Secure Programming with Intel SGX and Novel Applications

Kristoffer Severinsen

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Secure Programming with Intel SGX and Novel Applications

Kristoffer Severinsen

Secure Programming with Intel SGX and Novel Applications

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Intel's *Software Guard Extensions* (SGX) is a new technology introduced in recent generations of Intel processors. SGX is supposed to be able to create a trusted execution environment for user-space software that is protected from all privileged software running on the same system. The CPU creates a protected enclave in memory for the software and guards the memory using strict access control and encryption with keys derived from secrets embedded inside the CPU.

To be able to start developing confidentiality and privacy protected applications using Intel SGX, one must first be able to reason about the security guarantees that it provides, and for this, a better understanding of the mechanisms behind the technology is needed.

First, this thesis contains a *Systematization of Knowledge* of the Intel Software Guard Extensions technology, covering the technical details of the hardware mechanisms and a practical hands-on tutorial covering the usage of the basic functionality and features.

Second, the thesis describes the design of the *Accountable Decryption protocol*, a novel protocol that can guarantee accountable decryption of user-data by using the capabilities that SGX can provide. The protocol depends on a decryption device that can be trusted to only perform decryption requests if the evidence of the decryption is observable by the user. The thesis describes the implementation of a prototype decryption device for the Accountable Decryption protocol, which can provide confidentiality and integrity guarantees by using the SGX technology. The implementation is evaluated by discussing different security aspects of the implementation.

Third, the thesis contains discussions on different solutions on how SGX can be used to protect legacy software without any modifications, and discussions on some of the security issues of the proposed solutions.

Acknowledgements

First of all, I have to thank my supervisor, Christian Johansen, for guiding my work and setting ambitious milestones and giving me the opportunity to go abroad to give a workshop presentation and attending the POST conference in Uppsala, and thank you for the opportunity to take an internship at the University of Birmingham over the summer.

I have to thank Professor Mark Ryan for inviting me to collaborate on the Accountable Decryption protocol and for the help with my accommodations in Birmingham and at the School of Computer Science. Thank you for the many good discussions we had throughout the summer.

Thanks to the OffPad project (offpad.org) for sponsoring the internship; without the funding, I would have missed the opportunity to go abroad.

Thanks to my dad, Svein, for nurturing my inquisitive mind by patiently trying to answer my constant stream of questions about life, the universe, and everything while I was growing up.

At last, I would like to thank Lisa for all her love and support throughout my studies.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years there has been a large push to move computing and services to cloud infrastructures like Google Cloud, Azure, and Amazon Elastic Compute. Using cloud infrastructure instead of building their own provides a lot of convenience to service providers. Services can be easily and cheaply deployed to scalable infrastructure with a lot of extra features like different analysis and statistics tools.

Cloud infrastructure is very convenient to service providers, however, a lot of trust has to be put in the cloud provider. Using a cloud provider for storage should be fine because the data can be encrypted before being transported to the cloud, but when using the cloud for computation, the data must often be decrypted first and loaded into the memory of a virtual server in the cloud.

The virtual machine that runs the server is only protected from other virtual machines on the same platform and is not protected in any way from the cloud provider's Virtual Machine Manager or administrator management tools. Therefore we have to trust the cloud provider with access to our confidential data if we chose to use their services, and we have to trust that their infrastructure protects our virtual machine from lateral movement from co-hosted virtual machines.

Hardware technologies like the Trusted Platform Module (TPM) and Intel's Trusted Execution Technology (TXT) can be used to create a root-of-trust in the booting process of operating systems or virtual machines by creating a trusted execution environment (TEE) for the software. These technologies can to some extent give some integrity guarantees, but cannot protect the confidentiality of the software running inside the TEE. In the mobile space, ARM's TrustZone technology can create a TEE that is able to protect the confidentiality of privileged software from the untrusted user-space programs, but not the other way around.

In an effort to give the desired confidentiality and integrity guarantees to user-space software, Intel has improved on TXT by developing new hardware security features, an extension to their x64 instruction set called the *Software Guard Extensions* (SGX). This technology is supposed to be able to create TEEs for user-space software that are protected from privileged

software.

1.1 Outline

The rest of the thesis is organized as follows:

Chapter 1 . The rest of this chapter describes the background and motivation for the work presented in this thesis. We then define our research questions and research methodology and related work, before summarizing with the main contributions of the thesis.

Chapter 2 describes the technical background of the thesis. We will describe basic principles from computer security, cryptographic primitives used throughout the thesis, and other hardware security technologies.

Chapter 3 contains a detailed description of the Intel Software Guard Extensions technology. The chapter starts by describing the theoretical part of the technology, and from Section 3.2 and out we describe the practical principles of using the technology.

Chapter 4 describes a protocol for accountable decryption of user-data with help from a trusted hardware device and an append-only request log. We first describe the motivation and the problem that we aim to solve. We then detail the protocol and design before describing the implementation of a prototype system. The chapter concludes with a discussion on the implementation and security of the proposed design.

Chapter 5 discusses different approaches for securing the messaging server used by the *Signal Messenger* application. At the end of the chapter, there is a brief discussion on the viability and security of the proposed solutions.

Chapter 6 concludes and gives some critical reflections on the results of the tutorial, and the two experiments. We end the thesis by proposing some further work on the technologies and experiments described in this thesis.

1.2 Background and Motivation

The goal of this work is to provide a better understanding of the recent technology introduced in the newer Intel processors (starting with the Skylake architecture) under the name of Secure Guard Extensions (SGX). We base our work on the rather few existing resources that describe this technology, including Intel reports, workshop papers from Intel designers, and also a few refereed research works. However, the complexity of

programming with this new security technology is rather daunting and begs for more systematic investigations. Even more so in the case of Intel SGX which was designed with application developers as intended users, thus not highly skilled in such low-level security programming.

Our goal is to give a comprehensive hands-on tutorial on how one can start programming using Intel SGX. This includes how to enable and setup the hardware and firmware aspects, as well as how to prepare a programming environment, including SDKs and IDEs. It also includes higher level concepts that SGX uses as well as explaining the various choices that the SGX designers made and what were their security concerns and their implications on the programming part. This work tries to detail various aspects ranging from high-level architectural aspects, and workflow, to intermediary level of programming API and development process, to even lower levels like explaining why and how the memory and communication are protected by the on-chip Intel SGX architecture features.

In order to make this tutorial fit the actual practical aspects, and not just extract it from the experience of others (which are currently too few), we also went into developing two novel applications where the features of Intel SGX were crucial. One application is called Accountable Decryption and was developed together with the security team in University of Birmingham lead by Prof. Mark Ryan (where the ideas and initial descriptions came from). The second application is to secure the server side of the novel and popular end-to-end encryption protocol for instant messaging called Signal. Both these applications are interesting in their own right, and have specific applications in various fields, e.g., the accountable decryption can be instantiated in the health domain or in the police/military domain.

The practical work was done in the Security Lab of Institute for Informatics of the University of Oslo and required purchasing such new Intel processors that supported SGX. This work already proved useful as a quick upstart for works of other students and staff in Birmingham that wanted to start experimenting with SGX.

Apart from the investigative and developer work done with SGX, the two applications enabled by SGX proved to be the more exciting parts of this work. It is very motivating to know that one can develop an application that removes all trust assumptions from all vendors except the processor one; i.e., developing with SGX means that one is not relying even on the operating system, nor on secondary hardware like the TPM, nor on a specific boot sequence and correct drivers/platforms, nor on hypervisors in the clouds. Instead, on any platform that provides Intel SGX, our applications can run with very high-security guarantees.

In particular, the accountable decryption application is a general framework that could be instantiated in various highly motivating specific software. This is even more motivating in the current age of mass

surveillance and privacy infringements, as this is one of the main goals of accountable decryption. We detail several such specific possible implementations of accountable decryption later.

1.3 Problem Statement

The theoretical background and development process of the SGX technology is very complex and difficult to get started with. To develop a secure application using SGX the developer should have a firm understanding of the technology and what security guarantees that SGX can provide to their software. We want to create a comprehensive guide to the technology, including the theoretical background of SGX and a hands-on tutorial on how to develop software using the technology. The first research question thus is:

1. (a) What are the theoretical underpinnings of the SGX technology and what security guarantees can SGX provide?
   (b) What are the hardware and software requirements for developing and deploying software that uses SGX, and how can we program secure software using the technology?

Because the SGX technology is still quite new, there are not that many applications that make use of it. We want to explore how SGX can be utilized to create secure applications by developing an SGX application from the ground up, or by adapting existing applications to use SGX and be able to provide better security guarantees. These would provide the reader a walk-through exemplification of the usage of SGX in practice. The second research question thus is:

2. (a) How to create novel applications that makes use of the SGX technology in a meaningful way and provides security guarantees that otherwise would be difficult to achieve?
   (b) Can existing applications be adapted to make use of the SGX technology to achieve better security guarantees?

1.4 Research Methodology

Gordana Dodig-Crnkovic describes in *Scientific Methods in Computer Science* [18] three different scientific methods in computer science:

Theoretical Computer Science adhere to the traditions of Logic and Mathematics and follows the very classical methodology of building theories as logical systems with stringent definitions of objects and operations for deriving/proving theorems. Theoretical computer science seeks largely to understand the limits on computations and the power

of computational paradigms. Theoreticians also develop general approaches to solving problems [18].

Experimental Computer Science is most effective at on problems that require complex software such as the creation of software development environments, or the construction of tools to solve constrained optimization problems. The approach is largely to identify concepts that facilitate solutions to a problem and then evaluate solutions through construction of prototype systems [18].

Computer Simulation makes it possible to investigate regimes that are beyond current experimental capabilities and to study phenomena that cannot be replicated in laboratories, such as the evolution of the universe [18].

The research in this thesis adheres to the methods of experimental computer science, where we will experiment by proposing a solution to a real-world problem, and creating a prototype application using the SGX technology, and then evaluate the security of the solution. Before proposing solutions and experimenting, we must understand the underlying technology and the security guarantees that it can provide. Chapter 3 contains a systematization of knowledge regarding the SGX technology, and a hands-on tutorial with practical examples. Chapter 4 and 5 contain the experiments and the discussions of their results. Chapter 6 will conclude with the results, give some critical reflections, and hint about further work.

1.5 Related Work

The Software Guard Extensions (SGX) is a relatively new technology, only made available with the 6th generation Intel Skylake CPUs in 2015, so there is a limited amount of available research about the technology. Costan and Devadas's *Intel SGX Explained* is an excellent write up on the low-level workings of SGX, along with a very comprehensive technical background about the cryptography, and preceding hardware security technologies. A lot of the technical details about SGX in this thesis is based on their paper, however, it does not cover any practical information about how to go about writing SGX-enabled software.

There have been some notable papers [3, 7, 9, 53] describing applications of SGX. These papers are mostly concerned with adapting existing applications for use with SGX and we discuss some of their solutions in Chapter 5.

Because SGX claims quite strong security guarantees, there has also been some research [22, 39, 54] aimed at creating attacks against software using the SGX technology. We do not try to attack the prototype software given in this thesis, but we do discuss some of the possible security issues they have revealed.

1.6 Main Contributions

- In Chapter 3 we have compiled a *Systematization of Knowledge* of the Intel Software Guard Extensions technology along with a practical example of how to use some of the basic functionality of the technology to create a simple application. This answers research question 1
- In Chapter 4 we implemented a prototype of a protocol for accountable decryption by using the capabilities that SGX can provide. We also discussed the security considerations of our implementation. This answers research question 2a.
- In Chapter 5 we proposed solutions to how we could use SGX to secure the central server used by the Signal Messenger by running server application inside a secure SGX enclave. We discussed the viability and security issues with the proposed solution. This answers research question 2b.

Chapter 2

Technical Background

2.1 Security Concepts

This section covers different security principles and concepts used when describing the security of computer systems and software. The concepts are used throughout the whole thesis when describing the security of SGX technology and the proposed applications of said technology. The section will start by describing basic concepts like confidentiality, integrity, and availability to cryptographic primitives and concepts of hardware-based security.

2.1.1 Confidentiality

Confidentiality is the prevention of unauthorized disclosure of information [21]. Confidentiality can be achieved by enforcing access control to the sensitive information, or by using cryptography to make the information unreadable to unauthorized parties.

Information might not only be the content of a message, but also the fact that a message has been sent from one party to another. Messages, actions or events are *unlinkable* if an attacker cannot distinguish whether they are related or not. Hiding who is sending the messages, or engaging in the events requires another property such as *anonymity*.

Confidentiality protection guarantees that information can be transmitted over an insecure medium without an adversary being able to disclose the information.

2.1.2 Integrity

Integrity is the prevention of unauthorized modification of information [21]. This often refers to the prevention or detection of unauthorized alterations of information. Integrity can also be applied to the internal states of a computation, making sure the correct procedures are followed, ensuring the integrity of the computed information.

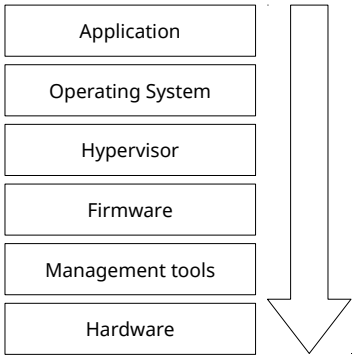

Figure 2.1: The hierarchical model of trust in cloud computing architecture. The layers below have full access to the resources of the layers above. [7]

Integrity protection in the context of communication guarantees that the receiver of a message will either, receive a message that was sent by the sender, or detects if a message, in any way has been altered.

2.1.3 Availability

2.1.4 Authentication

2.1.5 Freshness

Integrity does not protect against messages being replayed for the receiver. This is what is called a *replay attack*. Freshness protection guarantees that if multiple messages are sent, the receiver will obtain the latest message, or detect if a message is being replayed [13].

2.1.6 Trusted Computing Base

Trusted Computing Base was first termed in 1983 by the US Department of Defense in their "Orange Book" on trusted computing systems.

Definition 1 Trusted Computing Base [33]
The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of the TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g. a user's clearance) related to the security policy.

Modern computing systems rely on a hierarchical computing structure, where every layer trusts the layer below to behave correctly. The TCB of

software running in a cloud environment will not only include the security features of the software itself, but also all privileged software on the host computer. This might include the operating system, the *virtual machine manager* (VMM), also called a *hypervisor*, and the BIOS firmware. The hierarchy ensures we must implicitly trust all the levels below because the lower levels have access to every level above.

Most CPU architectures, including Intel's x86 implements *protection rings* to protect the operating system from user software. The operating system often runs in the lowest ring (ring 0 in x86), where it has all privileges over the hardware resources. User programs usually run in the highest ring (ring 3 in x86), and must request resources using *system calls*. Many CPUs also supports special "negative" protection rings used by a hypervisor; this allows the hypervisor to virtualize hardware resources for multiple virtual machines, with operating systems, all running in ring 0.

The software must also trust the underlying hardware to behave correctly. Figure 2.1 show how the hierarchical computing model creates a large TCB for a small application.

2.2 Cryptographic Primitives

Many of the security concepts presented in section 2.1 can be achieved by using cryptography; this section will describe some of the cryptographic primitives referenced throughout this paper.

2.2.1 Symmetric-key Cryptography

Symmetric-key encryption algorithms, sometimes, called secret-key, or single-key encryption algorithms are encryption schemes that are based on a shared secret between the communicating parties [45].

The shared secret-key k allows two users, Alice and Bob, to communicate securely over an insecure channel, like the Internet or a public Wi-Fi. Alice can use the key k to encrypt message x, yielding the ciphertext $y = e_k(x)$, which is transmitted to Bob. By applying the inverse operation to y, Bob is able to decrypt the ciphertext, and regain the message $x = d_k(y)$.

As seen in figure 2.2, if Alice and Bob agreed on the secret key k over a secure out-of-band channel, like by meeting in person, they are able to communicate securely over the insecure channel. An adversary, Oscar, is only able to read the ciphertext y, and without the key k, it should be *computationally infeasible* to recover the message.

By computationally infeasible we mean that the encryption algorithm should be constructed in a way that the best attack on the ciphertext should be no better than trying all possible keys in the key-space. This is what is called an *exhaustive key-search attack*. A modern symmetric encryption scheme using 128-bit keys have 2^{128} possible keys, and it is currently

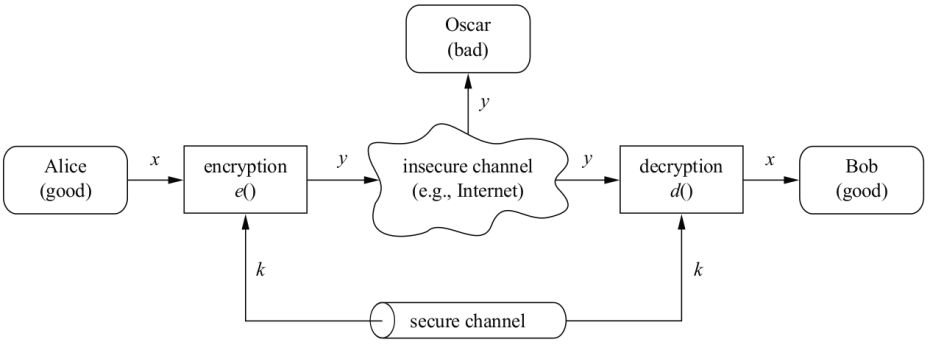

Figure 2.2: Symmetric-key cryptosystem. Alice encrypts the message x using the shared key k and sends the ciphertext message y over an insecure channel to Bob. The message y is decrypted using k to retrieve x. Oscar can only observe y. [45]

considered infeasible to recover the plaintext using exhaustive key-search attacks [45].

Symmetric-key algorithms can be split into two types, stream ciphers, and block ciphers. Stream ciphers use a key stream to encrypt every bit of the plaintext individually, while block ciphers encrypt entire blocks of plaintext bits at a time using the same key for every block. Because block ciphers use the same key to encrypt every block of data, identical blocks of data yield identical blocks of ciphertext.

There are many different modes of operation, giving different security guarantees. Using a block cipher in the way described above is called the *electronic code book* (ECB) mode.

Modes of Operation

To protect the confidentiality of a message with repeating patterns, a block cipher can be used in *cipher block chaining* (CBC) mode. By adding (XOR) each ciphertext block to the next plaintext block before encrypting it, repeating plaintext will no longer generate repeating blocks of ciphertext. By adding a random *initialization vector* (IV) to the first block of plaintext, each block of ciphertext is dependent on the previous block and the IV.

Encryption is not enough to create a secure channel; if Alice wants to be certain that (1) the encrypted message was created by Bob, and (2) nobody has tampered with the ciphertext, she would need the additional properties, *authentication*, and *integrity* for her messages.

The *Galois Counter Mode* (GCM) is an encryption mode that provides both these properties. GCM uses a block cipher as a stream cipher by encrypting the IV and a counter and using the ciphertext block as the keystream. To create a unique keystream, the counter is incremented for each block that is generated.

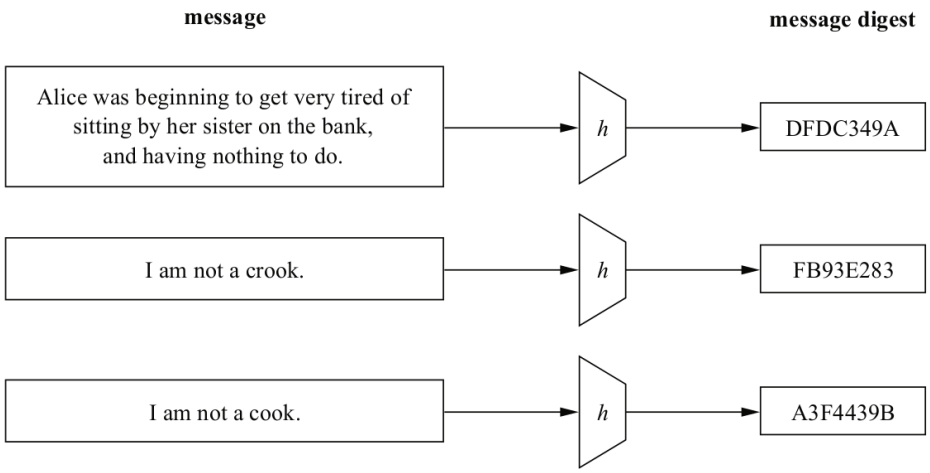

Figure 2.3: Principle of a hash function. Any message used as an input to the hash function *h* produces a fixed size message digest. By changing one character in a message, *h* will produce a very different digest. [45]

For authentication, GCM performs a chained Galois field multiplication over all ciphertext blocks to create an authentication tag. An *additional authenticated data* (AAD) parameter to the encryption can be used to authenticate some data that cannot be encrypted, like protocol versions or sequence numbers. The receiver generates the same authentication tag while decrypting the data, and if it matches the tag associated with the ciphertext, the decryption is authenticated and integrity protected [45].

