with no alternatives, you have taken away their control.

Another thing Rogaway touches on in his essay is the improper balance between certain types of cryptography research. I'm convinced that this balance is also missing in the tech sector as a whole. Former Facebook engineer Jeffrey Hammerbacher said "The best minds of my generation are thinking about how to make people click ads. That sucks." [66] Tying this back to the previous quote, if you are in fact making people click on ads you are definitely not empowering them. Similarly, former Google engineer Mikey Dickerson said "We allocate our resources to the point where we have thousands of engineers working on things like picture-sharing apps, when we've already got dozens of picture-sharing apps." [21] He is right and there are real problems in the world in need of solving. You don't have to work on solving the many problems I posed in this chapter, but please, whatever you do, work on something relevant.

Chapter 7

Conclusion

These are the security properties and practicality goals defined in the introduction:

S1a. *Handle user data securely (privacy).*

S1b. *Handle user data securely (integrity).*

S1c. *Handle user data securely (freshness).*

S2. *Protect against insider attacks.*

S3. *Service operation verifiable by clients.*

P1. *Practical performance.*

P2. *Incremental deployment with legacy client support.*

P3. *Support for software updates.*

The different pieces of this dissertation—secure services, Hugo and Attestation Transparency—together achieves all these.

We built secure services (§3) using secure enclaves (§2.3), achieving privacy (**S1a**), integrity (**S1b**), protection against insider attacks (**S2**) and performance (**P1**). This brings to clients the benefits of the cloud—including scalability, availability, elasticity, maintainability—while guarding against principal attacks (e.g. from insiders) that make cloud usage worrisome.

Hugo, a system for rollback protection (§4), provides freshness (**S1c**). It provides 'plug-in' style protection against rollback attacks for long-running services running inside secure enclaves. Hugo is concurrent, preserves obliviousness of the application and supports a variety of trusted counter backends. We introduce the notion of the *rollback budget*, providing a way to quantitatively specify a trade-off between performance and rollback protection. Hugo has excellent performance, achieving up to 13× the throughput and 50% of the latency of existing schemes.

Remote attestation (§2.3) provides verifiable operation (**S3**) while Attestation Transparency (§5) enables legacy and future clients (**P2**) as well as software updates (**P3**) through its policy mechanism. We have presented a system enabling flexible policy options to let users meaningfully choose security properties. Policies can be established by anyone, including software developers, auditors, independent organizations and communities. The policies are enforced through a compulsory transparency mechanism that brings malicious intent to light. This deters bad actors since they can be easily identified for purposes of legal action or reputation damage.

We have extended the certificate transparency model to code, providing a technical mechanism for users to rely on the security principal which ultimately ensures security properties—code. In addition, we provide flexible trust models that allow any user to meaningfully adduce behavior guarantees from actual implementations. We demonstrate that resulting systems can be nearly as efficient and scalable as existing services and provide strong protection from mischievous providers, foreign governments and sloppy cloud data center operations.

All proposed mechanisms include incremental deployment paths which make our techniques usable now for present-day clients, whereas future deployment will increase security guarantees.

In conclusion, we have presented a flexible, practical mechanism to build secure Internet services. Combined, the design of secure services, Hugo, and Attestation Transparency provide a *trustworthy cloud computing* solution that can be deployed today.

Bibliography

[1] Comodo, DigiNotar Attacks Expose Crumbling Foundation of CA System. ThreatPost, 2011.

[2] Devdatta Akhawe, Francois Marier, Frederik Braun, and Joel Weinberger. Subresource Integrity. W3C working draft, W3C, July 2015.

[3] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege Separation in HTML5 Applications. In *21st USENIX Security Symposium*, pages 429–444. USENIX, August 2012.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, 1997.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[7] Jethro G. Beekman. secserv. `https://jbeekman.nl/projects/secserv`, 2016.

[8] Jethro G. Beekman. sgx-utils. `https://jbeekman.nl/projects/sgx-utils`, 2016.

[9] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography (PKC)*, 2006.

[10] Sören Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Client-controlled cryptography-as-a-service in the cloud. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 19–36, 2013.

[11] Rick Boivie and Peter Williams. SecureBlue++: CPU support for secure execution. Technical report, IBM Research Report, 2013.

[12] Kirk Brannock, Prashant Dewan, Frank McKeen, and Uday Savagaonkar. Providing a Safe Execution Environment. *Intel Technology Journal*, 13(2):36–51, 2009.

[13] Shakeel Butt, H. Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 253–264, 2012.

[14] David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[15] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264. ACM, 2013.

[16] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing*, pages 7–16, 2006.

[17] Yanpei Chen, Vern Paxson, and Randy H. Katz. What's new about cloud computing security? Technical Report UCB/EECS-2010-5, EECS Department, University of California, Berkeley, 2010.

[18] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[19] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://eprint.iacr.org/2016/086`.

[20] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[21] Mikey Dickerson. One year after healthcare.gov: Where are we now? Keynote, Velocity Conference. `https://www.youtube.com/watch?v=7Vc8sxhy2I4`, September 2014.

[22] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[23] Ed Felten. A court order is an insider attack, 2013.

[24] Bill Gates. Subject: Trustworthy computing. `https://www.wired.com/2002/01/bill-gates-trustworthy-computing/`, 2002.

[25] Shay Gueron. Intel advanced encryption standard (AES) new instructions set. Intel Corporation white paper, 2012.

[26] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 165–181. USENIX Association, October 2014.

[27] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[28] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2005.

[29] Information Technology and Innovation Foundation. How much will PRISM cost the U.S. cloud computing industry?, 2013.

[30] Intel Corporation. Intel Software Guard Extensions Programming Reference, October 2014.

[31] Intel Corporation. *Intel 64 and IA-32 Architectures, Software Developer's Manual*, volume 2C: Instruction Set Reference, chapter 5: Safer Mode Extensions Reference. 2015.

[32] Intel Corporation. Intel Software Guard Extensions Enclave Writer's Guide, 2015.

[33] Intel Corporation. Software guard extensions developer guide, 2016.

[34] Balachandra Reddy Kandukuri, Ramakrishna Paturi V, and Atanu Rakshit. Cloud security issues. In *Proceedings of the 6th IEEE International Conference on Services Computing (SCC)*, 2009.

[35] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016.

[36] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[37] Beom Heyn Kim and David Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[38] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

[39] Nate Lawson. Final post on Javascript crypto, 2010.

[40] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*.

[41] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[42] Min Li, Wanyu Zang, Kun Bai, Meng Yu, and Peng Liu. Mycloud: Supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 59–68, 2013.

[43] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[44] John L. Manferdelli, Tom Roeder, and Fred B. Schneider. The CloudProxy Tao for trusted computing. Technical Report UCB/EECS-2013-135, EECS Department, University of California, Berkeley, 2013. Source code available at `https://github.com/jlmucb/cloudproxy`.

[45] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys)*, 2008.

[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[47] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology (ICISC)*, 2005.

[48] The Open Group. *SOA Source Book*, chapter Service Oriented Architecture. 2011.

[49] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *31st IEEE Symposium on Security and Privacy*, pages 414–429, 2010.

[50] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, 2011.

[51] Thomas Ptacek. Javascript Cryptography Considered Harmful, 2011.

[52] Phillip Rogaway. The moral character of cryptographic work. Cryptology ePrint Archive, Report 2015/1162, 2015.

[53] Rust programming language. `https://www.rust-lang.org/`.

[54] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), March 2011.

[55] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *36th IEEE Symposium on Security and Privacy*, pages 38–54, May 2015.

[56] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[57] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[58] Ryan Sleevi and Mark Watson. Web Cryptography API. W3C candidate recommendation, W3C, December 2014.

[59] James Staten. The cost of PRISM will be larger than ITIF projects. Forrester Research, 2013.

[60] Paul Stone. Pixel perfect timing attacks with HTML5. Presented at Black Hat USA 2013, 2013.