2.2.2 Hash Functions

Hash functions are an important cryptographic primitive and are used to compute a short and fixed size *digest* or the *hash value* of an arbitrarily long message. The hash value can be seen as an unique representation of the message, like a fingerprint [45]. Figure 2.3 shows the input and output behavior a hash function, and how changing only one letter in the input, creates a different and unrelated output.

Hash functions play an important role in digital signatures (see 2.2.4), where, because the hash is considered a unique representation of the message, only the hash of the message needs to be signed.

This is necessary, because; (1) the operations used in digital signatures are very slow to compute compared to the operations used in symmetric-key system, (2) a digital signature is the encryption of a document when using the private key instead of the public key, making the signature as long as the message itself, and (3) it could lead to security issues, because asymmetric cryptography (see section 2.2.4) operates on blocks the same size as the key length, the blocks of the message would have to be signed individually. Without making signature blocks dependent on previous

blocks, an attacker could remove or reorder parts of the message [45].

The short hash value computed over the document is unique for the given document; there is no feasible way to create a different document with the same hash, making a signed hash cryptographically equivalent to signing the whole document.

A cryptographic hash function must have the following properties in order to be secure:

1. preimage resistance
2. second preimage resistance
3. collision resistance

Preimage resistance means the hash function must be a *one-way* function, meaning, it must be infeasible, given the hash value, to find the message given as input to the hash function.

Second preimage resistance means it should be computationally infeasible, given a message, to find a different message with the same hash value.

Collision resistance means it should be computationally infeasible to find any two messages with the same hash value. Because of the *Birthday Paradox*, this is the most difficult property to achieve and means that a hash function that generates 160-bit hash values, only provide $\sqrt{160} = 80$ bits of security.

2.2.3 Merkle Trees

The Merkle Tree [44] is a binary tree, where the leafs are hashes of a data item, and parent nodes are hashes of the concatenation of the two child nodes. The Merkle tree can be used to authenticate an individual or subsets of data items in a larger set of data items. The main building block in a Merkle tree is a cryptographic hash function, used to recursively build a binary tree of hash values over the entire set.

Given a set of data items $Y = (y_0, y_1, \ldots, y_n)$, the algorithm can authenticate an arbitrary y_n. To authenticate y_i we define the function $\mathtt{H}(i, j)$ as follows:

1. $\mathtt{H}(i, i) = \mathtt{hash}(y_i)$
2. $\mathtt{H}(i, j) = \mathtt{hash}(\mathtt{H}(i + j - 1)/2, j)\ , \mathtt{H}((i + j + 1)/2, j))$

$\mathtt{H}(i, j)$ is a function of $y_i, y_i + 1, \ldots, y_j$, and can be used to authenticate y_i through y_j. The only value needed to authenticate the whole set is the root node $\mathtt{H}(0, n)$, which is a unique representation of the entire set.

If using a normal hash function over the entire set, we would also only need the one value to authenticate the entire set, but it could not be used to authenticate a single item in the set.

Figure 2.4 shows how the value y_4 can be authenticated. The partial tree illustrated by the lines between the nodes represents the proof that y_4

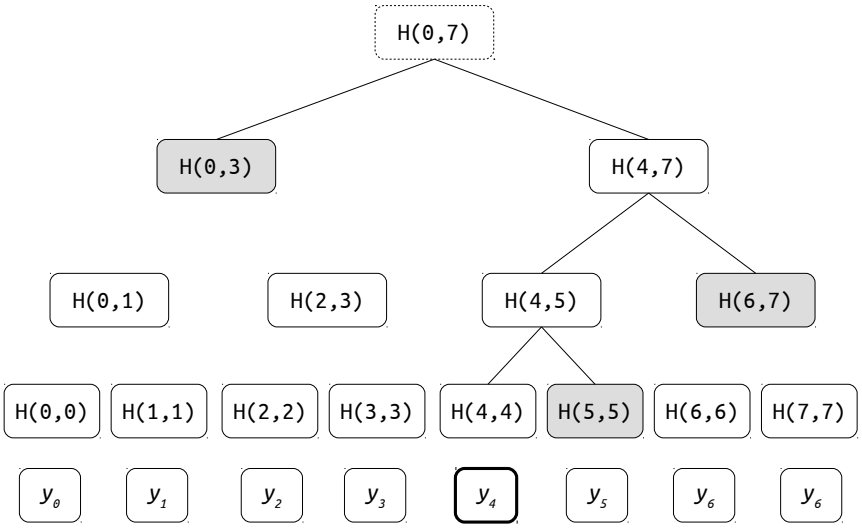

Figure 2.4: A Merkle Tree. The figure shows the authentication of a value $y_4 \in Y = (y_0, \ldots, y_7)$. The dotted node is the root tree hash and is a unique representation of all the leaf nodes (y_0 to y_7) in the tree. The drawn edges represent the sub-tree needed to authenticate y_4. The dark nodes are the values needed to create the sub-tree.

is part of $\mathtt{H}(0,7)$. By providing the hash values from the nodes in the partial tree along with the item y_4, the root node can be recomputed, proving the item is part of the set Y, represented by the root node $\mathtt{H}$. This root node can be distributed over a secure channel, or signed using a *digital signature* (see chapter 2.2.4).

The operations used to authenticate if $y_4 \in Y$:

1. $\mathtt{H}(4,4) = \mathtt{hash}(y_4)$
2. $\mathtt{H}(4,5) = \mathtt{hash}(\mathtt{H}(4,4), \mathtt{H}(5,5))$
3. $\mathtt{H}(4,7) = \mathtt{hash}(\mathtt{H}(4,5), \mathtt{H}(6,7))$
4. $\mathtt{H}(0,7) = \mathtt{hash}(\mathtt{H}(0,3), \mathtt{H}(4,7))$

Using this method, only $\log_2 n$ values are needed to create the partial Merkle tree used to authenticate a data item. About half of these values are redundant. In figure 2.4 we observe that only the values represented in the darker nodes are needed to recompute the root node $\mathtt{H}(0,7)$. $\mathtt{H}(4,4)$ is computed from the data y_4, and from this and $\mathtt{H}(5,5)$, $\mathtt{H}(4,5)$ can be computed. Finally, the computed candidate root node $\mathtt{H}(0,7)$ can be compared with the known root node $\mathtt{H}$. If $\mathtt{H}(0,7) = \mathtt{H}$, y_4 is proven to be part of the set.

The cryptographic properties provided by cryptographic hash functions (section 2.2.2) guarantees the the authentication tree actually authen-

ticates the chosen leaf. We observe that, by changing one bit in any of the data items; its computed hash value would be different, and this change would cascade to the top, and produce a different root hash.

2.2.4 Asymmetric-key Cryptography

Asymmetric-key cryptography, also called public-key cryptography is based on very different principles than symmetric-key cryptography, where the same secret key is used for encryption and decryption. In asymmetric-key algorithms, there are different keys for encryption and decryption; the encryption key, or public-key can only be used for encryption and does not need to be kept secret. The decryption key or private-key is used for decryption and must be kept secret.

While symmetric-key encryption has been used for thousands of years (like the *Caesar Cipher*), asymmetric-key cryptography is a very new concept, discovered in the 1970s. While modern symmetric-key encryption schemes like AES are very secure and very fast, they have some shortcomings compared to asymmetric-key cryptography:

The key distribution problem arises when Alice and Bob want to establish a secure channel between each other for the first time. If they use the insecure channel to exchange the secret key, the key is no longer secret, and cannot be used to create the secure channel. Alice and Bob could meet in person for the key exchange, but this would not be practical in the context of digital communications. Symmetric key distribution schemes like the *Kerberos protocol* can be used to establish a secure channel between two users that do not already share a secret, but the protocol requires that the parties have a secure channel to a central service provider.

The number of keys when keeping pair-wise keys between n parties is $\frac{n(n-1)}{2}$. In an organization with $n = 2000$ employees, each employee would need to keep $n - 1$ keys each, and the key distribution service would need to generate almost 2000000 key pairs, and distribute almost 4000000 keys.

No non-repudiation because the shared secret gives both parties the same capabilities. Alice and Bob would be able to authenticate messages between each other, but Alice would not be able to prove to a third party a that Bob had sent a given message because she could have created the message herself.

Figure 2.5 shows the basic principle of public-key encryption. Bob can generate his key k and can share the public part of the key k_{pub} with Alice. Alice encrypts her message and sends it to Bob, who is able to decrypt it using the private part of the key k_{pr}.

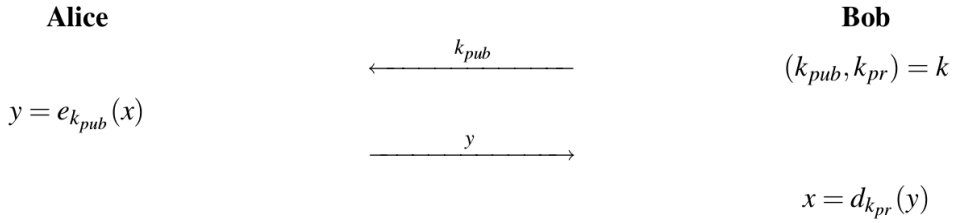

Figure 2.5: Basic protocol for public-key encryption. Bob shares is public-key k_{pub} with Alice. Alice encrypts the message x using k_{pub}. Only Bob can decrypt y using his private key k_{pri}. [45]

Definition 2 One-way function [45]
A function $f()$ is a one-way function if:

1. *$y = f(x)$ is computationally easy, and*
2. *$x = f^{-1}(y)$ is computationally infeasible.*

Unlike symmetric-key algorithms, asymmetric-key algorithms are often based on number-theoretic functions, and specifically *one-way functions*. There are two popular one-way functions used in asymmetric cryptography schemes. The first is the *integer factorization problem*, where the problem is, given two large primes, it is easy to compute the product. However, it is very difficult to factor the product. The second is the *discrete logarithm problem*, where the problem is based in group theory, and depends on the difficulty of computing discrete logarithms modulo a prime when the parameters are very large [45].

RSA

The RSA (Rivest-Shamir-Adleman) crypto scheme is based on the integer factorization problem. By representing the plaintext bitstring as an element x in the integer ring $\{0, 1, \ldots, n-1\}$, the encryption and decryption algorithms are very simple:

RSA Encryption [45]
Given the public key $(n, e) = k_{pub}$ and the plaintext x, the encryption function is:

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n$$

where $x, y \in \{0, 1, \ldots, n-1\}$

RSA Decryption [45]
Given the private key $d = k_{pr}$ and the ciphertext y, the decryption function is:

$$x = d_{k_{pr}}(y) \equiv y^d \bmod n$$

where $x, y \in \{0, 1, \ldots, n-1\}$

In practice, x, y, n, and d are very large numbers (over 1024 bits). The exponent e and d are often called the public- and private exponent and the prime number n is called the public modulus. These parameters are carefully generated to achieve the desired properties.

RSA Key Generation [45]
Output: public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$

1. Choose two large primes p and q.
2. Compute $n = p \cdot q$.
3. Compute $\Phi = (p-1)(q-1)$.
4. Select the public exponent $e \in \{0, 1, \ldots, \Phi(n) - 1\}$ such that
$$gcd(e, \Phi(n)) = 1$$
5. Compute the private exponent d such that
$$d \cdot e \equiv 1 \bmod \Phi(n)$$

Finding the large primes in step 1 is not trivial, and is usually done using probabilistic algorithms, where the primes are tested until we are reasonably sure they are in fact prime. In step 4 and 5, we can use the *Extended Euclidean Algorithm* to find a public exponent e with the required property, and at the same time finding the inverse of e, and thus also finding the private exponent:

$$d = e^{-1} \bmod \Phi(n)$$

Because RSA encryption and decryption is based on performing modular exponentiation on very large numbers (above 1024-bit), the naive approach to calculating the result by straightforward exponentiation would require around

$$2^{1024} \approx 10^{300}$$

multiplications, and would not be possible to compute. Fast exponentiation makes RSA feasible and can be achieved by the *square-and-multiply* algorithm, also called binary exponentiation [45]. The algorithm scans the bits in the exponent left to right (from most to least significant bit). For every bit in the exponent, the current result is squared. If the currently

scanned bit is a 1, a multiplication of the current result by x is executed after the squaring. The algorithm dramatically reduces the complexity of exponentiation to an average of

$$1.5 \cdot 1024 = 1536$$

multiplications, but can in some cases pose a security risk to the private key.

RSA in Practice

There are some security consideration to be able to use RSA in practice [45].

- RSA is deterministic, and thus the same plaintext will always encrypt to the same ciphertext.
- Plaintext values of 0, 1 or -1 would encrypt to itself.
- RSA is malleable, meaning an attacker can make predictable changes to the plaintext by manipulating the ciphertext. For example, by replacing the ciphertext y with $(s^e y)$, where s is some integer. The receiver would decrypt:

$$(s^e y)^d \equiv s^{ed} x^{ed} \equiv sx \bmod n$$

A modern padding scheme like *Optimal Asymmetric Encryption Padding* (OAEP) is a specified standard for use with RSA, and solves all the issues above by adding some randomness to the structure of the plaintext.

Another issue with RSA arises while decrypting the ciphertext; during the square-and-multiply algorithm, the CPU performs different operations based on the bits in the private decryption exponent d, and because these operations take a different amount of time (and power) to compute, the decryption exponent can be observed through different side-channels, like the timing behavior or the power-draw of the microprocessor. One possible countermeasure could be having all operations perform in constant-time (and power) [45].

RSA can be used for establishing a shared secret key for use with symmetric-key encryption schemes. Alice can generate a secret key, encrypt it using Bobs public-key, then transmit it to Bob. This approach is what is called a *hybrid encryption scheme*. Using RSA like this works fine, but it does not provide *forward-secrecy*, i.e., if Bobs private-key was compromised at a later time, all of their secret-keys, or session-keys, along with all of their communications could be compromised.

Diffie-Hellman Key Exchange

The Diffie-Hellman Key-Exchange (DHKE) was the first asymmetric-key scheme to be published. It is widely used in modern encryption protocols

Diffie–Hellman Key Exchange

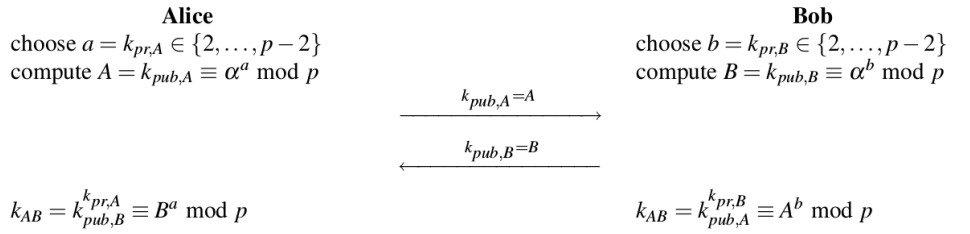

Figure 2.6: The Diffie-Hellman Key Exchange protocol. Alice and Bob can compute the shared secret k_{AB} by raising the public-key they received from the other party to their own private-key. [45]

like *Transport Layer Security* (TLS), *Internet Protocol Security* (IPSec) and *Secure Shell* (SSH) to solve the key-distribution problem.

DHKE is based on the discrete logarithm problem and the basic idea behind DHKE is that exponentiation in a finite cyclic group of order p, where p is prime, is a one-way function and that the exponentiation is commutative.

$$k = (\alpha^x)^y \equiv (\alpha^y)^x \bmod p$$

The value k is the shared secret that can be used to derive a session key between the two parties [45]. The values α and p, also called the *domain parameters*, are public parameters that both parties must know before starting the protocol. First, a large prime p is generated, then choosing an integer $\alpha \in \{2, 3, \ldots, p-2\}$, before publishing both p and α. When both Alice and Bob know the domain parameters, they can perform the DHKE protocol. In practice there are standardized domain parameters [35] that are included with common cryptographic libraries.

From figure 2.6, we observe that both Alice and Bob compute the same session-key k_{AB}, and can use this key to encrypt their communication using a fast and secure symmetric-key scheme.

DHKE can also be implemented using *elliptic-curve cryptography* (ECC), which is also based on the discrete logarithm problem. Using ECC, the keys can be much shorter, because of how the best attacks on traditional asymmetric-key (prime factorization) are not applicable to groups of elliptic-curves. Because elliptic-curve Diffie-Hellman (ECDH) uses shorter parameters, it is faster and easier to implement on power-constrained hardware devices like smart cards [45].

Using DHKE like in figure 2.6 is not safe at all, and is what is called *anonymous Diffie-Hellman*. Alice and Bob have no way of knowing if the public-keys $k_{pub,A}$ and $k_{pub,B}$ they received actually came from either Alice or Bob. If the adversary Oscar is able to intercept the protocol

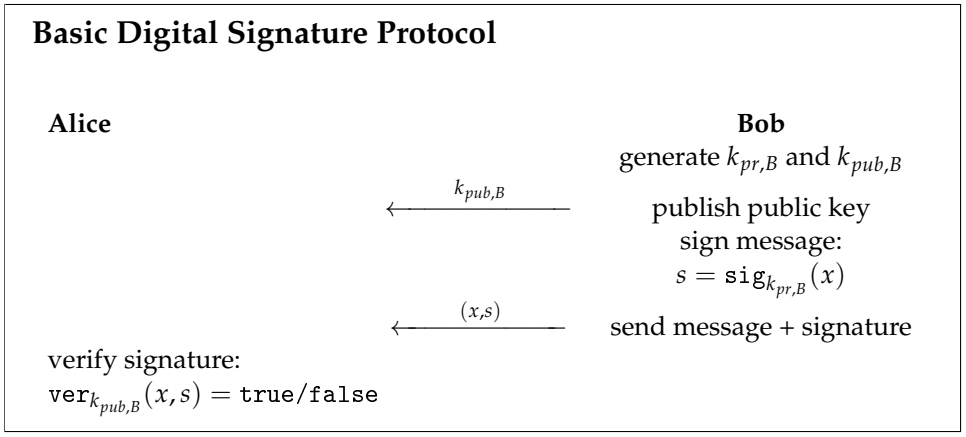

Figure 2.7: A basic Digital Signature protocol. Bob shares his public-key with Alice, who can use it to verify messages from Bob. Only the person who knows the corresponding secret-key can create a verifiable signature. The signature can be seen as a message that is encrypted by the private-key and therefore can only be decrypted by the public-key. [45]

messages between Alice and Bob, Oscar is able to perform a *man-in-the-middle* (MITM) attack on the protocol. Oscar can create is own pair of public keys, and establish secure channels both with Alice and Bob by posing as the other party in the protocol. We cannot trust the public-keys $k_{pub,A}$ or $k_{pub,B}$ if they cannot be securely linked to Alice or Bob. To achieve this link, we can use a digital signature scheme to sign the public-keys.

Digital Signatures

A digital signature, like its analog counterpart, is used to tie a document to some entity. The purpose of a digital signature is to provide message integrity (see section 2.1.2) and *message authentication*, or data origin authentication, along with *non-repudiation*. The last property is the key principle needed for digital signatures that cannot be provided by symmetric-key cryptography. If a message has been signed by the sender, the sender cannot deny creating the message.

The principle behind using asymmetric-key cryptography for digital signatures is to use the private-key to sign a message, and then the verifying party can use the corresponding public-key to verify the signature. The signature itself is a large integer that only could have been generated by the holder of the private-key. A basic digital signature protocol only needs two operations, *Sign* and *Verify*, as seen in figure 2.2.4. The sender signs the message to generate the signature, then sends the message and signature to the receiver. The receiver then uses the verify operation on the message and signature, which returns either valid or invalid.

RSA can be used to create digital signatures; the RSA Signature scheme works by encrypting the message x using the private exponent d. This

creates a "ciphertext", or signature s:

$$s = x^d \bmod n$$

The signature s can only be decrypted, or verified by the public exponent e. The mathematical relationship between the public exponent and the private exponent (described in section 2.2.4) ensures this property. The verification operation s^e turns the signature back into the original message:

$$s^e = (x^d)^e = x^{de} \equiv x \bmod n$$

The verifier then compares x' with the original message x, and if they match, the signature is verified [45].

2.3 Hardware Security

This section describes some basic concepts and technologies related to hardware-enabled security. The CPUs in modern architectures have been extended by many security features; we have already described the protection rings in section 2.1.6, used to protect the hardware resources. Other mechanisms, like *Write Protect* (WP) and *No-Execute Enable/Disable* (NXE/XD) bits are used to protect memory pages from being overwritten, and data pages from being executed [26].

2.3.1 Software Attestation

The security model of Intel SGX depends on running the trusted parts of the software in isolated containers. This model hinges on software attestation (see section 3.1.6 about how SGX implements software attestation). The software inside the container is measured by a trusted hardware module before the container starts running. The software then has the hardware module create an *attestation signature* by signing the measurement with its private key. The attestation signature could only have been signed by the tamper-resistant hardware module, and thus the *verifier* can be convinced that the software is correctly running inside an isolated container and protected by the trusted hardware [13].

2.3.2 TPM

The *Trusted Platform Module* (TPM) introduced the software attestation model. The TPM design relies on a tamper-resistant chip to hold the attestation key and to perform software attestation. The TPM relies on the software running on a system to report its own cryptographic hash to the TPM, where the TPM extends the measurement of all the reported software. The TPM expects the software in every stage of the booting process to report the hash of the software in the next stage, thus creating a measurement of all software loaded during the booting process.

A problem with the TPM's model is that different computers from different vendors are running a variety of different OS kernels, drivers and firmware, and there is no source of the expected measurements of the different software modules.

The TPM can also be used for other cryptographic operations like encryption and decryption of data using a key that never leaves the TPM chip. These operations are called *seal* and *unseal* in the TPM documentation. Combined with the software attestation, the TPM can be used to only decrypt the keys used for full-disk encryption if the measurements of the firmware and bootloader software were as expected.

Intel's *Trusted Execution Environment* is a set of hardware features in their CPUs, and is the precursor to the Software Guard Extensions. Intel TXT depends on the TPM to allow software running on a system to create a measured launch environment for launching other software. This can be used for the secure booting process described earlier, but also to start the measurement chain when launching a software container like a virtual machine.

2.3.3 ARM TrustZone

ARM TrustZone [2] is a set of hardware modules for the ARM processor architecture that can be used to partition the system's resources between a *secure world*, which hosts trusted containers, and a *normal world* which runs the normal software stack. The two worlds have independent memory spaces and different privileges. The code running in the normal world does not have access to the secure world address space, while code running in the secure world can get access to the normal world address space. The hardware enforces the partitioning, making sure no secure world resources can be accessed by normal world code without using defined API's to access the resources. Typical use cases for the *trusted execution environment* provided by ARM's TrustZone include; trusted boot, authentication, payment content protection, crypto and mobile device management[1].

[1] https://developer.arm.com/technologies/trustzone 13/10-17

Chapter 3

SGX Tutorial

3.1 Background on Intel SGX

The Intel Software Guard Extensions (SGX) [42] is a set of extensions for Intel's instruction set architecture (ISA) on their newer lines of CPUs (6th generation, and newer). SGX is meant to provide a trusted execution environment (TEE) for user-space applications; to achieve this, SGX allows applications to create a protected memory area inside its address space called an *enclave*. This protected environment enforces strict access control and transparent encryption to provide confidentiality and integrity protection, even from privileged software such as the BIOS, hypervisor or the operating system (see figure 2.1 of the trust hierarchy). The software running inside the protected enclave is cryptographically measured, and can be reported back to the client; allowing for the trusted execution of software and secure provisioning of secrets to a remote and untrusted platform, as is often required in distributed systems.

As with the TPM [4, 15] or TXT [23], the security model in SGX relies on software attestation. However, Intel SGX reduces the trust requirements from the CPU & TPM providers to just the CPU provider; and thus reduces the size of the *Trusted Computing Base* (TCB) compared to the TPM by assuming all software outside the enclave to be untrusted.

While the TPM is a discrete hardware module, soldered to the platform chipset, SGX is contained inside the CPU package. The functionality provided by SGX is mostly implemented in the CPU's microcode, but the protection from physical attacks on main memory is provided by a hardware unit inside the CPU called the *memory encryption engine* (MEE). The MME transparently decrypts and encrypts reads and writes to the protected parts of memory; this ensures that the data is only kept as plaintext when residing in the cache, inside the CPU.

This section will describe different technical aspects of the SGX architecture and programming model in some detail.

3.1.1 SGX Memory Management

In figure 3.1 we observe the different layers of abstraction used the the SGX memory model. This section will describe the memory management, layout and organization used in SGX to isolate the trusted enclave.

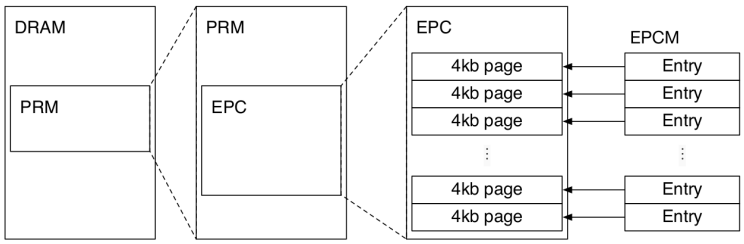

Figure 3.1: SGX memory organization. The PRM is a reserved part of the physical DRAM. The EPC is allocated inside the protected PRM. The EPC holds the pages containing enclave code and data. The EPCM contains a metadata entry for each page in the EPC. [13]

SGX Physical Memory Layout

To protect the enclave's memory from privileged software, SGX uses a part of physical memory called the *Processor Reserved Memory* (PRM), which is a continuous part of DRAM that cannot be accessed by any software code, not even by SMM (System Management Mode) code, which is at the lowest BIOS level, normally used by motherboard manufacturers to control fan speed or to emulate missing hardware.

SGX reserves a part of the PRM for the *Enclave Page Cache* (EPC), and EPC pages are only accessible when the processor is running in *Enclave mode*. Software outside an enclave cannot access the EPC, which is a key aspect of the memory isolation enforced by SGX to ensure its strict security guarantees.

The EPC hold all of the physical memory pages that backs the virtual memory of SGX enclaves. The EPC consists of 4 KB memory pages, which are managed by untrusted system software like the operating system (OS) or hypervisor by using special instructions from the extended instruction set to allocate and initialize EPC pages inside the PRM.

To keep track of the identities of running enclaves, SGX will use one EPC page per enclave to hold the *SGX Enclave Control Structure* (SECS). The SECS stores metadata about the enclave, including sensitive information like the enclave's cryptographic measurement, and is therefore only accessible to the CPU's SGX implementation. If enclave code was able to modify its own measurement, the software attestation scheme would provide no security guaranties.

Because the EPC pages are managed by the untrusted system software, SGX checks the correctness of the allocation decisions made by the system software. SGX keeps track of the allocations, and stores information about each EPC page in the *Enclave Page Cache Map* (EPCM), also inside the PRM.

Field	Bits	Description
VALID	1	0 for un-allocated EPC pages
PT	8	Page type (PT_REG, PT_SECS, PT_TCS..)
ENCLAVESECS		Identifies the enclave (enforces isolation between enclaves)
ADDRESS	48	the virtual address used to access the page
R	1	allow reads by enclave code
W	1	allow writes by enclave code
X	1	allow execution of code inside the page

Table 3.1: An Enclave Page Cache Map (EPCM) entry containing metadata about a single 4kb EPC page. [13]

Table 3.1 shows the fields stored in the EPCM. The VALID, Page Type (PT) and ENCLAVESECS fields are used to track the ownership of the pages and the ADDRESS, R,W and X fields store the information about the intended memory layout of the enclave.

Enclave Memory Layout

An enclave shares the same virtual memory layout as its host process, and the enclave is loaded like any dynamically loaded library (.dll files on Windows or .so on Linux) into the host process. The part of the enclave's virtual memory that is mapped onto EPC pages is called the *Enclave Linear Address Range* (ELRANGE). This range of virtual memory contains the initial code and data loaded into the enclave, and is not accessible to the host process since the EPC pages reside in the PRM. Read or write operations to virtual addresses in the ELRANGE from code running outside the enclave will result in undefined behavior.

The address translation from the ELRANGE to the EPC pages is handled by the OS or hypervisor; this means the CPU must protect against memory mapping attacks [66] against the enclave. When an address translation results in the physical address of an EPC page, the CPU uses the information stored in the EPCM when the page was first allocated to ensure that virtual addresses given to the address translation process matches the expected virtual address in the page's EPCM entry. The CPU will also ensure that the ELRANGE is mapped to EPC pages. This is to stop the system software from mapping the virtual memory inside the ELRANGE to regular memory, which would make the ELRANGE accessible to the system software.

The EPCM also stores access permissions (table 3.1) that override the permissions in the page table. These permissions must be defined by the enclave author (*Independent Software Vendor* (ISV)) when first allocating the

EPC page.

To enter the enclave, a hardware thread must switch context, and enter enclave-mode. The initial enclave context must be set up to transfer the control to some predefined entry points inside the enclave.

SGX uses the *Thread Control Structures* (TCS) to store information about these entry-points. The TCS must be protected from the system software, else the system could enter the enclave at arbitrary locations. Like the SECS, the TCS is stored in a dedicated EPC page, and can only be accessed by the SGX implementation.

SGX supports multiple hardware threads concurrently executing enclave code. Each of these threads must be associated to an TCS. It is the enclave author's responsibility to allocate a TCS for all the number of threads he allows to execute inside the enclave concurrently. If allowing multiple threads, he must also make sure the enclave code is thread-safe. The TCS also contains a pointer to the thread-local storage (TLS).

If enclave code is interrupted during execution, or if the enclave code needs to make calls to code outside the enclave (called a *OCALL*), the execution context of the processor is stored in a number of continuous EPC pages called the *State Save Area* (SSA).

3.1.2 Life Cycle of an Enclave

Creation

To create an enclave, the system software will call the `ECREATE` instruction, which will turn a free EPC page into the SECS for the new enclave. `ECREATE` will verify the correctness of the newly created SECS, and make sure to mark the enclave as uninitialized in the SECS.

While the SECS is uninitialized, the system software is able to use `EADD` to load the initial code and data pages into EPC pages. `EADD` will check the virtual address that the system provides, and make sure it falls within the ELRANGE. `EADD` will not succeed if the system tries to a add page to an already initialized enclave, or if trying to add an EPC page that has already been added. After adding a page, the system must call `EEXTEND` to measure the content of the newly added EPC page, and update the enclave measurement (MRENCLAVE).

In practice, the enclave author defines which software libraries should be loaded into the enclave by including them in a special meta-data file. The SGX toolchain uses these library definitions when building the enclave static library, which in turn is loaded into the EPC pages using the `EADD` instruction during enclave creation (see section 3.4 for a practical example of the enclave definition file). The enclave creation and measurement is wrapped into a high-level function call, provided by the Software development Kit (SDK) that only needs the path to the enclave static library.

Initialization

After loading the initial code and data pages into the enclave, the system software calls `EINIT` to initialize the enclave. `EINIT` requires a *INIT token* to initialize the enclave, and this token is created by an Intel provided enclave called the *Launch Enclave*. The Launch Enclave will check if the identity of the enclave author is present in a whitelist that is provided by Intel before providing an INIT token to the `EINIT` instruction. After receiving the INIT token, `EINIT` will mark the enclave as initialized in the SECS, and no more pages can be added to the enclave.

The enclave author must enroll in an Intel developer program by submitting their identity and the public part of their signing key to Intel to be added to the whitelist, and in turn be able to initiate an SGX enclave.

Teardown

When the enclave is done with its computations, the system software will deallocate EPC pages used by the enclave with the `EREMOVE` instruction. To free an EPC page, `EREMOVE` will mark the EPC page as invalid in the EPCM. The enclave teardown is complete when at last, the page containing the SECS is freed.

3.1.3 Enclave Thread Mechanisms

Any process that has mapped the EPC pages into its virtual address space is able to execute the enclave code. When a logical processor is executing enclave code, it is said to be in *enclave mode*, and can access the regular EPC pages that reside in the PRM. The enclave author decides how many TCS are allocated to the enclave, and this then decides how many threads can enter the enclave code concurrently.

Enclave Entry

To enter the enclave, and execute code inside it, the host process must use the `EENTER` instruction, which will perform a controlled jump into enclave code. Only unprivileged software running in CPU ring 3 can execute the `EENTER` instruction. `EENTER` will not perform a privilege level switch, and the logical processor will still be running in ring 3, but in enclave mode. The `EENTER` instruction is comparable to an `SYSCALL`, in that an untrusted caller wants to execute code in a protected environment. And like a `SYSCALL`, `EENTER` will store the context of the caller, so that it can be restored when the enclave code returns.

`EENTER` requires the virtual address of the TCS as input, and the instruction will then proceed to set the instruction pointer to the *entry point offset* field stored in the TCS. The entry point is defined by the enclave author, and any change to this definition would result in a different measurement when initializing the enclave. This guarantees that the

enclave code will only be invoked at well defined points, and these invocations are referred to as *ECALLS*.

In practice, the entry points are just function inside the enclave's memory range that the enclave author has explicitly defined to be an entry point into enclave code (see section 3.4 for an example).

Enclave Exit

When the logical processor is running in enclave mode, and is either done executing enclave code, or needs to access resources outside the enclave, the processor can perform a *Synchronous Enclave Exit*, where the `EEXIT` instruction will return the processor to user-space (ring 3) outside the enclave and restore the registers and context that was stored by `EENTER`.

If a hardware exception or interrupt occurs while a logical processor is executing enclave code, the system must perform an *Asynchronous Enclave Exit* (AEX) before invoking the system's exception handler. An AEX will save the enclave's execution context, and restore the state stored by `EENTER`.

The AEX will set up the system's exception handler to return to an *asynchronous exit handler* in the enclave's host process. The AEX handler will use the `ERESUME` instruction to resume the enclave that was interrupted. An AEX will backup and set all registers to a predefined value, so not to leak any secrets from the execution state.

3.1.4 Enclave Measurement

To measure the enclave, a secure hash (SHA-256) is computed over the input to the `ECREATE`, `EADD` and `EEXTEND` instructions. `EINIT` will finalize all the intermediate measurements, and store the final measurement in the MRENCLAVE field in the enclave's SECS. The enclave author must provide the exact steps used to recreate an enclave with the expected measurement. In practice this means that the client launching the enclave must use the same enclave code, the same version of the SGX build tools and drivers, and use the same enclave configurations to recreate an enclave with the same measurement.

`ECREATE` will extend the enclave measurement with the size parameters given to `ECREATE`, making sure the enclave size and SSA have the same values that the enclave author expected. Enclave attributes given during enclave creation are not part of the enclave measurements, instead they are included in the information included in the attestation process.

`EADD` will measure the ENCLAVEOFFSET field given for the given page, which is the offset of where the page is expected to be mapped into the enclave's virtual address space relative to the base address of the ELRANGE. The EPC page type and access permission fields are also measured by `EADD`. The measurement of these values ensures that the memory layout of the enclave is the same as what the enclave author had intended. The security guarantees given by SGX relies on all enclave code

and TCS pages being measured during the enclave creation. If not, the enclave code or entry points could be tampered with. The `EADD` instruction does not measure the content of the page added, only the aforementioned properties.

To extend the enclave measurement to include the content of a page, the `EEXTEND` instruction must be executed for the given page. `EEXTEND` will measure the content of EPC pages in 256-byte increments, along with the intended offset within the enclave. The design decision to load and measure pages separately likely comes from some latency constraints for SGX instructions [13].

At last, `EINIT` will finalize the measurement stored in the MRENCLAVE field and set the INIT field to true in the SECS, making it impossible to add any more pages to the enclave.

3.1.5 Enclave Identity

Software attestation relies on a cryptographic measurement of the trusted software to establish its identity. A big drawback of using only the measurement to identify the trusted software is that there is no relation between the identities of two different versions of the software. SGX supports two different identity systems for its enclaves, the first is based on the enclave measurement and the second is based on public-key certificates issued by the enclave author. The SECS is almost synonymous with an enclave's identity, and holds both of the enclave's identity types; MRENCLAVE holds the cryptographic measurement of code and data, and MRSIGNER hold the measurement of the author, or signers public-key.

The enclave cannot initially hold any secrets, because both the code and initial data are public. After an enclave has been loaded, it can generate or receive secrets to their confidentiality-protected environment. Enclaves have the ability to use their identity to derive unique keys that can be used to encrypt the data and securely store these secrets to a more stable storage medium, outside the protected environment of the enclave.

Secret Migration Process

Enclaves have the ability to derive a symmetric encryption key based on its own measurement that it can use to encrypt secrets, which can be securely stored outside the enclave. The secret is said to be *sealed* by the enclave. Because the key is derived from the measurement of the enclave's code and initial data, only the same enclave, with the same measurement can derive the key to decrypt the data.

Sealing a secret with a key derived from the enclave's measurement is the strictest from of sealing policy, and would not allow the author to update the enclave software without again having to provision the secrets to the enclave. To be able to migrate secrets between different versions of the same enclave, SGX relies on a one-level certificate hierarchy, where

the enclave author is the CA. When initializing the enclave, the EINIT instruction will use the information in the enclave certificate to populate the SECS fields that describe the enclave's *certificate-based identity*, called MRSIGNER, and is a measurement of the signers public-key material.

The secret migration process uses the EGETKEY instruction (described later in section 3.1.5) to derive a symmetric key based on the enclave's certificate based identity. The secret is encrypted using an authenticated encryption scheme (AES-GCM) and passed to the untrusted host application.

The host application passes the secret to the target enclave, who is able to derive the same symmetric key, based on the sending enclave's certificate-based identity. The target enclave *unseals* the secret, concluding the migration process.

The migration process does not guarantee freshness, making the process susceptible to replay attacks. To protect the migration process from replay attacks, SGX provides the enclave with the ability to create a persistent monotonic counter that is stored in hardware. The counter value can be incremented and can be sealed along with the secret data, making the enclave able to detect replay attacks when unsealing the secrets if the unsealed counter value and the system's counter value do not match up.

Enclave Certificates

SGX enclaves are required to have a certificate issued by the enclave author/ISV. The certificate is generated by the SGX toolchain and signed with the enclave author's private key. EINIT will verify the signature in the certificate, and then check if the the author has a licence from Intel to create enclaves. After verifying that ENCLAVEHASH in the certificate is equal to the enclave measurement (MRENCLAVE), EINIT will proceed with filling in the enclave's SECS with the information contained in the certificate.

Field	Bytes	Description
ENCLAVEHASH	32	Must be equal to enclave measurement
ISVPRODID	32	Differentiates modules signed by the same public-key
ISVSVN	32	Differentiates versions of the same module
ATTRIBUTES	16	Constrains the enclave's attributes
MODULUS	384	RSA Key modulus
EXPONENT	4	RSA key public exponent 3
SIGNATURE	384	RSA signature (3072-bit, SHA256, PKCS#1 v1.5)

Table 3.2: A subset of the enclave signature structure (SIGSTRUCT). The SIGSTRUCT contains the enclave authors signed certificate of the expected enclave measurement, enclave production ID, and enclave version number. [13]

The enclave's *certificate-based identity* is determined by the MODULUS, ISVPRODID and ISVSVN fields in the certificate structure.

A SHA-256 hash of the MODULUS field in the certificate represents

the authors identity, and is stored in the MRSIGNER field in the enclave's SECS. Different enclave modules by the same author are differentiated by a unique ID (ISVPRODID), and security updates to an enclave module is represented by incrementing the security version number (ISVSVN) of the enclave. During a security update of the enclave, the ISVSVN is used by SGX to enforce that secrets only are migrated to an target enclave with an equal or higher ISVSVN.

Enclave attributes are not covered by the MRENCLAVE measurement, and during remote attestation the attestation service can refuse to provision secrets to an enclave that was build with unacceptable attributes. During a local secret migration process, this is not practical, and to ensure that an enclave does not migrate secrets to an enclave build with unacceptable attributes, the enclave certificate includes fields describing constraints on enclave attributes.

Enclave Key Derivation

Secrets sealed by an enclave are also tied to the specific CPU that launched the enclave. The root of trust in the SGX programming model is the CPU itself, and this is achieved by deriving all cryptographic keys from not only the enclave's identity, but also, secrets embedded inside the CPU package.

The CPU has two secrets stored into its e-fuses, the first, called the *seal secret* is generated by the CPU itself, and not known to the manufacturer. The second is called the *provisioning secret*, and is manually fused into the CPU by Intel at manufacture time.

The symmetric key derivation service provided by the `EGETKEY` instruction uses the identity information from the calling enclave's SECS and the two hardware secrets fused into the CPU to derive keys. The `EGETKEY` caller must supply the instruction with a pointer to a KEYREQUEST structure, containing the information needed to derive the right key. Table 3.3 shows the different fields in the KEYREQUEST structure.

Field	Description
KEYNAME	The key type, like Seal key
KEYPOLICY	The type of identity to use
ISVSVN	the enclave SVN used in derivation
CPUSVN	SGX implementation used in derivation
ATTRIBUTEMASK	Selects enclave attributes
KEYID	Random bytes to differentiate keys

Table 3.3: The KEYREQUEST structure. The request contains information needed to derive the different types of keys. For example, a Seal key needs a policy determining what enclave can unseal the data. State could be saved using the strict MRENCLAVE policy, and a enclave update process must use the MRSIGNER policy, and the version number (SVN) of the updated enclave software. [13]

The algorithm results in the same key across CPU power cycles, and the key is impossible to derive without knowledge of the secrets stored in the CPU's e-fuses, enabling the enclave to securely seal and unseal data stored on the untrusted storage.