[61] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[62] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[63] Trusted Computing Group. TPM Main Specification, 2011.

[64] Trusted Computing Group. PC client platform, TPM profile (PTP) specification, 2015.

[65] Trusted Computing Group. Trusted platform module library. ISO/IEC 11889:2015, 2015.

[66] Ashlee Vance. This tech bubble is different. *Bloomberg Businessweek*, April 2011. `https://web.archive.org/web/20150202014230/http://www.bloomberg.com/bw/magazine/content/11_17/b4225060960537.htm`.

[67] Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pages 21–30. ACM, 2008.

[68] Edward Wobber, Martín Abadi, Mike Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Trans. Computer Systems*, 12(1):3–32, 1994.

[69] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel Trusted Execution Technology. Presented at Black Hat DC 2009, 2009.

[70] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[71] Andrew C Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, 1982.

Glossary

AT / Attestation Transparency A mechanism providing *transparency* of the code behind Internet services, making it possible to detect surreptitious service changes. (as presented in this dissertation, §5)

CA / Certification Authority A principal in *PKI* trusted with the issuance of certificates, which bind names to cryptographic public keys.

CT / Certificate Transparency [38] A mechanism providing *transparency* of certificate issuance, making it possible to detect incorrectly issued certificates.

EK / Endorsement Key A key embedded in a *TPM* which is the hardware root-of-trust for that TPM. Therefore it can be used as the *identity* of a TPM.

Hugo A framework and programming paradigm for protecting the freshness of persistent state for long-running services. (as presented in this dissertation, §4)

identity A globally unique and unambiguous identifier for a security principal. When identifying software, this could be a *measurement*.

IPC / Inter Process Communication A mechanism for different processes to exchange information.

(fully) isolated execution An execution environment for software that is isolated from other processes on the same system such that the integrity of the computation is guaranteed. This requires a break from the traditional hierarchical privilege model of modern computing in which the hypervisor or Operating System is strictly more powerful than the software it is running.

measurement A cryptographic hash of a piece of software as it would be loaded into memory or executed. Since a change in the software will yield a different hash, a measurement of software can be used as its *identity*.

PKI / Public Key Infrastructure A global infrastructure for the verifiable identification of security principals and their cryptographic public keys using certificates. *Certification Authorities* are tasked with the verification of identities of principals that wish to obtain such a certificate for identification purposes.

PMA / Protected Module Architecture Hardware capable of running limited functionality in an *isolated execution* environment.

remote attestation The ability to prove to third parties that you are running software with a particular identity on your hardware.

rollback budget The amount of data could potentially be rolled back when an adversary is trying to subvert the freshness of persistent data. (as presented in this dissertation, §4)

Rust [53] A fast and secure programming language that is capable of low-level control and high-level abstractions.

SCT / Signed Certificate Timestamp A promise by a *CT* log that it will publish a particular certificate in its logs soon.

sealing The authenticated encryption of data with an encryption key based on the *identity* of the *secure enclave* and the platform it is running on.

secure enclave A trusted computing primitive providing fully *isolated execution*, *sealing* and *remote attestation*.

secure service A service that stores and handles user data securely, preserving its confidentiality and integrity, even in the face of malicious insiders. (as presented in this dissertation, §3)

SGX / Software Guard Extensions An x86 instruction set extension by Intel, implementing *secure enclaves*.

TCB / Trusted Computing Base The extent of hardware and software that must be trusted to maintain its integrity in order to guarantee proper and secure functioning of a system.

TLS / Transport Layer Security The most widely used protocol for establish secure communication channels. Its predecessors were known as Secure Sockets Layer (SSL).

TPM / Trusted Platform Module A tamper-resistant cryptographic chip that can be used to store cryptographic secrets and other information and which can interact with the CPU of a system to securely capture the execution state of that system.

transparency Providing public information to allow the public to verify proper operation of processes. See also *Attestation Transparency* and *Certificate Transparency*.

unalterable secure service A *secure service* which can be depended upon to not change its functionality and therefore its security guarantees. (as presented in this dissertation, §3)