There are two possible key derivation policies to choose from when building the enclave. The keys will be derived from the enclave content (MRENCLAVE), or from the author's public key (MRSIGNER). Using MRENCLAVE to derive keys will result in a key that is only readable by the same enclave running on the same platform, while using MRSIGNER is a more relaxed policy that allows for enclaves by the same author to derive the same key.

There are two types of security version numbers (SVN); the author defined SVN, called the ISVSVN, and the SVN of the SGX implementation in the CPU, called CPUSVN. The target enclave's SVNs are always used in the request to derive keys, giving SGX the ability to refuse the key request if either of the the given SVNs are greater than the current enclave's SVN. This prevents secrets from being migrated to an outdated enclave version, or to a platform running an outdated SGX version.

During the secret migration process to a newer version of the enclave software, the old version must request a Seal key by filling in the KEYREQUEST structure with the ISVSVN of the new enclave software. The new enclave need only know the KEYID from the old enclave to derive the same key, and be able to unseal the secrets.

The enclave author can sign multiple enclave modules using the same signing key and must use PRODID to differentiate the different modules. PRODID is always used in the key derivation, else all enclave modules from the same author would derive the same keys when using the MRSIGNER key derivation policy.

Last, the enclave's attributes are included in the key derivation. The attributes contain some settings that could compromise enclave memory, like the DEBUG field or the *X-Feature Request Mask* (XFRM).

The XFRM is used to decide what processor extended states are allowed in the enclave. Certain processor extended states could compromise the security of the enclave [12, Section 6.7]. A debug enclave is not encrypted or access controlled, and gives no security guarantees, so the DEBUG attribute flag is always included in key derivation, making it impossible for a production enclave to migrate secrets to an insecure debug enclave.

3.1.6 SGX Software Attestation

Local Attestation

After an enclave has been initialized, it has the ability to create an *attestation report* using the `EREPORT` instruction. The enclave can use this report to prove its identity to another target enclave running on the same platform. The report binds a message to the enclave's identity by using a Message

Authentication Code (MAC) with a symmetric key, called the Report key, that is only shared between the target enclave and the SGX implementation.

`EREPORT` creates a REPORT structure, and populates it with the identity and attribute information from the enclave's SECS, the SGX security version number (CPUSVN), and an optional user-data field (64 bytes) that can be used to pass additional data to the target enclave. The structure is also populated with the MRENCLAVE and attributes of the enclave which will be able to verify the report.

To create the MAC, `EREPORT` needs information about the target enclave's identity and attributes in order to derive the MAC key that authenticates the report. The platform secrets are used to derive keys, allowing different enclaves running on the same platform to derive the same key and be able to verify the MAC computed over the attestation report.

The target enclave uses the `EGETKEY` instruction to derive the same report key from its own MRENCLAVE, and a key identifier given alongside the report. It follows that the report can only be verified by the target enclave, and it can be convinced that the message is from the reporting enclave.

Enclaves on the same platform can use the security guarantees given by the attestation reports to create a secure channel, by using the message field in the report to set up an authenticated Diffie-Hellmann key exchange.

Remote Attestation

A local attestation report can only be verified by the target enclave running on the same platform. For an enclave to attest itself to a remote party, the report must be verifiable by the remote party. To achieve this, SGX uses an architectural platform service enclave (PSE) called the *Quoting enclave* to sign the report using Intel *Enhanced Privacy ID* (EPID) group signature scheme.

The remote attestation service requires an *Attestation key* to sign the report generated by an enclave, and this key can only be provisioned by Intel. To receive this key, SGX uses another special enclave called the *Provisioning enclave*, which uses the provisioning secret from the e-fuses, which is shared between the CPU and Intel, to authenticate with Intel's provisioning service. The provisioning enclave will receive the Attestation key, seal it and store it on the untrusted system.

The Quoting enclave can derive the same sealing key, unseal the Attestation Key and proceed to replace the MAC in the local attestation report with an *Attestation Signature*. The signed report is called a *Quote* and can be used verify the enclave to the remote party. To verify the Quote, Intel provides an online service called the *Intel Attestation Service* (IAS). The service exposes a simple API for developers to send in the Quote, and then receive a response stating if the signature is valid or not, i.e., it was signed by a Quoting enclave running on genuine Intel hardware. The EPID

group signature scheme used by SGX to sign the Quote is a proprietary technology, so only Intel is able to verify the EPID signature in the Quote.

Attestation Flow

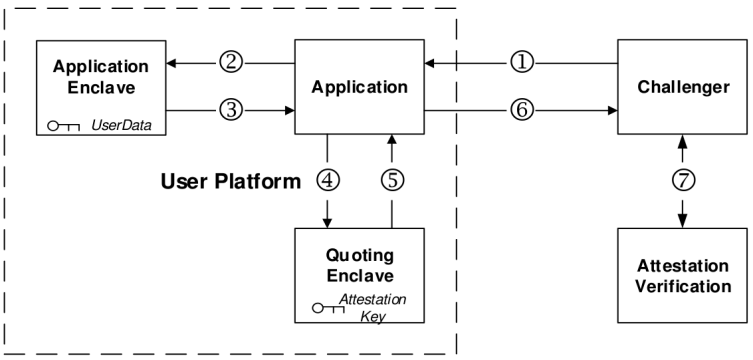

Figure 3.2: Attestation flowchart. The Challenger whats a Quote about the Application Enclave running on the User Platform. The Applications facilitates the attestation process. The Quoting Enclave signs the attestation report from the Application Enclave. The Challenger verifies the Quote before trusting the Application Enclave. [31]

After deploying an SGX enabled application to a remote platform, the author will want the application to demonstrate that it has been properly instantiated on the platform. She can challenge (with a nonce) the deployed application to prove that the trusted enclave part of the application code is using enclave he provided (1).

The application requests an attestation report from its enclave (2). The enclave generates the report containing the enclave information. The enclave can add user data like an RSA public key or a nonce to the integrity protected report, and pass it to the application (3).

The application delivers the report to the Quoting enclave (4), which will verify the report using local attestation and then create the Quote. The Quote is signed using the Attestation Key and sent back to the application (5), which then responds to the challenger with the Quote (6).

The challenger can verify the signature in the Quote using the IAS (7), and then compare the enclave information in the Quote against the trusted/expected configuration and only provision secrets or render service to the application of the configuration matches the trusted configuration (8). The trust policies are up to the challenger; she might only want to render service to the newest version of the enclave by comparing the SVN, or only if the enclave was signed using a specific signing by matching the MRSIGNER field in the Quote. She must take care to check the attributes in the Quote because the enclave will not be secure if it is running in with the DEBUG attribute. [31]

3.2 Hardware environment

The security of SGX enclaves are backed by hardware secrets and the memory encryption engine. Because of this, SGX is only available in 6th generation (Skylake) or newer Intel CPUs.

Not all platforms using a new Intel processor supports SGX. Because SGX needs to reserve part of the systems main memory for the enclaves (see section 3.1.1), there have to be BIOS support to enable SGX. OEM vendors do not typically list the BIOS features in the product sheets for their hardware, so determining if there is support for SGX in the BIOS before purchasing hardware can prove difficult. There are unofficial lists[1] of hardware that supports SGX, which could be consulted.

When enabling SGX in the BIOS, a maximum of 128MB can be reserved for SGX. To achieve this, the BIOS will expand the Processor Reserved Memory (PRM), making it unavailable to other applications.

3.3 Software environment

Intel provides platform software for both Windows and Linux operating systems. This section will only describe the platform software (PSW) and software development kit (SDK) for Linux (Ubuntu 16.04).

The SGX driver/kernel module for Linux is temporarily hosted in its own project on Intel's Github page [2]. In the future it will be upstreamed into the mainline kernel, until then, the module must be compiled and loaded to access SGX. The driver exposes the SGX hardware to the operating system, and is responsible for initializing and validating new enclaves, and in turn, tearing them down. The driver will perform all of the system calls related to the creation (`ECREATE`, `EADD`, `EEXTEND`, `EINIT`) of new enclaves, like allocating and adding the EPC pages to the enclave, measuring the EPC pages and extending the enclave measurement (see section 3.1.2).

3.3.1 SGX Platform Software

As described in section 3.1.2, to launch an SGX enclave, an Intel signed enclave must be involved in the launch process. The Launch Enclave, Quoting Enclave and the other SGX architectural enclaves are provided with the PSW package. The Linux SGX project [36] contains the source code for both the PSW and the SDK, and provides the instructions how to compile and install the packages.

After installing the PSW package, the *Architectural Enclave Service Manager Daemon* (AESMD) service must be enabled. The AESMD service exposes the API for applications to use the services provided by the

[1] https://github.com/ayeks/SGX-hardware

[2] github.com/01org/linux-sgx-driver

architectural enclaves, like launching enclaves, or creating the Quote needed from remote attestation (see section 3.1.6).

Some services [28], like a trusted timestamp and a monotonic counter service makes use of the Intel Management Engine (ME). For enclaves to make use of these services in Linux, the *Intel Capability Licensing Client* must also be installed on the platform. The monotonic counter could be used for rollback protection of sealed states, but there's many limitations because of how the monotonic counter is implemented. The counter values are stored in non-volatile flash memory inside the Management Engine, and this memory wears out and stops working if frequently used [39].

Another issue using these services arises since the ME is located on the chipset and not inside the CPU package, expanding the TCB. There have been reported multiple security issues with the Intel ME and it cannot easily be deactivated [46]. These issues could be of concern, but are beyond the scope of this thesis.

3.3.2 SGX Software Development Kit

With the driver and platform software installed, applications can launch enclaves. To develop applications that make use of SGX enclaves, Intel provides a Software Development Kit that wraps the low-level interfaces to a more friendly high-level API [29].

3.4 Hands-on Tutorial

This section will describe some of the common features of the SGX SDK using a small practical example.

Enclave definition

Writing an application that makes use of SGX enclaves requires some special considerations. The enclave is only meant to contain a small and trusted part of the application code, making the first task of the author to identify the security-critical parts of the code. This code must then be partitioned out into a self-contained library. This library is then compiled to a static library or an *enclave image*. This image is what is loaded into the EPC pages in the protected parts of memory.

To protect an untrusted system from executing arbitrary code inside the enclave, the enclave author explicitly defines the access-points to the trusted library. These access-points are defined using the *enclave definition language* (EDL). Listing 3.1 show the EDL file of a small enclave; first, the EDL file defines what other libraries are included inside the enclave. The example enclave uses the *seal* and *unseal* functions and these functions are provided with the SDK and defined in `sgx_tseal.h`.

Listing 3.1: Enclave Definition Language (EDL) file. The EDL file defines partition between the trusted and untrusted parts of the application. Trusted ECALLs and untrusted OCALLs are defined with information about the direction and size of the buffers that are passed by reference.

```
1  enclave {
2  /* trusted libraries here. */
3    include "sgx_tseal.h"
4
5    trusted { /* define ECALLs here. */
6      public sgx_status_t seal_secret(
7        [out, size=s]   sgx_sealed_data_t* sealed_data,
8        [out, count=1]  size_t*             s);
9
10     public sgx_status_t print_secret(
11       [in, size=s]  sgx_sealed_data_t*  sealed_data,
12                     size_t              s);
13   };
14
15   untrusted { /* define OCALLs here. */
16     void ocall_print_int([in] int* i);
17   };
18 };
```

The two access-points or `ECALL`s are defined on line 6 and 10. The definitions look a lot like normal functions definitions, but where the pointers to the buffers used for passing data in or out of the enclave are defined, the direction and size of the buffers have to be defined. The first `ECALL`, `seal_secret` defines an outbound buffer of size `s` to copy some sealed data out of the enclave to the untrusted *caller*. The second `ECALL`, `print_secret` defines an inbound buffer containing some sealed data of size `s`. The *caller* provides the sealed data, which is then copied into the trusted enclave memory and unsealed.

The enclave image must contain all the trusted code, but because there are some resources, like system-calls, that are not available to enclave code, the enclave will have to ask the untrusted operating system for these resources (file I/O, network socket etc.). `OCALL`s are untrusted functions outside the enclave that the enclave code is able to call. These are also defined explicitly in the EDL file. On line 16 in Listing 3.1, an `OCALL` that makes use of a *syscall* to print an integer to the screen is defined.

By observing the enclave definition, we are able to get some idea of the functionality provided by the enclave. The enclave has two access-points, where one returns a sealed blob containing some secret, and the other takes the sealed blob and reveals the secret by printing it using the untrusted system resources.

Listing 3.2: The seal_secret ECALL. The function will seal a secret number using the SGX SDK crypto libray. The sealed data is passed by reference *out* of the enclave (see buffer direction in Listing 3.1).

```
1  // generate a sealed secret and pass it to caller
2  sgx_status_t seal_secret(
3    sgx_sealed_data_t* sealed_data, int32_t* s) {
4
5    int32_t        secret = 42;
6    uint32_t       secret_size = sizeof(int32_t);
7    uint32_t       sealed_size = sgx_calc_sealed_data_size(
8      0,             // length of additional data
9      secret_size); // length of plaintext
10
11   sgx_status_t status = sgx_seal_data(
12     0,             // length of additional data (GCM)
13     NULL,          // pointer to additional data
14     secret_size,   // length of plaintext
15     &secret,       // pointer to plaintext
16     sealed_size,   // sealed_data_size
17     sealed_data); // pointer to sealed data
18
19     *s = sealed_size; // return sealed_size
20     return status
21 }
```

Enclave code

Listings 3.2 and 3.3 contain all of the enclave code except for three lines at the start of the file that `include` the trusted SGX runtime system library, the cryptographic library used for sealing and unsealing data and the enclave header. The enclave code only implements the two `ECALL`s described in section 3.4.

`seal_secret` defines a secret number, one could imagine it generating a random number or receiving a secret over a secure channel across the network. SGX seals data using the symmetric AES cipher in Galois Counter Mode (GCM) (section 2.2.1), and thus sealed data is both confidentiality and integrity protected, and there is also the option to add additional data (AD) that is only integrity protected. The secret is sealed and stored in a buffer provided by the caller (`sealed_data`) before the function returns. The seal and unseal functions do not take any key as input because the underlying implementations use the `EGETKEY`to derive the key based on the enclave measurement (see section 3.1.5)

The untrusted application that called `seal_secret` receives the sealed data, but is not able to decrypt or tamper with the data because of the guarantees provided by AES-GCM. In this example, however, the

Listing 3.3: The print_secret ECALL. The function receives a reference to some sealed data. The data is unsealed inside the enclave, and then revealed by calling a OCALL that prints the secret to the terminal.

```
1  // unseal secret number and print it using OCALL
2  sgx_status_t print_secret(
3    sgx_sealed_data_t* sealed_data, uint32_t s) {
4
5    int32_t        secret;
6    uint32_t       secret_len = sizeof(int32_t);
7
8    sgx_status_t status = sgx_unseal_data(
9      sealed_data,  // pointer to sealed data
10     NULL,         // pointer to ad
11     NULL,         // pointer to ad length
12     &secret,      // pointer to plaintext
13     &secret_len); // pointer to plaintext length
14
15   ocall_print_int(&secret);
16   return status;
17 }
```

untrusted application can have the enclave reveal the secret number by passing the sealed data to the other `ECALL`, `print_secret`. The secret is unsealed if the sealed data is authenticated, and then revealed by calling an `OCALL` that will print the secret to the standard output on the platform.

Because it is only the enclave code that can be trusted (if remotely attested), the enclave code cannot trust `OCALL`s. In this example, the enclave code cannot trust that the secret was indeed printed, or if the system printed the actual secret number. The only guarantee we have in the example code is that the confidentiality of the secret number has been compromised; but only because we explicitly allowed it.

Main application

The main application (Listing 3.4) is what bootstraps the enclave code; this is *untrusted* code that runs in the normal execution environment on the host platform.

The main application is not trusted, and cannot handle any sensitive data; but we have to rely on it to properly initiate the enclave and pass messages between different enclaves or remote resources. If we want to perform remote attestation (section 3.1.6), we rely on the main application to request an attestation report from the enclave, and pass it to the *Quoting Enclave* to generate the *Quote*. The cryptographic guarantees provided by the Quote is where the trust is grounded; the untrusted system cannot tamper with the Quote, but it could attack the availability (section 2.1.3) of the enclave by stopping the messages used in the attestation process.

Listing 3.4: Main function in the untrusted application. After initializing the enclave from the static library file called "enclave.signed.so", the application allocates a buffer to hold the sealed blob generated by the seal_secret ECALL. When the ECALL returns, the buffer contains encrypted and integrity protected ciphertext. The ciphertext is passed to the other ECALL, which intentionally reveals the secret.

```
1  int main(void) {
2    uint32_t ret_status;
3
4    initialize_enclave( &global_enclave_id,
5                        "enclave.token",
6                        "enclave.signed.so");
7
8    size_t seal_size   = sizeof(sgx_sealed_data_t)
9                        + sizeof(int);
10
11   uint8_t* sealed_data = malloc(seal_size);
12
13   seal_secret( global_enclave_id,
14                &ecall_status,
15                (sgx_sealed_data_t*)sealed_data,
16                &seal_size ); // actual seal_size returned
17
18   print_secret( global_enclave_id,
19                 &ecall_status,
20                 (sgx_sealed_data_t*)sealed_data,
21                 &seal_size );
22   free(sealed_data);
23   return 0;
24 }
```

Listing 3.4 shows how a basic main application could look like. After initializing the enclave by providing the path to the signed enclave image and launch token (see section 3.1.2) the application is able to execute the `ECALL`s implemented by the enclave. `initialize_enclave` is a wrapper for some boilerplate code for reading the enclave image and enclave token into memory. Loading the enclave into the protected memory is done by the `sgx_create_enclave` function, which is provided by the SGX runtime library.

The `OCALL` (listing 3.6) is implemented by the main application, making the enclave able to output the secret using untrusted system. Because the main application is outside of the protected enclave, the code inside cannot trust that `OCALL`s perform the expected behavior. However, like untrusted networks, the application can be trusted with passing message from the enclave to other enclaves or to the network interface. Enclaves running

Listing 3.5: The enclave initialization function. The enclave token and enclave code is used to create the enclave. The enclave ID is used as a handle to the created enclave. The function contains some additional code used to load, update, and store a launch token. (created by the architectural Launch Enclave if MRSIGNER is authenticated to launch enclaves, i.e. the author has an Intel developer License).

```
int32_t initialize_enclave( sgx_enclave_id_t* eid,
                            uint8_t* token_path,
                            uint8_t* enclave_path) {

// Step 1: try to retrieve the launch token file saved by last
    transaction, or create a new token file.
// (...)

// Step 2: initialize an enclave instance
  sgx_create_enclave(
    enclave_path,   // enclave image file
    SGX_DEBUG_FLAG, // 0 or 1
    &token,         // token structure
    &token_updated, // returns 0 or 1
    eid,            // enclave id
    NULL            // misc. attributes (optional)
  );

// Step 3: save the launch token if it is updated
// (...)

  return 0;
}
```

Listing 3.6: The print_int OCALL. The function is defined in the EDL file, allowing the enclave to call this function, which is located outside the enclave.

```
void ocall_print_int(int* i) {
    printf("%d\n", *i);
}
```

on the same system are able to establish an authenticated and encrypted channel between each other using the shared secret embedded in the CPU.

3.4.1 Enclave Communication

Establishing a secure channel over the network can be achieved by terminating a *Transport Layer Security* (TLS) connection inside enclave. Intel provides a build script for compiling *OpenSSL* to a static library for use inside enclaves [30]. The SDK exposes the enclave to a secure way to get randomness directly from the CPU, and this combined with the OpenSSL library makes the enclave able to generate asymmetric-key pairs and setup

Listing 3.7: Enclave configuration file. The enclave configuration describes the acceptable enclave configuration. It contains the enclave ID and version numbers, information about the how much memory to allocate in the EPC, the number of thread storage structures to allocate and the attribute mask that decides what attributes are acceptable. The configuration file is covered by the author signature, and the enclave will not boot if the configurations are changed.

```
<EnclaveConfiguration>
  <ProdID>0</ProdID>
  <ISVSVN>0</ISVSVN>
  <StackMaxSize>0x40000</StackMaxSize>
  <HeapMaxSize>0x100000</HeapMaxSize>
  <TCSNum>10</TCSNum>
  <TCSPolicy>1</TCSPolicy>
  <DisableDebug>0</DisableDebug>
  <MiscSelect>0</MiscSelect>
  <MiscMask>0xFFFFFFFF</MiscMask>
</EnclaveConfiguration>
```

secure channels with remote resources.

3.4.2 Enclave Build Tools

Building the enclave involves linking and compiling all dependencies to a static library; there is also a tool called the *Edger8r Tool*) included in the SDK used to generate the interface between the application and the enclave by reading the EDL file; the *SignTool* is used to create the *signature structure* (table 3.2) containing the enclave author's public-key and check if the enclave image has been constructed correctly before signing the enclave image. To build the enclave correctly, the SDK includes some sample enclave projects that contain *makefiles* that can be used as templates.

The SignTool uses the information in the enclave configuration file (listing 3.7) to set the *Security Version Number* (SVN) and *Product ID* (PRODID) of the enclave.

Chapter 4

Accountable decryption using Intel SGX

4.1 Introduction

4.1.1 Motivation

Decryption is accountable if the users that create ciphertexts can gain information about the circumstances of any decryptions made by an *decrypting agent*. The information should be meaningful, giving all the details required by the user. We describe a protocol that forces decrypting agents to create such information. The information cannot be discarded or suppressed without detection. The protocol relies on a trusted hardware device embodied by the Intel SGX technology. Many applications can be imagined, e.g., give patients control over their electronic health record, allow verifiable oversight for investigatory powers like police or national security, more transparency in corporate mail. The introduction section of this chapter is based on a workshop contribution to the *Security Principles and Trust Hotspot 2017* by Prof. M. Ryan [51].

4.1.2 Problem Statement

To achieve what we mean by *accountable decryption protocol*, there are some highlevel requirements that needs to be fulfilled.

- Users can create ciphertexts using a *public key encryption* scheme, like RSA.
- Decrypting agents (Decryptors) are capable of decrypting the ciphertexts without the help of the user.
- When a Decryptor decrypts a ciphertext, it unavoidably creates evidence that is accessible to the user. The evidence cannot be suppressed or discarded without detection.

- The users should be able to gain whatever information they require about the nature of the decryptions being performed, by examining the evidence.

Evidence

Intuitively, if the Decryptor possesses both the ciphertext and the related decryption key, it is impossible to detect if whether she applies the key to the ciphertext or not. This implies that the key has to be guarded by some trusted third party that controls its use and cannot be influenced by the Decryptor. To be able to trust the third party not to collude with the Decryptor we want this third party to be implemented by some trusted hardware device. The device contains the secret decryption key dk corresponding to ek. The dk must never leave the device.

In order to make the evidence persistent, we assume a log L. The log can be organized in various ways, e.g., like an append-only distributed ledger, realized using blockchain technology [5, 10, 59, 67], or simply as an append-only Merkle tree, like used in certificate transparency [19, 34, 52].[1] The log maintainer publishes the *root-tree-hash* (RTH) H of L, and is capable of generating two kinds of proofs about the consistency and correctness of the log:

A proof of presence of some data in the log. More precisely, given some data record d and an RTH H of the log, the log maintainer can produce a compact, proof that d is present in the log represented by H.

A proof of extension that is a proof the log in maintained append-only. Given a previous RTH H' and the current one H, the log maintainer can produce a proof that the log represented by H is an append-only extension of the log represented by H'.

(Details of such proofs can be found in e.g. [52], and will be further described in section 4.3) This means that the maintainer of the log is not required to be trusted to maintain the log correctly. It can give proof about its behavior.

Decryption

The Decryptor can perform decryptions only by using the device. The device, in turn, will perform decryptions only if it has a proof that the decryption request has been entered into the provably append-only log.

The device maintains a variable containing its record of the most recent root tree hash (RTH) that it has seen of the log. For each decryption the Decryptor performs the following actions:

[1] http://www.certificate-transparency.org/

- Obtain from the device its last-seen RTH H.
- Enter the decryption request R into the log.
- Obtain the current root tree hash H' of the log.
- Obtain from the log a proof π of presence of R in the log with RTH H'.
- Obtain from the log a proof ρ that the log with RTH H' is an append-only extension of the log with RTH H.

The Decryptor presents $(\text{R,H}', \pi, \rho)$ to the device. The device verifies the proofs, and if they are valid, it performs the requested decryption. It updates its record of the last-seen RTH with H'.

The user can find evidence about decryptions by inspecting the log. Depending on application and the needs, the evidence stored in the log could be organized in various ways; in its most simple form, the log only need store the hash value of the ciphertext that is decrypted, allowing users to detect if their ciphertexts are being decrypted.

Currency

As described so far, the protocol is insecure because the device could be tracking a version of the log which is different from the version that the users track. Although both the device and users verify proofs that the log is maintained append-only, there is no guarantee that it is the same log. The log maintainer can fork the log, maintaining each branch independently but append-only.

To ensure that users track the same version of the log that device tracks, we introduce an additional protocol for the SGX. In this second protocol, device accepts as input a *verifiably current* value v. The value v is could be provided by the user like a *nonce*. The device outputs its signature $\texttt{sign}_{sk}(v, H)$ on the value v and its current RTH H. The device holds a signing key sk; depending on the implementation, it could also be the same key as the one used for decryption. The corresponding verification key vk is published along with the encryption key ek. Like the decryption key dk, the signing key sk can never leave the device, making the device the only entity that can decrypt create signatures that can be verified using the public verification key vk.

When the user wants to inspect the log, he asks the device for the signed $H$and can verify that is consistent with their view of the log.

Summary

The protocol cannot protect the user from being denied access to the log, or the device, nor can the user stop a Decryptor from decrypting the ciphertexts she has already published. The user is able to detect if she is

being denied access, and should then assume the agreement concerning the accountable decryption has broken down. Depending on what data she has been sharing, she can choose to stop sharing it, or even re-encrypt her data, if all she shared was the corresponding encryption keys.

The device and the protocols are designed to guarantee just one thing; that if decryptions take place, this fact can be detected by the user.

4.2 Protocol Design

4.2.1 Protocol Description

As often in software systems, we distinguish two phases: the *configuration phase* (or initialization, when the whole system is being bootstrapped), which we detail more in Section 4.3; and the *operational phase*, which we detail in this section. After the configuration phase, all the cryptographic material is in place and all actors are running the required pieces of software. In the operational phase (or runtime) encryptions and decryptions of pieces of data are constantly being made, as well as accounting operations (by the user, who the scheme tries to protect).

The accountable decryption (AD) scheme that we propose is composed of four protocols, their steps being pictured at a high level in Figure 4.1. The main protocol is the AD-Decryption, which is displayed with normal arrows. This is also the protocol that has most details and does not change from application to application. The other protocols are rather application dependent, but their essential contribution to the whole AD scheme is required and pictured as such in Figure 4.1. We detail all these further down.

The participants in the AD protocol and the software components that are needed are divided into five parts.

The User (also called the encryptor) is the one that provides Data to some Application Provider (like a "FindMyPhone provider"). The user is the one producing possibly private data (like location information, or health data, or purchase records, or browsing information) and wishes to protect this data against unlawful accesses. In consequence, the user sends to the Application provider the data encrypted. One should not confuse this with some form of transport encryption (like SSL) which would only be meant to protect the messages from a network attacker.

The Application Provider (AP) is a main entity in the system and is meant to forward messages, and possibly store some information useful for more efficient implementations of the AD scheme. Moreover, the AP also stores all the encrypted data from the user. This actor is not trusted and may collude with the decryptor and the log service. Even

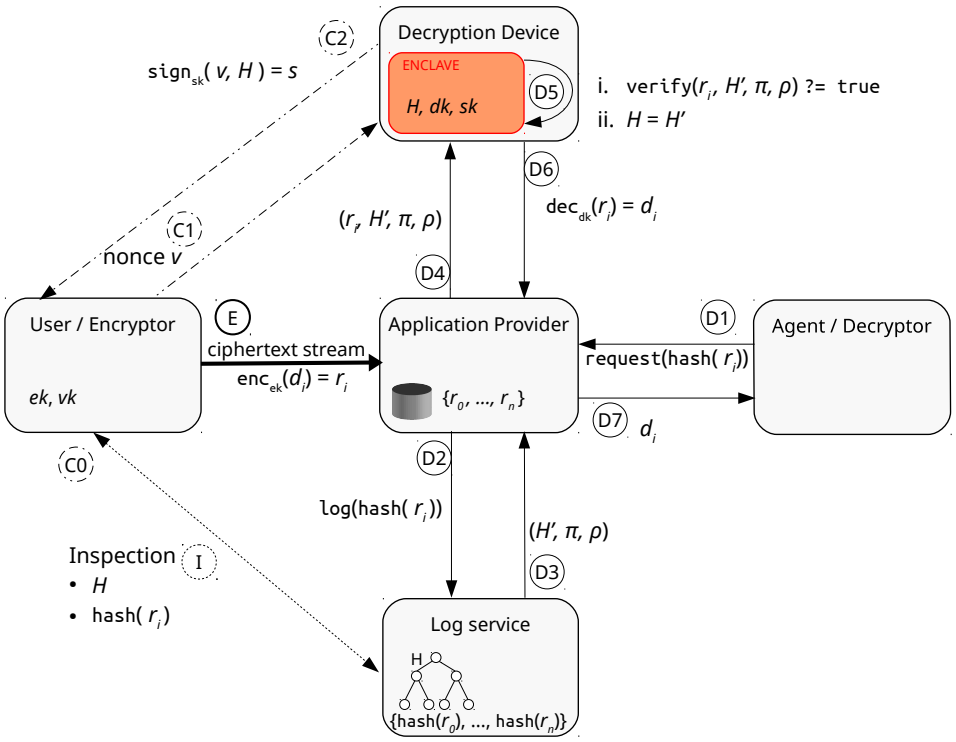

Figure 4.1: General view of the protocols involved in the Accountable Decryption scheme. The figure shows the different actors in the AD protocols. The User encrypts (E) data records using the public encryption key *ek* and sends them to the App. Provider. The user verifies the currency (C) of the Decryption Device using the public verification key *vk*. The Agent requests decryption (D) of data records from the App. Provider. The request are added to the Log, and the evidence/proofs are supplied to the Decryption service with the requested record.

more, the decryption device could be under the management of this entity as well.

The Decryption Device (SGX) is the trusted device and is implemented using the Intel SGX technology. One can possibly think of alternative implementations, and identify minimal requirements such that the overall properties of the scheme still hold; however, we do not go into this study in this chapter. The device keeps secret cryptographic keys (for decryption and signing) and performs decryption on requests from the AP.

The Decryption Agent (also called decryptor) is the one that the user wishes to account for decryptions. Examples can be law enforcement or intelligence systems that need access to information about the user, like location during some incident, or could be health access

control systems which need access to patients health records during emergency situations or during normal GP visits, etc. The decryptor is assumed to be in some form of contract or policy with the user that specifies how the decryptor is expected to behave with regards to such decryptions. This contract is used by the user during the Inspection protocol to monitor how the data is being decrypted by the Decryption Agent. If this deviates from the contract, such actions should be visible in the log service and detectable by the user by inspecting the log.

The Log Service (L) is the component of the system which is trusted to store all the decryption requests from the decryptor. In consequence, the log has to be maintained in an *append-only* fashion, like an electronic ledger (e.g., implemented using blockchain technology), meaning that once a decryption request is entered into the log, it cannot later be deleted or modified. The trust in this component is, however, obtained through cryptographic proofs, and is thus not an assumption. As such, this component can be implemented by an untrusted party like the Application Provider. These cryptographic proofs need to be verifiable by both the Decryption Device and also by the user during the Inspection protocol.

4.2.2 Cryptographic Building Blocks

We collect here the various cryptographic primitives and software constructions that we use in the AD scheme, along with their notations.

We use an *asymmetric encryption* scheme having an encryption key ek and a decryption key dk along with the two operations $enc_{ek}(-)$ and $dec_{dk}(-)$ for encryption respectively decryption, having the usual property that for any data d: $dec_{dk}(enc_{ek}(d)) = d$.

We use a *digital signature* scheme having a signing key sk and a verification key vk along with the two operations $sign_{sk}(-)$ and $ver_{vk}(-)$ for signing of information respectively verifying signatures.

We use a cryptographic hash function $h()$ to index records. The encryptor keeps at least the hashes of all his ciphertexts to be able to recognize what ciphertext was decrypted.

We need the properties provided by cryptographic hash functions because if two ciphertexts create the same hash, the encryptor could not know which of the two ciphertexts was decrypted. Another issue could arise in an application where the decryptor could influence the ciphertext entries, he would be able to construct an entry with the same hash as the one he wanted to decrypt. I.e., if every entry in a patients health journal was one ciphertext record and the doctor was expected to only look up the patients allergies, he could decrypt the allergies section, then add invisible characters, or reformat the entry until it got the same hash as for example, the mental health entry and store the new entry. He would now be able to

decrypt the mental health entry, but it could appear like he reopened the allergies entry.

We use *Merkle trees*, which are built using a cryptographic hash function (see section 2.2.3), and denote usually the head of the tree as H (maybe indexed or primed).

4.2.3 Security Assumptions

SGX Assumptions

Using the Software Guard Extensions we assume it is possible to create a trusted execution environment for our software; more specifically:

- we assume that the software is integrity protected and that the software can convince us of this.
- we assume that the computations executed by the integrity protected software can be kept confidential.
- we assume that the hardware secrets used by the SGX implementation cannot be extracted without destroying the platform, and thus an attestation signature generated by the SGX implementation is unforgeable.

Cryptographic Assumptions

In section 4.2.2 we describe some cryptographic building blocks used to realize the accountable decryption protocol. We have some assumptions on what guarantees these primitives provide.

- Hash functions
  - we assume that a cryptographic hash function generates a hash value that is a unique representation of any given input. By unique, we mean that we assume it would be unfeasible to find two different inputs that give the same hash value.
- Merkle tree
  - we assume the Merkle tree inherits the guarantees given by the hash function, and that the root tree hash is a unique representation of the leaves in the tree, including their value and order.
  - we assume any internal node in the Merkle tree is a unique representation of all its children, including their value and order. Any tree that does not contain all the leaf nodes we will refer to as a subtree.
- Public key cryptography

- we assume that it is unfeasible to decrypt a ciphertext created from a public key without the corresponding private key.
- we assume digital signatures can be verified using a public key, and the signature could only have been generated by the corresponding private key.

4.2.4 Proof Structure

The proof of presence π and proof of extension ρ take the from of two trees. Figure 4.2 shows the proofs π and ρ for the request r_7.

The proof ρ can be considered the two minimal sub-trees needed to recompute the current root hash H and the new root hash H'.

The proof π is the minimal sub-tree containing all the leafs we want to prove is present in the tree.

In figure 4.2, the request r_7 has been added to the log; we observe that because all the new leafs included in H' are needed to prove that H' is an extension of H, $\pi \in \rho$ when adding one or more requests to the log.

The device only stores the root node H of the log; proving the presence of an item in the log is achieved by providing a proof tree that includes the hash of the item as a leaf node. The guarantees provided by the cryptographic hash function ensures it is computationally infeasible to find a different tree with the same root hash.

This property is used in the first part of the proof of extension ρ to recompute the current H. The device can now use the proof π to verify that after adding the leaf r_7 to the tree represented by H, the new root is H'. In Figure 4.2 the proof of presence π is the same sub-tree as the second tree in the proof of extension ρ, but this is not always true, like in the case the extension was from $\mathtt{H}(0,6)$ to $\mathtt{H}(0,60)$. In this case, a lot of leaves would be represented by their internal parent nodes, and the proof of presence for the leaf r_7 would be different.

The log must be append-only, and this property is provided by storing the root node inside the device, and that the root hash can only be extended.

4.2.5 Encryption and Decryption protocols

The encryption protocol (denoted in Figure 4.1 by the thick arrow and the E step) is left rather open as it depends on the precise forms of data and the kinds of information a decryptor would want to obtain from it, hence the kind of decryptions. At a minimum, the encryption protocol specifies that the user is sending data to the application provider encrypted with some asymmetric encryption scheme, where the decryption key is only known by the decryption device. The user would, in for example the "FindMYPhone" application constantly update their location and send such cyphertexts, i.e., a stream on encrypted data. The AP would store the

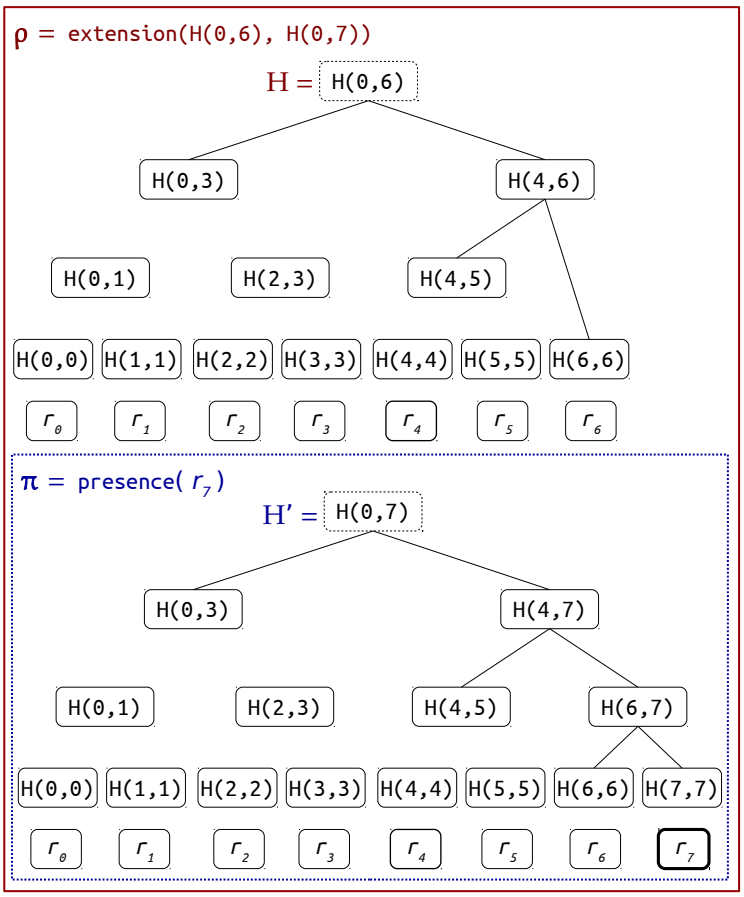

Figure 4.2: The proof of extension and proof of presence when adding the request r_7 to the log.

encrypted data indexed by the hash of it, i.e., each r_i can be retrieved through its $hash(r_i)$. Depending on the application (and thus on the intended kinds of decryptions) this data may be stored with more metadata around it.

The decryption protocol (denoted in Figure 4.1 by the normal arrows and the D steps) is the main part of the accountable decryption scheme. We detail it here.

D1: *Decryptor* $\rightarrow$ *AP* : `request(hash(`r_i`))` , with r_i the cyphertext of the required `data_i`

The decryption protocol is initiated by the decryptor through issuing a request to the AP including the identifier of the data that is intended to be decrypted.

How the decryptor obtains this hash is an implementation issue, and what kind of (meta-)information might be revealed about the data in this process of obtaining the hash are abstracted away for the

decryption protocol.

D2: $AP \rightarrow L : \texttt{log}(\texttt{hash}(r_i))$

The AP forwards this request to the log service. We encode this into a general function `log` which needs at least the identifier of the request. However, when introducing this request into the Log more information might be needed.

D3: $L \rightarrow AP : (H', \pi, \rho)$, with $H' = H + \texttt{hash}(r_i)$

The log service includes this new request into the append-only data-structure, which we denote by the extension of the head of the Merkle tree, H by the new request $\texttt{hash}(r_i)$. The Log needs to produce two proofs: the *proof of presence* ensures that the new request was indeed included in the new tree; and the *proof of extension* ensures that the new tree H' is indeed an extension of the old tree. These three elements are returned to the AP.

D4: $AP \rightarrow SGX : (r_i, H', \pi, \rho)$

The AP then forwards to the Decryption Device the cyphertext r_i along with the proofs that this decryption request have been included in the Log. The Device needs all these in order to check the correctness of the logging, including that r_i.

D5: $SGX : \texttt{verify}(r_i, H', \pi, \rho) = \begin{cases} True \rightarrow H = H' \\ False \end{cases}$

The device will check the proofs π and ρ provided with the request r_i, and if they are verified, the root hash H is updated to H', and protocol proceeds to step **D6**, else if the proofs cannot be verified, the protocol is stopped.

D6: $SGX \rightarrow AP : \texttt{dec}_{dk}(r_i) = data_i$

The device performs the decryption using the secret decryption key that it stores. In order to check the proofs the device stores the latest head H of the log against which the H' is compared. The decrypted data is being sent back to the AP.

Note that both in this step and in all others we assume that the communication channels are secured somehow from the network attacker. So the AD scheme should be built on top of a secure communication protocol (like TLS if this happens over the network).

D7: $AP \rightarrow Agent : \texttt{data}_i$

The AP forwards the data to the Agent that requested it.

A few remarks are in order. We already made the observation that the D protocol assumes another underlying scheme for securing the transmitted messages while in transit between the four entities involved. Since we do

not make assumptions about the communication medium between these entities we cannot make recommendations on how these messages could be secured. This is because the communication may not necessarily be over the network, but maybe very close on the Data Bus of the motherboard, if the Device is implemented on the same hardware of the AP that runs an Intel SGX processor.

The D protocol, and indeed the whole AD scheme, are not meant to protect the data after it is decrypted. Therefore, we do not care about subsequent decryption requests for the same ciphertext, because these are futile as the data once released to an untrusted party like the AP is no longer under the control of the user. One could very well combine with our scheme other access control schemes like for DRM control that would control how the data is being handled. Our AD scheme is meant to provide a reliable mechanism for detecting if and which ciphertexts are being decrypted. Under special implementations of our scheme, more information can possibly be counted. This information would be provided in step D2 besides the hash identifier; this is why we left the function `log` unspecified.

For the sake of the presentation, in the one decryption session above, we assumed to work with only one request, and the implementation of the Log would include only this single request. However, one would think of more efficient implementations, especially there where the granularity of the data is very fine. The session could very well work also in *batch mode* where a set of n requests are being handled at once, and included in the Log. Here would then work with one proof of extension, and n proofs of presence.

One would also want to investigate how to implement the AP to handle multiple sessions at once, for multiple users and Agents.

4.2.6 Inspection of the Log

The user should be able to inspect the log to retrieve whatever accounting information the specific application intended. This protocol is depicted as the dotted double arrow between the user and the Log service as step I. Since this is application specific, we leave it at a very abstract level. The only requirement is that the user can at least retrieve the list of requests, and is able to calculate the head of the tree, the same structure that the device is assumed to work against. Moreover, the user should at least be able to find out if some hash (i.e., request) is in the log.

When we discuss implementations for specific applications we will provide more interesting details for inspection protocols. For a specific application, the inspection protocol has to be correlated in some form with the encryption protocol. Moreover, the minimal output of the inspection protocol, i.e., the head H is necessary for the currency protocol in section 4.2.7. This is why we denoted this as step $C0$ in Figure 4.1.

4.2.7 Currency protocol

In this section we describe the *currency protocol*, denoted by the C steps and dashed and dotted line in figure 4.1.

C0: $Log \rightarrow User : H'$

The user receives the root hash H' from the log. The user would keep all her ciphertext records $R = \{r_0, \ldots r_n\}$, or just their hash values $R' = \{\texttt{hash}(r_0), \ldots \texttt{hash}(r_n)\}$; using R or R', she can identify what data d_i corresponds to $\texttt{hash}(r_i)$ in the log. When inspecting the log she can verify if the root hash $H' = \sum_{i=0}^{n} \texttt{hash}(r_i)$, and would then know what data items d_i has been disclosed.

C1: $User \rightarrow SGX : v$, where v is a random integer.

The user request the current root hash from the SGX device. The request includes a challenge to sign the nonce v along with the root hash to ensure the freshness (section 2.1) of the response.

C2: $SGX \rightarrow User : (H, s)$, where $s = \texttt{sign}_{sk}(v + H)$

The user receives the root hash H and the signature s from the SGX device. The user then checks if:

$$\texttt{verify}_{vk}(s) \wedge (H = H')$$

and if true, the request log is fresh and contains all the decryption request that the device has ever performed. The user can be convinced that the log contains every data item that has been disclosed because it would be computationally infeasible to construct a different sequence of requests that gives the same root hash.

4.3 Protocol Implementation

This section describes the implementation of the decryption device described in Section 4.2.5 using the Intel Software Guard Extensions to implement the device in software running in a trusted execution environment. The goal of the implementation is to use the tools provided by SGX to implement a prototype of a trusted piece of software that could run securely in a cloud environment and provide the decryption service for the accountable decryption protocol.

4.3.1 Defining the Enclave

The device enclave should have the following capabilities:

- Generate the two asymmetric key-pairs inside the enclave.
- Export public keys to outside the enclave.

Listing 4.1: Device state structure. The permanent state held by the device is the root hash of the Merkle Tree log, and the two RSA key-pairs used for decryption, signing. The public-keys can be exported from the RSA type when needed.

```
struct state_t {
  // Merkle-tree root
  uint8_t  *root_hash;
  // RSA key-pairs
  RSA      *decrypt_key;
  RSA      *sign_key;
};
```

- Initialize the root tree hash (RTH).
- Verify proof trees, and update the root hash.
- Decrypt ciphertexts provided along with the proofs.
- Serve decryption requests across the network.

We start the implementation process by identifying the sensitive assets to be protected inside the enclave, and how to interface with the enclave. The sensitive assets we need to protect inside the enclave is the root hash, the private decryption key, and the private signing key. We create a global structure (Listing 4.1) to hold the assets. These assets should be initialized when first creating the device enclave.

The interface with the device software is defined in the EDL file (Listing 4.2). The EDL syntax looks like that of a header file, but to be able to securely copy data in and out the enclave, the direction and size of the buffers must be known to the SGX runtime.

To support the decryption protocol we need an `ECALL` that will decrypt a provided ciphertext. The `ECALL` must also receive the proofs as arguments to be able to verify them before decrypting the ciphertext. The structure of the proofs are discussed in Section 4.3.3.

The currency protocol is used to receive the latest root hash from the device. To convince the client that he sees the latest root hash, the `ECALL` will receive a nonce as an argument, and return the root hash along with a signature of the nonce and root hash through the outbound buffers (values passed by reference, and the SGX runtime copies the data out of the enclave).

There are also needed some architectural `ECALL` to provide a means to bootstrap the device. There should be one `ECALL` to initialize the enclave to the initial state, or to a restored state if the enclave software was shut down (described in Section 4.3.2).

Listing 4.2: SGX device interface definition (EDL file). The enclave definition defines the interface to the device, and the additional libraries that should be loaded into the enclave during enclave creation (OpenSSL and the SDK seal library). The ECALLs are defined with additional information about the flow of information (in or out of the enclave).

```
enclave {
  include "sgx_tseal.h"
  from "sgx_tsgxssl.edl" import *;

  #define RSA_SIZE 384 // bytes

  trusted {
    public sgx_status_t t_initialize_enclave_state(
      [in, size=sealed_size]    sgx_sealed_data_t* sealed_state
          ,
                                size_t    sealed_size);

    public sgx_status_t t_get_public_keys(
      [out, size=enc_key_len]   uint8_t   *enc_key,
      [out, count=1]            size_t    *enc_key_len,
      [out, size=verif_key_len] uint8_t   *verif_key,
      [out, count=1]            size_t    *verif_key_len);

    public sgx_status_t t_decrypt_record(
      [in, size=proof_len]      uint8_t   *proof,
                                size_t    proof_len,
      [in, size=RSA_SIZE]       uint8_t   *encrypted_record,
      [out, size=RSA_SIZE]      uint8_t   *decrypted_record,
      [out, count=1]            size_t    *decrypted_len);

    public sgx_status_t t_get_root_tree_hash(
      [in, size=nonce_len]      int8_t    *nonce,
                                size_t    nonce_len,
      [out, size=rth_len]       uint8_t   *root_tree_hash,
      [out, count=1]            size_t    *rth_len,
      [out, size=sig_len]       uint8_t   *signature,
      [out, count=1]            size_t    *sig_len);
  };
};
```

The second architectural `ECALL` is used to provide the client with the public keys generated when the enclave was initialized. The client uses these keys for encryption, and for verifying the signed root hash.

The implementation of the architectural `ECALL` needs some security considerations to be trusted; we will discuss these in Section 4.4.1.

4.3.2 Enclave Initialization

The first thing to happen after creating the enclave is to initialize its internal state (Listing 4.1). The state consists of the two asymmetric key-pairs used

Listing 4.3: Device initialization. After creating the device enclave, the device must be initialized to generate the initial state, or to restore the last state.

```
sgx_status_t t_initialize_enclave_state(
  sgx_sealed_data_t* sealed_state,
  size_t sealed_size) {

  // Initialize state for the first time
  if(sealed_state == NULL) {
    // entropy pool to seed the PRNG used by OpenSSL
    size_t entropy_size = RSA_KEY_ENTROPY_LEN;
    uint8_t *entropy = malloc(entropy_size);

    // Initialize RTH (hash of empty str)
    t_sha256sum("", 0, &state.root_hash);

    // seed PRNG and generate decryption key-pair
    sgx_read_rand(entropy, entropy_size);
    state.decrypt_key   =
      t_RSA_generate_key(RSA_KEY_SIZE, entropy);

    // reseed PRNG and generate signature key-pair
    sgx_read_rand(entropy, entropy_size);
    state.signing_key   =
      t_RSA_generate_key(RSA_KEY_SIZE, entropy);

    free(entropy);
  // Unseal and restore the state.
  } else {
    // Allocate buffer for unsealed state
    uint8_t *buf = malloc(STATE_SIZE);

    // Unseal and populate state
    sgx_unseal_data(sealed_state,
      NULL, NULL, buf, sealed_size);

    // reinitialize state
    t_deserialize_state(buf, &state);

  return SGX_SUCCESS;
}
```

for the decryption and currency protocols, and the Merkle tree root hash of the request log.

The enclave software is to be deployed to some cloud environment, and because the enclave software is sent to the cloud in the clear, it cannot initially contain any secrets, so these must be generated securely with the first use.

The first time initializing the device enclave, the root hash is set to

the hash of some value using a specific hashing function, agreed upon with the log provider, in this case, the hash of an empty string (line 12, Listing 4.3). The SGX cryptographic library bundled with the SDK provides an implementation of the *Secure Hash Algorithm 2* (SHA-2) with a 256-bit long hash value.

The enclave must generate the two asymmetric key-pairs. We want to use the RSA cryptosystem (Section 2.2.4). To securely generate the large primes used in RSA, the key generation will require some random entropy. The SGX runtime provides a secure way to receive random entropy directly from the CPU using the `RDRAND` instruction [27].

The SGX cryptographic library does not include a full RSA implementation. To generate the keys and perform decryption, we use the *OpenSSL* cryptographic library to perform the key generation, reseeding the generator between each key-pair to ensure the keys are independent.

If the platform running the enclave needs to be restarted, or some of the device software crashes, the enclave must restore the state. The state is sealed and stored in a stable storage device on the untrusted platform after the first initialization and every time the root hash is updated. If the sealed state is provided when the enclave is initialized, the state is instead restored.

The availability (Section 2.1.3) of the device enclave can be compromised by deleting or otherwise withholding the state from the enclave. By deleting the state, the same key-pairs can never be restored, making all encrypted data records useless.

4.3.3 Verification of Proofs

The proofs are represented using the *JavaScript Object Notation* (JSON) data interchange format to simplify the interface between a device server and the decryptor client. The proofs are transformed to JSON objects following an agreed upon structure; for example, the proof of presence from Figure 4.2 can be seen in Listing 4.4. During operation, the `"Hash"` field would contain a string of the actual hash value encoded to hexadecimal or base64. The proof of extension would look very similar to the JSON object in Listing 4.4 but would contain two of the top-level tree object (line 3), one that computes to the current root tree hash, and another tree that computes to the new root tree hash.

To verify the proofs, we recompute the root hash by performing *post-order* traversal over the binary tree, where the hash value from the left child is "added" together with the hash value from the right child before returning the sum (see Listing 4.5 for python pseudo-code). The traversal routine will return the calculated root hash of the given tree, along with a list of all the hash values collected from the leaves.

Listing 4.4: JSON representation of the proof of presence from Figure 4.2. The natural tree structure of JSON-objects nicely represents the binary Merkle Tree. The hexadecimal representation of the hash-values are replaced with the same notation format as used in Figure 4.2.

```
"ProofOfPresence": {
   "RootHash" : "H(0,7)",
   "Tree": {
      "Hash": null,
      "Left": {
         "Hash": "H(0,3)","Left": null,"Right": null},
      "Right": {
         "Hash": null,
         "Left": {
            "Hash": "H(4,5)","Left": null,"Right": null},
         "Right": {
            "Hash": null,
            "Left": {
               "Hash": "H(6,6)","Left": null,"Right": null},
            "Right": {
               "Hash": "H(7,7)","Left": null,"Right": null}}}}}
```

Listing 4.5: Recursive algorithm to traverse the JSON encoded proof tree (Listing 4.4) and compute the root hash and leaf order.

```
def traverse(node, order):
    # Get the node hash value
    h = node["Hash"]

    # If node is a leaf, end recursion
    if h is not None:           # "Hash" is not 'null'
        order.append(h)         # add hash to order list
        return h.decode("hex") # return binary hash value

    # Recursive calls get left and right hash
    l = traverse(node.get("Left"), order)
    r = traverse(node.get("Right"), order)

    # Return hash of left + right hash
    return hashlib.sha256(l + r).digest()
```

The request to the device would also contain the ciphertext r_7 to be decrypted. Before decrypting r_7, we need to know if the request has been added to the Merkle tree log. The implementation represents the ciphertexts in the log by the 256-bit SHA-2 hash value $H(7,7) = \texttt{hash}(r_7)$ of the ciphertext to be decrypted.

After traversing the tree represented by the root hash $H(0,7)$, we check if $H(7,7)$ is present in the list of hashes returned by the traversal routine:

$$\texttt{order}_{H(0,7)} = \{H(0,3), H(4,5), H(6,6), H(7,7)\}$$

If we find $H(7,7)$ in the list, we know it was used in the computation of the root hash $H(0,7)$, and it's presence in the tree is proved.

To prove that the tree $H' = H(0,7)$ is an extension of the tree $H(0,6)$, both trees are traversed, and if:

$$\texttt{order}_{H(0,6)} = \{H(0,3), H(4,5), H(6,6)\} \in \texttt{order}_{H(0,7)}$$

we know that the tree H' was computed from a tree that also includes the values in the tree $H(0,6)$. If $H(0,6)$ is equal to the current root hash H stored in the device's state, the state can safely be incremented to the new root hash $H = H'$, and then proceed to decrypt the ciphertext r_7.

4.3.4 Decryption

The decryption `ECALL` (Listing 4.6) will copy the cipertext and proofs into the enclave's protected memory area. The provided ciphertext is measured, and the hash value is used when verifying the proofs. As described in the previous section, the hash value of the ciphertext must be present in the extended Merkle tree provided in the proofs. If the ciphertext is proven to be present in the request log, the new device is ready to advance it's state to the new root hash.

Before committing to decrypt the ciphertext, the device must back up its new state to some stable storage location. When the device gets a confirmation that the state has been sealed and stored, it can proceed to decrypt the ciphertext to the buffer that copies the plaintext out of the enclave.

Like during the RSA key generation, the implementation of the RSA decryption routines is provided by the OpenSSL libraries included with the enclave. The prototype uses the OAEP (see Section 2.2.4) padding scheme for RSA. The OAEP scheme makes the RSA ciphertexts non-deterministic, which would be necessary for some applications. If the ciphertext is only a randomly generated symmetric key, it would not matter, but in the case, the client encrypts his location data directly using RSA, deterministic encryptions could leak information.

4.3.5 Main Application

A small and untrusted main application is used to set up and create the device enclave in memory. When the enclave has been set up, the application will initialize the enclave. If the device enclave is to be resumed, the application provides a buffer containing the sealed state as a parameter when initializing the enclave.

During the decryption `ECALL` the enclave will store its state. This requires a `OCALL` out of the enclave with a buffer containing the sealed state. The feature to store and restore the enclave state has yet to be

Listing 4.6: Device decrypt ECALL. The function receives the proofs and the ciphertext to decrypt. After verifying the proofs, the root tree hash is updated before the ciphertext is decrypted and passed back to the untrusted application.

```
sgx_status_t t_decrypt_record(
  uint8_t *proof,
  size_t proof_len,
  uint8_t *encrypted,
  uint8_t *decrypted,
  size_t *decrypted_len) {

  // Measure the hash value of the ciphertext
  uint8_t ct_hash[SHA256_DIGEST_LENGTH];
  t_sha256sum(encrypted, RSA_SIZE, &ct_hash);

  // Verify proofs
  // The new root tree hash is calculated during verification
  uint8_t *new_rth = t_verify_proof(proof, proof_len, &ct_hash)
      ;
  if (new_rth == NULL) // could not verify proof
    return SGX_ERROR_UNEXPECTED;

  // Update, seal and store the global state
  // Will succeed if able to store updated state to storage
  int updated = t_update_root_tree_hash(new_rth);
  if (!updated) // could not update/store RTH
    return SGX_ERROR_UNEXPECTED;

  // Decrypt record in the outbound decrypted buffer
  *decrypted_len = t_rsa_decrypt(state.decrypt_key, encrypted,
      decrypted);
  if (ret < 0) // decryption error
    return SGX_ERROR_UNEXPECTED;

  return SGX_SUCCESS;
}
```

implemented but could be implemented as a file stored to a local storage device. Another option is to let the enclave use a network socket to establish an authenticated TLS connection to a remote storage device.

The main application exposes an interface to the operational `ECALL`s (*decrypt, getRootHash, getPublicKeys*) to a server application that handles remote procedure calls (RPC) from the decryptor.

4.3.6 Prototype and Future Work

The implementation consists of a working protocol prototype [55] implemented using the *Golang* programming language, and an SGX enclave prototype implemented using C and C++. The protocol prototype does not implement a trusted execution environment using SGX but implements the

full functionality, except the missing feature described in Section 4.3.5. The protocol prototype was implemented as part of the collaboration with Prof. Mark Ryan, where an M.Sc. student at the University of Birmingham was implementing the logging service which was responsible for crafting the proofs and providing a web interface to review the log. The prototype is able to verify the proofs generated by the logging service and provide the decryption service described in the design.

The prototype was implemented to inter-operate with the log service using the gRPC[2] remote procedure call framework. The framework lets us describe the RPC interface (procedures and message formats) using a specified syntax, and then using code generation, we can generate the interface code to the various different programming language with minimal effort. The log service was written in Java, and the generated RPC interface let it call the protocol functions implemented by the prototype. Some difficulties were encountered by the different RSA implementations used by Java and Golang, where ciphertext generated by Java code threw decryption errors in the Golang RSA library. This was solved by using the older *Public-Key Cryptography Standards #1 v1.5* (PKCS1v15) encryption/decryption scheme for RSA instead of OAEP.

The SGX enclave implementation [55] uses the C++ and C programming languages. The implementation features a secure enclave that generates RSA keys and decrypts ciphertexts that were encrypted using the generated key. The SGX enclave does not verify proofs before decrypting the ciphertexts. The proofs are represented using the JSON data interchange format (see Listing 4.4), and due to the SGX enclave programming model, there were difficulties including a library for handling JSON objects inside the enclave. A possible solution is to let the untrusted application parse the JSON proof structures outside the enclave and flatten the trees into arrays. The flattened proofs can be copied into the enclave and verified before decrypting the ciphertext. To interface with remote parties, the RPC interface can easily be generated for C++ as well.

Another option could be to implement the server application using a high-level language that could provide memory and type safety, and directly call the enclave functions using a foreign function interface (FFI). Memory safety could protect the application from *memory corruption attacks* like buffer overflows. Projects like the *Rust SGX SDK*[17], lets us implement the enclave itself using the *Rust*[3] programming language, providing the enclave code memory and type safety.

To implement the software attestation features required to safely deploy the device, the enclave needs another `ECALL` that creates the attestation report. The main application will have to implement a lot of extra infrastructures because it will be responsible to start up the Quoting enclave, receive the target information structure from the Quoting enclave,

[2] https://grpc.io
[3] https://www.rust-lang.org

and pass it to the device enclave. The device enclave will create its attestation report, with the Quoting enclave as the target. This allows the Quoting enclave authenticate the attestation report that is passed to it by the untrusted application (see Section 3.1.6 for a more detailed description of remote attestation flow). The complexity of developing the attestation feature placed it outside the scope of the project, that was the implementation of the accountable decryption protocol.

4.3.7 Configuration and Take in use

The protocols that we described assume the existence of some primitive cryptographic material and properly configured participants, i.e., the SGX, the service provider and the log. We describe here how this configuration can be properly obtained, and how it can be protected against crashes of any part of the system.

We use the configuration of a *Find My Phone* service as an example of how the protocol could be used to provide accountability of how the phone location data is being used by the service provider. In this example configuration, the location data used to track the phone should only be decrypted in case the phone is lost. One could also imagine that if the phone is used by a child, the parents could be authorized to request the location of the child by proxy of the phone.

The service provider would present a web interface with at least the three basic views; (1) The Account view, containing account information, (2) the Service view, where the user can find her phone and, (3) the Log view, where the user can observe the request log. The Account view would be where users could set up their account, and who is allowed to request the location data of the phone. The Service view would present the user with a map, and a button to request the position of the phone. The information tied to the request could be the IP-address, user-name, time-stamp and optionally, a text field to input the reason for the decryption. The Log view would present the user with a log of all the decryption requests, along with the information about the request, and the request ID. The log would not be very long and could be presented as a simple list. The Merkle tree, represented as JSON could be verified in the browser by client-side JavaScript, or downloaded by security-conscious users to be verified manually. By verify, we mean checking if the root hash of the tree is the same as we observe in the web interface, in the client application, and if directly querying the device. From a usability standpoint, the hash could be represented as a *hash phrase*, where the hash value is mapped into a space of a sequence of n words. The root hash would then read as a phrase of words from a dictionary, instead of a 64 character long sequence of hexadecimal numbers.

The device or enclave would be a per user construct kept by the service provider. The enclave code must be open-source so that users and security specialists could audit the code, and be convinced it cannot leak

the private-key material, or perform decryptions without creating an entry in the log. When a user signs up to the service, a new enclave is initialized, and the public keys are generated and exported to the user's account.

After creating the account, the user would download the client application to his phone. The client would retrieve the public keys and is now ready to start sending encrypted location data to the service provider. However, first, the client will request a *Quote* from the enclave, to verify that the service is backed by a genuine SGX enclave and that the software running inside the enclave is the same as the open-source code. The enclave measurement included in the Quote should be reproducible by anyone with the SGX SDK. The service provider would sign the enclave, and the measurement of their public key would also be present in the Quote. The last, but very important information contained in the Quote would be the report data field; this field would contain a measurement of the two public keys, generated during the enclave initialization. The client application could be open-source as well, and then users and specialists could be convinced that the application did proper verification of the public keys and Quote before starting to send location data to the service provider.

4.3.8 Protocol Operation

During the operation of the accountable decryption service, the client will be creating ciphertexts with the public key it received from the device enclave. The ciphertexts are uploaded and stored with the service provider and labeled with some relevant metadata. For the Find My Phone service described in the previous section, the ciphertexts would contain location data and the metadata could be the corresponding timestamp. The service provider would keep the ciphertext, and make them indexable through their web interface.

The decryptor in this scenario would often be the phone owner if he lost his phone, or parents wanting to look up where the child has forgotten their phone. Depending on the user agreement with the service provider, decryption could also be used by law enforcement in criminal investigations, or during a search-and-rescue type of missions.

When requesting a ciphertext record to be decrypted, the service provider will construct the request object, containing the ciphertext and relevant information, and then add it to the log. Their web view would be updated to show information about the decryption and the updated root hash. The log would then create the proof object, containing the proof of extension/presence, which then is passed to the device as part of the request object. The device would decrypt the location data and display it in through the web interface.

The log would have to show the decryption request and the current root hash to the user; because the user is able to use the currency protocol to request the current root hash directly from the device, she would notice if the web interface and the device gave her different values for the root

hash.

4.4 Discussion

4.4.1 Security Aspects

The goal of SGX is to create a trusted execution environment for a small and trusted software module. In the SGX model, all software except for the trusted module running inside the SGX enclave is untrusted and not a part of the TCB. By reducing the TCB to only include the enclave, the trusted module inside the enclave must be very carefully constructed to be resistant to attacks from privileged software like the OS kernel running outside the enclave. This section will discuss some of the security aspects that must be considered in order to ensure the confidentiality and integrity of the trusted module are protected from potentially malicious privileged code running outside the enclave.

SGX Security Assumptions

In Section 4.2.3 we describe some assumptions we make about the security guarantees provided by SGX.

The guarantees provided by SGX are all based on the confidentiality of the CPU specific secrets, and how SGX uses these secrets to create the signed Quote. Without the Quote, and the ability to create a secure channel using the additional data field in the Quote, SGX could not convince us about the integrity of the device software. We have to assume the confidentiality of the CPU secrets to trust the remote attestation process (described in Section 3.1.6). A more worrying issue is that only an Intel service can verify the Quote, and this would mean they could report a spoofed signature as valid if they were pressured to do so by for example law enforcement. However, we are already trusting Intel as the manufacturer of the CPU package itself, but we are expanding the TCB to encompass the Quote verification service.

SGX makes guarantees about keeping the enclave's memory confidential from privileged software, and physical memory probing by enforcing strict access control to the memory region, and by encrypting the physical memory. However, SGX cannot guarantee anything about the software running inside the enclave. In Section 3.4 we showed a practical example that would disclose the secret inside the enclave if asked to. This could of course also happen by exploiting a bug in the enclave software as well. One could imagine a *Heartbleed*[4] like a bug in the enclave software, whereby not checking the sanity of the parameters on incoming requests, and the bounds on buffers, the enclave software would read to much from its memory and possibly disclose confidential information. Some of these security

[4] http://heartbleed.com/

issues could be resolved by using a memory safe programming language to compile the enclave software (discussed in Section 4.3.6).

SGX Attack Model

We assume a powerful adversary with root privileges on the same local system as the enclave. The adversary can start, stop, and terminate enclave software whenever he wants; the adversary can even start multiple identical enclaves at the same time. The adversary can serve multiple identical enclaves a different version of the sealed state. The adversary is able to read, modify, block, and delay all messages sent by the enclave.

The adversary cannot read the enclave's runtime memory. The adversary does not have access to the processor specific seal and provisioning secret stored inside the CPU's e-fuses, and therefore cannot derive the secret key used for sealing, and attestation (Section 3.1.5). We also assume that the adversary cannot break the cryptographic primitives used by the SGX implementation, nor the primitives used by the trusted enclave module.

Roll-back Attacks Against Sealed State

The AD protocol depends on the root hash state of the device and the log to be consistent, and the client being able to get an authenticated and fresh root hash from the device. The currency protocol is used to achieve this property on the protocol level, but the make the implementation practical, we had to introduce the ability to store and restore the state in case the system needed to restart. This introduces another vector of attack on the device enclave; a roll-back attack on the sealed state.

Roll-back attacks are aimed at compromising the integrity of the enclave's state. An adversary is able to roll-back the enclave's state by stopping the enclave, and then, when resuming the enclave, presenting it with an older version of the enclave's state. The enclave will authenticate the state when unsealing it, so the adversary is not able to create arbitrary valid states, only reuse old ones.

The attack is possible because the runtime memory of the enclave is erased when stopping the enclave, and if there is no way to guarantee the freshness of the sealed state, the resumed enclave will be compromised by restoring a stale state; specifically, the integrity of the root hash will be compromised by the attack.

The attack would also allow the adversary to create a "fork" of the enclave, two identical enclaves are started on the same platform using the same initial state. The attack would allow the attacker create a "clean" version of the decryption device that is exposed to the client and another version of the device that can be used by third parties to perform decryption using a cloned version of the log. This would make decryption unaccountable and compromise the security of the AD protocol.

To protect against these types of attacks, SGX enclaves have access to a system-wide monotonic counter. The counter can only be incremented, and by having the device enclave keeping the counter as part of its state, incrementing it for every version of the sealed state stored outside the enclave. When restoring the device enclave, it will unseal the state presented to it and retrieve the counter value from the system. If the unsealed state has a different counter than the counter received from the system the device will detect that the state has been tampered with.

The way Intel's monotonic counter service stores the counter in non-volatile flash memory inside the Intel Management Engine (ME) creates issues with scaling. There are also some security concerns since the ME is not part of the CPU package, but part of the chipset and mounted to the motherboard [39]. A more detailed description of the system-wide monotonic counter can be found in Section 3.3.1.

By implementing roll-back protection using either the local monotonic counter, or a distributed counter, as proposed in [39], the adversary cannot create a forked device, or reset the device to an old state.

Side-channel Attacks Against Enclaves

The accountable decryption protocol makes some assumptions about the trusted execution environment provided by SGX (Section 4.2.3). Specifically, we assume that the secret RSA private key is kept confidentiality protected inside the enclave and that the secret AES key used when sealing the state is also not disclosed. Both of these cryptographic schemes can be vulnerable to *cache-based timing attacks*, where the timing of the different operations can reveal information about the secret key. [24, 25]

The SGX attacker model is to protect user-level software from privileged software running on the same platform. However, the Intel CPU provides privileged software, like the OS kernel with powerful profiling tools and the ability for fine-grained hardware thread pinning, allows privileged code to run a spy-process on the same hardware thread as the enclave software. This means that the spy-process shares the cache with the enclave software, and can use different types of probing operation to get information about the memory access-pattern of the enclave. By probing different memory locations, and exploiting the property that fetching memory from the cache is much faster than from main memory, the attacker gets information about the enclave's memory access pattern when accesses to its own memory produces a cache miss.

Root-level cache-timing attacks against an AES software implementation running inside an SGX enclave are able to extract the AES secret from the enclave [22]. The AES implementation that was used was known to be vulnerable to these timing attacks and showed that SGX could not protect against cache-timing attacks. However, the AES implementation provided with the SGX SDK used for sealing was not vulnerable to this exact type of timing attack.

In a cloud environment virtual machines are co-located on the same physical machine; utilizing this, another side-channel attack on SGX enclaves uses a malicious co-located enclave to steal the RSA private key from the enclave using a cache-based timing attack. This attack could also be very difficult to detect because the malicious code is running inside its own enclave. The use of random noise during the computations can be used to hide the secret RSA key from this type of timing attack (see Section 2.2.4). [54]

It is obvious that Intel SGX does protect against side-channel attacks against the software running inside the enclave. This means the enclave authors must take care to protect their software from side-channel attacks. Intel states that these types of attacks are edge-cases and that SGX cannot protect against them [47].

Cryptographic Primitives

We assume the cryptographic primitives cannot be broken if implemented correctly and when using long enough keys to make the key-space large enough to make brute-force attacks infeasible. As of November 2017, the SGX SDK includes only a limited cryptographic library. The library only implements the AES algorithm (Rijndael [14]) using 128-bit keys. This building block is used to implement AES in Galois Counter Mode (GCM) and Counter Mode (CTR), and Cipher-based Message Authentication Code (CMAC). For cryptographic hashes, the library implements SHA-2 with 256-bit long hash values. The only asymmetric cryptographic primitives included in the SGX SDK are digital signatures, using either 3072-bit RSA or the elliptic curve digital signature scheme (ECDSA) with 256-bit keys. NIST recommendations for key management considers all of these secure to at least the year 2030[5] [6].

The SGX OpenSSL library [30] used to implement the device is a general purpose cryptographic library and implements many more primitives and with longer keys. The API supports the AES algorithm with the full 256-bit key length, SHA-2 with 512-bit long hash values, and RSA crypto, and signature scheme with 4096-bit keys. It also supports the MD5 hash algorithm, which is considered insecure, and the SHA-1 hash algorithm, which is considered legacy for many use-cases [6]. Hash collisions using SHA-1 was demonstrated in 2017 by researchers at Google [58], but needed around 6500 CPU years, or 100 GPU years to find a collision.

4.4.2 Other Applications and Configurations

The "Find My Phone" service described in Sections 4.3.7 and 4.3.8 is just one specific implementation of the *accountable decryption* protocol, but we could imagine it being used for many other applications where we want some

[5] https://www.keylength.com

entity to have access to our data, but have accountability of its use. Patient journals could be another application, where we want to know what type of health information is being inspected, and that it is only our doctor that is inspecting the journal.

Another possible use for the protocol could be in electronic voting systems. Instead of returning the plain-text, the device could tally up the decrypted votes, and at the end return the final results. This log combined with the root hash in the device would provide the proof that every encrypted vote was counted, and then no vote was counted twice.

The Currency protocol could be implemented differently than in our approach; we chose to let the user directly challenge the device with a nonce in order to get a fresh root hash from the device. Another option could be for the device to periodically publish the root hash using an unpredictable value that everyone can verify.

The *Tor Project*[6] has specified a distributed random number generator to create a global shared random number each day [61]. This random number could be signed along with the root hash; the user could then verify freshness by checking the value published by the distributed system herself. We have already mentioned the *ROTE* paper [39] for using a distributed system to protect against roll-back attacks, and we could imagine the same system could be used to generate the global random variable or a trusted time-stamp for the Currency protocol.

4.5 Summary

In this chapter, we have described the design and implementation of *Accountable Decryption* protocol. The design relies on a trusted hardware device and using the capabilities that Intel's Software Guards Extensions can provide we implemented the device in software that can be deployed on all newer Intel CPUs.

We have discussed some of the security aspects of the implementation and found that using SGX is no "silver-bullet" for security in cloud applications because of cache-based timing attacks against the cryptographic primitives running inside the enclave.

We also discussed how roll-back attacks could break the security of the protocol by replaying decryption requests to it, and how to protect the device against this type of attack.

[6] https://www.torproject.org/

Chapter 5

Securing the Signal server using SGX

5.1 Introduction

This chapter presents discussions on how entire applications could be secured using the hardware security technology provided by Intel Software Guard Extensions (SGX). We have chosen one specific application, the Signal messaging protocol. Much of this chapter was first presented in April 2017 during the 5th Workshop on Hot Issues in Security Principles and Trust (HotSpot) [56].

Signal is a recent secure messaging protocol, descendant from the classical off-the-record protocol, which has lately become popular partly due to the Snowden revelations. Signal, or variants of it, are now implemented, or under way of being implemented, in various major chat products, like from Facebook, WhatsApp, Google.

However, when studying communication protocols, usually one focuses on the protocol itself. In contrast, we are here focusing on securing the central message distribution and user discovery of such an end-to-end secure communication protocol, i.e., the centralized server application.

We make use of hardware enabled security features provided by the recent technology of Intel's SGX, part of the newer Skylake architectures. However, working with SGX can be tedious, therefore, we are looking at simpler ways of programming, using the recent SCONE secure containers. These are the secure counterparts of the popular Docker containers, implemented using SGX. Our work of implementing Signal using Intel's SGX can also be seen as an exploration and testing of the new features and performance of this new security technology from Intel.

5.2 Motivation

Secure messaging protocols have been around for more than a decade, with off-the-record (OTR) protocol[1] [8, 16] being a prominent example. OTR also has been implemented in standard instant messaging clients for quite some time, e.g., in Adium[2] (for MacOS), Jitsi[3] (cross-platform), or through plug-ins in the popular Pidgin[4] (for Linux). However, these have not seen wide adoption, partly due to usability difficulties [57, 63, 64], but also partly due to lack of motivation from the users. The Snowden revelations, however, triggered more concern, and recently we have seen an explosion in secure messaging implementations, with prominent example being the Signal protocol (formerly known as TextSecure). A few recent studies appeared about secure messaging in general [50, 63], as well as formal analysis of Signal/TextSecure [11, 20, 32].

The Signal messenger, like many other secure messenger applications[5] relies on a centralized infrastructure to achieve asynchronous[6] communications between clients. The content of messages going through the server are end-to-end encrypted, but information about the sender and receiver is known to the server in order to route the messages [50]. This, and other, metadata can be used to obtain sensitive information about the clients [40]. The Signal server also facilitates contact discovery for users, and is performed by users uploading their contact list to the server. Even if phone numbers are anonymised, the numbers could be deanonymised by constructing a social graph from all the contact lists.

Keeping the metadata secure requires trust in a large software and hardware stack. Even more so when the Signal server is set up in a cloud environment, since the trusted computing base (TCB) will include privileged software like firmware, hypervisor, the providers cloud management software, and in many cases the operating system as well, since cloud-providers supplies pre-built images. Not only do you have to trust the providers hardware and software stack, one also has to trust the cloud-provider's personnel, like the system administrators, and other personnel with physical access to the hardware. New research suggests you even have to trust other customers of the cloud-provider due to attacks like *memory massaging attacks* [49] which can fully compromise co-hosted cloud VMs [60]. To secure the server and metadata in this threat-model we have to remove the privileged software from the TCB, and protect the application's memory from lateral attacks.

[1] https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

[2] https://www.adium.im

[3] https://jitsi.org/Main/About

[4] https://developer.pidgin.im/wiki/ThirdPartyPlugins

[5] https://en.wikipedia.org/wiki/Comparison_of_instant_messaging_clients

[6] Note that OTR was intended for synchronous communications as with chats, and is thus not usable for securing SMS like asynchronous communications (a few modifications are needed, see [20]).

The threat-model assumes that the central server are secure from malware and hardware attacks. However, considering the large TCB, we would like to remove this assumption by using Intel Software Guard Extension (SGX) [1, 41] to keep the metadata encrypted in memory. Moreover, we think that the same technology of Intel's SGX can be used to similarly secure the desktop Signal clients. However, for mobile clients (e.g., running inside Android environments) one needs to investigate alternatives (e.g., ARM's TrustZone).

Our motivation is similar in spirit to the motivation of the authors of the recent article [9] where they want to secure the data handled by the Apache ZooKeeper (used for coordination of distributed systems) against privileged software, like hypervisors. They compare two approaches, either implementing the whole application inside an SGX enclave, or implementing only specific security functionalities inside enclaves, e.g., for encrypting data before storing or passing it around.

5.3 Technical Details

We first present the technologies that we plan to use, and in the end we give a short presentation of the Signal instant messaging protocol.

5.3.1 Intel SGX

The SGX model (described in section 3.1) is to create a secure enclave with integrity and confidentiality protection from privileged code running on the same system at the secure enclave. However, the enclave is only meant to hold a small, trusted and security-critical software model.

To secure existing software using SGX without rewriting the software, there is the option to put the whole application inside the enclave. The enclave could provide the application with confidentiality and integrity guarantees, and remote attestation would allow us to deploy the application with a cloud provider with the same security guarantees.

5.3.2 Linux Containers

The concept of *software containers* tries to solve the problem of managing software dependencies [43]. Conflicting or missing dependencies can be a big problem when deploying software to different services. If the developer does not have the same versions of the software as running in the production environment, a dependency conflict might break the functionality of the application. To solve this containers uses an isolated runtime environment, and pack the dependencies together with the application inside the container. A popular implementation is the *Docker*

containers,[7] which also provide a repository[8] of Docker images curated by both trusted developers and the Docker community, making it very easy to pull and deploy applications.

Unlike virtual machines, that virtualizes the hardware of a machine, Linux container software like Docker uses OS-level virtualization to isolate processes and their runtime environment. This makes containers much more lightweight than VMs since they do not need to boot up a full operating system to start the application. A container has all the files and binaries needed to run the application, but uses the operating system for services like I/O and resource management. Docker builds on the technology of Linux Containers (LXC)[9] to provide containerization. Using *kernel namespaces* the containers get their own view of system resources, and using *control groups* these resources can be limited by the host.

5.3.3 Approach: SCONE secure containers

An alternative approach to the SecureKeeper re-implementation using small enclaves is to run the entire unmodified application inside one enclave. Previous work on this approach include *Haven* [7], where a library operating system [37] and a shield module that handles scheduling threads, memory management and a file system were included inside the enclave to be able to run unmodified windows applications inside the enclave. A drawback of this solution is the large subset of Windows that the library OS includes, which adds considerable extra code to the TCB.

Recent work [3] in running unmodified applications inside a single SGX enclave make use of Docker containers instead of a library operating system. Thus, the objective of *SCONE* [3] is to make a *secure container* mechanism by placing the application and application-specific libraries of Docker containers inside an enclave.

Running unmodified applications inside enclaves requires a standard C library (libc) interface and an external interface to execute system calls, since enclaves do not support system calls. SCONE includes the *musl*[10] libc library and the *Linux Kernel Library* [48] (LKL) to create a small Linux library OS.

Exiting and entering enclaves is an expensive operations, since it needs to do a context switch from the protected stack, and then sanitize the CPU registers so as to not leak information. To minimize enclave exits and entries, SCONE uses the hybrid (M:N) threading model, and supports asynchronous system calls by writing system calls on a queue outside the enclave. As seen in figure 5.1, the kernel module on the host will execute the system calls from the call queue, and put responses in the response queue.

[7] https://www.docker.com
[8] https://store.docker.com
[9] https://linuxcontainers.org/lxc/
[10] https://www.musl-libc.org

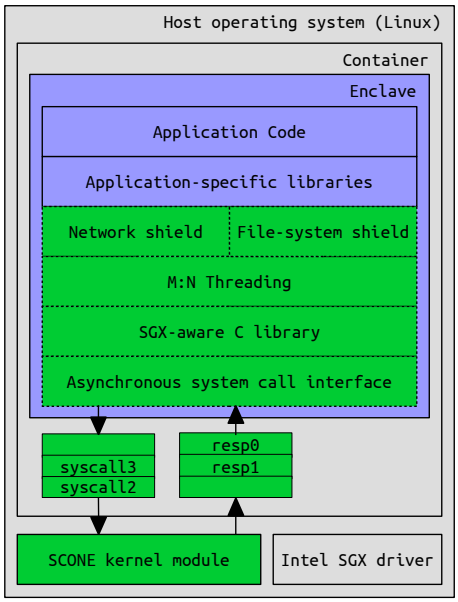

Figure 5.1: The SCONE architecture (green). The host OS uses a custom kernel module to execute system-calls on behalf of the SCONE container. The container runs the SGX enclave (blue) that contains the application. The enclave also contains the I/O shields, a thread scheduler, a minimal C library and a system-call dispatcher. [3]

To protect the enclave code from a malicious operating system, the system call interface does various checks on the system call parameters and responses, like checking if pointers and buffers resides inside or outside the enclave. The authors of [3] describe three different *shield modules* to protect I/O operations: the *file-system shield*, *network shield* and *console shield*. The file-system shield protects the confidentiality and integrity of files by transparently encrypting files used by the containers overlay file-system, which resides outside the secure enclave. The network shield encrypts the container's network interface transparently using TLS. The console shield encrypts the unidirectional console stream from the application using symmetric-key encryption; enabeling the operator to decrypt the console stream from a trusted environment. The shield modules are extendible, in case the containerized application has additional interfaces that require protection.

To create the secure containers for SCONE, the applications must be built as a SCONE executable by statically compiling them with the application-specific libraries, and the SCONE libraries. There is also need for some additional configurations in order to enable and configure the different shield modules. When complete, the secure Docker image can be published using the standard Docker Store. The secret information needed by the enclave to encrypt the file-system and console stream is provided by

a special configuration file called the *startup configuration file* (SCF). The SCF is not included in the image, but is sent to the enclave over the TLS secured channel after SGX has verified the integrity and identity of the enclave.

Side-channel attacks on SGX enclaves was discussed in section 4.4.1 , and the threat models of neither SCONE or Haven considers side-channel attacks. We discussed how these type of attacks could be performed by the privileged software on the host platform, and research [65] has shown that these types of attacks can extract complete text documents and outline of images processed by a legacy application running inside a Haven container.

5.3.4 Other approaches: SecureKeeper and Graphene libOS

Implementing native SGX support for the Signal server will require some additional code to invoke the special system calls used to create and enter the enclave. One approach is to identify the critical sections in the server that handles the metadata and routing of messages, and implement them using SGX. *SecureKeeper* used this approach to implement *Apache ZooKeeper* using SGX [9]. In SecureKeeper, an SGX enclave is used as a secure entry point for TLS encrypted requests to the server. The requests are decrypted inside the enclave, then the payload and path field of the requests are encrypted again before passing the request out of the enclave to the normal ZooKeeper code. The re-encryption of the payload and path is transparent to the ZooKeeper cluster, and is basically working like a disk-encryption scheme for the cluster. For SecureKeeper this approach worked well, and with little overhead, as measured by the authors. However, this may require more detailed programming knowledge of the system to be secured, as well as a re-implementation.

Scone provides a very small TCB by only including a common C library inside their secure container, but it should also be possible to run the Signal server inside a full library OS (also called unikernels). The Graphene library OS [62] supports multi-process applications, and have recently added support for running unmodified binaries inside SGX Enclaves[11]. Graphene claims to be able to run Java applications on top of OpenJDK inside a enclave with minimal development efforts.

5.3.5 Signal protocol

Signal is an instant messaging protocol devised with similar goals as the off-the-record protocol, i.e.,

end-to-end security or confidentiality, where only the intended conversation partners are able to read a message; in particular, the message should not be available to a third party like an intermediary server (offering some infrastructure support);

[11] https://github.com/oscarlab/graphene/wiki/Introduction-to-Intel-SGX-Support

deniability which, given a sequence of messages and the relevant keys, ensures that there is no way for a judge to prove that a certain message was authored by a certain user;

forward secrecy which ensures that previously encrypted messages cannot be decrypted upon obtaining current and/or future keys;

future secrecy which ensures that a message cannot be decrypted even if keys from previous sessions are being compromised.

The Signal protocol goes over several phases, and a third trusted server is also involved, besides the two honest parties participating in the conversation. By honest parties it is only assumed that their long-term cryptographic material is not compromised.

The *registration phase* involves the Trusted Server, as well as Google Cloud Messaging system (GCM). The trusted server needs the phone number of the participant to which a verification token is sent to check the ownership of the phone; the exchange of messages is done through HTTP and uses basic authentication. Various cryptographic material will be stored on the server for this user, some used to encrypt messages sent to GCM, others, the pre-keys, are used in encrypting messages.

A *key comparing phase* can be done by the human parties, similar to what is done in OTR. In this stage the Signal App can compute a QR code, to make the comparison automatic.

Sending message phase involves the trusted server to provide the stored pre-keys to be used in a complex key derivation algorithm called Axolot-ratcheting. This this conversation with the server, more information than just the message is being sent. In particular, identities of the participants.

Sending subsequent messages, or sending reply messages within the same session, does not involve the trusted server.

5.3.6 Signal server

In [3] the authors built and benchmarked secure Docker images of *Redis*, *NGINX*, and *Memcached*. These applications are implemented in C, and can run natively inside the secure container. The Signal server, however, is a Java application, and needs to run on top of a *Java Virtual Machine* (JVM). A lightweight JVM like JamVM[12] could be statically compiled to use the SCONE libraries and system call interface and included inside the enclave.

The Signal server has at least three security critical-parts. When sending a message to some user using Signal, the message is end-to-end encrypted

[12] http://jamvm.sourceforge.net

using the recipients public-key, but the message header also contains the phone number of the recipient [50]. The network shield of SCONE will protect this data since the traffic is only decrypted inside the enclave. Both the SecureKeeper and the SCONE approach would be able to secure this information since both will protect the communication end-points. The server already tries to protect the user-data on the server by only storing a hash of the phone numbers. To look up the contact information would require hashing the phone number and searching the database. Both the computations and the database should be hidden to the host, and the approach used by SCONE should accomplish this by running all of the application inside the enclave and using the file-system shield of SCONE to protect the database.

To facilitate asynchronous first-time communications between users, the Signal server also stores a number of precomputed Diffie-Hellman public-keys from the users. These *pre-keys* are used to generate a shared secret between the users. Since pre-keys are tied to a user, this information should also be protected, and the same mechanism as above could be used to protect this information.

SCONE have not yet implemented support for SGX remote attestation, but using this feature is quite important if deploying the server to the cloud; remote attestation will confirm that the server is protected by a genuine SGX implementation and that the application code has not been modified.

5.4 Summary

In this chapter we proposed some approaches to secure legacy applications using technologies like secure containers, or by rewriting the security-critical parts of the application. We used the Signal server as an example, and observed that it would be possible to secure it using technologies like SCONE, Haven or other library operating systems running inside a SGX enclave. However, we are forced to consider side-channel attacks against the enclave from privileged software as practical and able to break the confidentiality of the operations performed inside the enclave.

In late 2017 the developers of the Signal Messenger have started to implement a secure module for their server to provide private contact discovery using SGX. The module is not a security improvement on the Signal protocol, but used to improve the privacy for the users on their platform [38].

Chapter 6

Conclusion

This thesis has given a theoretical and practical hands-on tutorial to Intel's Software Guards Extensions (SGX) technology. Based on the work of Costan and Devadas [13] and the Intel documentation, we have covered the theoretical background of the technology and tried to explain how it works, and how it can provide the claimed security guarantees. Using the theoretic background and Intel's own development reference [29] we have covered the practical parts of developing applications using SGX, and we have shown how we can use SGX to develop secure applications that can be deployed to cloud infrastructure with close to the same security guarantees as when running the application locally.

The second contribution of the thesis is the development of a novel protocol that enforces accountability to parties wanting to decrypt user data. The protocol depends on a trusted decryption device that was implemented using the SGX hardware-enabled protection features. These protection features allowed us to generate, store and use the secret decryption keys on the remote host, without the host being able to observe them.

Using the software attestation features provided by SGX we can be assured that the application has not been tampered with by the remote host, and allows us to authenticate the public encryption keys we received from the deployed application. We have discussed the security of the protection features and discovered that there are some caveats to the use of SGX to secure applications running on a remote host, and especially side-channel attacks, which can be very difficult to discover and protect against. We also discussed how the protocol could be broken by implementation details, and how we can protect against the proposed roll-backs attack by providing freshness guarantees on the device's state.

The third contribution of this thesis is a proposal for running the central server of the Signal instant messaging service inside an SGX enclave. While the Signal server was used as an example, the approach of using the SCONE containers [3] or by running the application on top of a full library operating systems like Haven [7] inside SGX enclaves can be generalized and applied any existing server application deployed to a remote host. We

also discussed some security issues with the proposed solution.

6.1 Critical Reflections

While a working prototype of the device used by the *Accountable Decryption* (AD) protocol was finished, we did not finish developing all the functionalities using the SGX technology. There was not enough time to complete implementation of the proof verification using SGX, and therefore we have not made performance evaluations of the device either. The results from such tests could tell something about the viability and scalability of the proposed system.

Regarding the AD protocol, throughout the thesis we have assumed the protocol itself to be secure. We have not tried to analyse or formally verify the security of the protocol. The result of such an analysis could help us improve the security of the proposed protocol, or it could possibly prove the protocol to be insecure.

Chapter 5 discusses the possibility of securing the Signal server using the SGX technology; this work was the preliminary work of this thesis. The Signal server runs on top of the Java Virtual Machine (JVM), and because of the technical difficulties of running a full JVM inside an SGX enclave, the work was postponed.

6.2 Further Work

A new survey could provide some numbers on how well SGX applications perform; we could evaluate the performance of the verification and decryption routines from the Accountable Decryption protocol, both inside, and outside an SGX enclave. The survey could tell us about the economy of running this type of decryption service.

In our discussion on the AD we mostly assumed one enclave per user, but this is not necessary; in a shared enclave implementation, where all the users used the same public log and device, there would be a lot of attestation/public-key request to the same enclave, and it would be interesting to evaluate the performance of the remote attestation request against a single enclave.

The security discussions on the AD protocol went to some length into the issues of side-channel attacks against SGX enclaves 4.4.1. Surveying if the device is vulnerable, and if, how to fix such vulnerabilities could lead to interesting results.

Distributing the AD protocol could be the next step in scaling the performance, along with the additional security guarantees that could be provided by for example the ROTE system [39].

Another interesting avenue of further research would be to do formal verification of the Accountable Decryption protocol.

Bibliography

[1] Ittai Anati et al. "Innovative technology for CPU based attestation and sealing." In: *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. ACM, 2013. URL: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[2] ARM. *Building a Secure System using TrustZone® Technology*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf (visited on 11/12/2017).

[3] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 689–703. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[4] W. Arthur, D. Challener, and K. Goldman. *A Practical Guide to TPM 2.0*. APress, 2015. DOI: 10.1007/978-1-4302-6584-9.

[5] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)." In: *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Matteo Maffei and Mark Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. ISBN: 978-3-662-54455-6. DOI: 10.1007/978-3-662-54455-6_8. URL: https://doi.org/10.1007/978-3-662-54455-6_8.

[6] Elaine Barker and Quynh Dang. *NIST Special Publication 800–57 Part 1, Revision 4*. 2016. DOI: 10.6028/NIST.SP.800-57pt1r4.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven." In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 8:1–8:26. ISSN: 0734-2071. DOI: 10.1145/2799647.

[8] Nikita Borisov, Ian Goldberg, and Eric Brewer. "Off-the-record Communication, or, Why Not to Use PGP." In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*. WPES '04. Washington DC, USA: ACM, 2004, pp. 77–84. ISBN: 1-58113-968-3. DOI: 10.1145/1029179.1029200.

[9] Stefan Brenner et al. "SecureKeeper: Confidential ZooKeeper Using Intel SGX." In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. New York, NY, USA: ACM, 2016, 14:1–14:13. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988350.

[10] Konstantinos Christidis and Michael Devetsikiotis. "Blockchains and smart contracts for the internet of things." In: *IEEE Access* 4 (2016), pp. 2292–2303.

[11] Katriel Cohn-Gordon et al. "A formal security analysis of the Signal messaging protocol." In: *2nd IEEE European Symposium on Security and Privacy*. IEEE, 2017. URL: https://eprint.iacr.org/2016/1013.

[12] Intel Corporation. *Intel® Software Guard Extensions Programming Reference*. 2014. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[13] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[14] Joan Daemen and Vincent Rijmen. "AES proposal: Rijndael." In: (1999).

[15] S. Delaune et al. "Formal analysis of protocols based on TPM state registers." In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*. Cernay-la-Ville, France: IEEE Computer Society Press, June 2011, pp. 66–82. DOI: 10.1109/CSF.2011.12.

[16] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Secure Off-the-record Messaging." In: *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*. WPES '05. Alexandria, VA, USA: ACM, 2005, pp. 81–89. ISBN: 1-59593-228-3. DOI: 10.1145/1102199.1102216.

[17] Yu Ding et al. "POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave." In: ACM Conference on Computer and Communications Security. Dallas, USA, 2017. URL: https://github.com/baidu/rust-sgx-sdk/blob/master/documents/ccsp17.pdf.

[18] Gordana Dodig-Crnkovic. "Scientific methods in computer science." In: *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*. 2002, pp. 126–130.

[19] Benjamin Dowling et al. "Secure Logging Schemes and Certificate Transparency." In: *Computer Security – ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*. Ed. by Ioannis Askoxylakis et al. Cham: Springer International Publishing, 2016, pp. 140–158. ISBN: 978-3-319-45741-3. DOI: 10.1007/978-3-319-45741-3_8. URL: https://doi.org/10.1007/978-3-319-45741-3_8.

[20] T. Frosch et al. "How Secure is TextSecure?" In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. Mar. 2016, pp. 457–472. DOI: 10.1109/EuroSP.2016.41.

[21] Dieter Gollmann. *Computer security*. John Wiley & Sons, Inc., 2010. DOI: 10.1002/wics.106.

[22] Johannes Götzfried et al. "Cache Attacks on Intel SGX." In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-4935-2. DOI: 10.1145/3065913.3065915. URL: http://doi.acm.org/10.1145/3065913.3065915.

[23] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[24] Berk Gülmezoğlu et al. "A Faster and More Realistic Flush+Reload Attack on AES." In: *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*. COSADE 2015. Berlin, Germany: Springer-Verlag New York, Inc., 2015, pp. 111–126. ISBN: 978-3-319-21475-7. DOI: 10.1007/978-3-319-21476-4_8. URL: http://dx.doi.org/10.1007/978-3-319-21476-4_8.

[25] Mehmet Sinan Inci et al. *Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud*. Cryptology ePrint Archive, Report 2015/898. http://eprint.iacr.org/2015/898. 2015.

[26] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (visited on 10/27/2017).

[27] *Intel® Digital Random Number Generator (DRNG) Software Implementation Guide | Intel® Software*. URL: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide (visited on 10/28/2017).

[28] *Intel® SGX Trusted Platform Service Functions*. URL: https://software.intel.com/en-us/node/709050 (visited on 10/18/2017).

[29] *Intel® Software Guard Extensions SDK*. Aug. 21, 2017. URL: https://software.intel.com/en-us/documentation/sgx-sdk-developer-reference (visited on 10/21/2017).

[30] *intel-sgx-ssl: Intel® Software Guard Extensions SSL*. original-date: 2017-04-26T22:09:28Z. Sept. 25, 2017. URL: https://github.com/01org/intel-sgx-ssl (visited on 10/21/2017).

[31] Simon Johnson et al. *Intel SGX: EPID Provisioning and Attestation Services*. 2013. URL: https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services (visited on 11/09/2017).

[32] N. Kobeissi, K. Bhargavan, and B. Blanchet. "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. to appear. 2017. URL: http://prosecco.gforge.inria.fr/personal/bblanche/publications/KobeissiBhargavanBlanchetEuroSP17.pdf.

[33] Donald C Latham. "Department of defense trusted computer system evaluation criteria." In: *Department of Defense* (1986).

[34] Ben Laurie et al. *Certificate Transparency Version 2.0*. Internet-Draft draft-ietf-trans-rfc6962-bis-26. Work in Progress. Internet Engineering Task Force, July 2017. 54 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-trans-rfc6962-bis-26.

[35] Matt Lepinski and S Kent. "RFC 5114-Additional Diffie-Hellman Groups for Use with IETF Standards." In: *See Section 2.1. 1024-bit MODP Group with 160-bit Prime Order Subgroup* (2008).

[36] *linux-sgx: Intel SGX for Linux**. original-date: 2016-02-23T23:41:25Z. Oct. 13, 2017. URL: https://github.com/01org/linux-sgx (visited on 10/18/2017).

[37] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 461–472. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451167.

[38] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Signal. Sept. 26, 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 10/21/2017).

[39] Sinisa Matetic et al. "ROTE: Rollback Protection for Trusted Execution." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 48. URL: https://eprint.iacr.org/2017/048.

[40] Jonathan Mayer, Patrick Mutchler, and John C. Mitchell. "Evaluating the privacy properties of telephone metadata." In: *Proceedings of the National Academy of Sciences* 113.20 (May 17, 2016), pp. 5536–5541. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.1508081113.

[41] Frank McKeen et al. “Innovative Instructions and Software Model for Isolated Execution.” In: *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’13. Tel-Aviv, Israel: ACM, 2013, 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368.

[42] Frank McKeen et al. “Innovative instructions and software model for isolated execution.” In: *HASP@ ISCA*. 2013, p. 10. URL: http://css.csail.mit.edu/6.858/2015/readings/intel-sgx.pdf (visited on 01/30/2017).

[43] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment.” In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[44] Ralph C. Merkle. “A Certified Digital Signature.” In: *Advances in Cryptology — CRYPTO’ 89 Proceedings*. Ed. by Gilles Brassard. New York, NY: Springer New York, 1990, pp. 218–238. ISBN: 978-0-387-34805-6. DOI: 10.1007/0-387-34805-0_21. URL: https://doi.org/10.1007/0-387-34805-0_21.

[45] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[46] Erica Portnoy and Peter Eckersley. *Intel’s Management Engine is a security hazard, and users need a way to disable it | Electronic Frontier Foundation*. May 8, 2017. URL: https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it (visited on 10/18/2017).

[47] *Protection from Side-Channel Attacks | Intel® Software*. URL: https://software.intel.com/en-us/node/703016 (visited on 11/06/2017).

[48] O. Purdila, L. A. Grijincu, and N. Tapus. “LKL: The Linux kernel library.” In: *9th RoEduNet IEEE International Conference*. June 2010, pp. 328–333. URL: http://ieeexplore.ieee.org/document/5541547/.

[49] Kaveh Razavi et al. “Flip Feng Shui: Hammering a Needle in the Software Stack.” In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 1–18. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

[50] Christoph Rottermanner et al. “Privacy and data protection in smartphone messengers.” In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2015, p. 83. DOI: 10.1145/2837185.2837202.

[51] Mark D Ryan. “Making Decryption Accountable.” In: *Security Principles and Trust Hotspot 2017* (2017).

[52] Mark Dermot Ryan. "Enhanced Certificate Transparency and End-to-End Encrypted Mail." In: *21st Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014. URL: http://www.internetsociety.org/doc/enhanced-certificate-transparency-and-end-end-encrypted-mail.

[53] Felix Schuster et al. "VC3: Trustworthy data analytics in the cloud using SGX." In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 38–54.

[54] Michael Schwarz et al. "Malware Guard Extension: Using SGX to Conceal Cache Attacks." In: *CoRR* abs/1702.08719 (2017). arXiv: 1702.08719. URL: http://arxiv.org/abs/1702.08719.

[55] Kristoffer Severinsen. *sgx-decryption-service: Client server interaction using gRPC*. original-date: 2017-07-06T16:25:53Z. Nov. 12, 2017. URL: https://github.com/sewelol/sgx-decryption-service (visited on 11/12/2017).

[56] Kristoffer Severinsen, Christian Johansen, and Sergiu Bursuc. "Securing the End-points of the Signal Protocol using Intel SGX based Containers." In: *Security Principles and Trust Hotspot 2017* (2017), p. 1.

[57] Ryan Stedman, Kayo Yoshida, and Ian Goldberg. "A User Study of Off-the-record Messaging." In: *Proceedings of the 4th Symposium on Usable Privacy and Security*. SOUPS '08. Pittsburgh, Pennsylvania, USA: ACM, 2008, pp. 95–104. ISBN: 978-1-60558-276-4. DOI: 10.1145/1408664.1408678.

[58] Marc Stevens et al. "The first collision for full SHA-1." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 190.

[59] Melanie Swan. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[60] Jakub Szefer et al. "Eliminating the hypervisor attack surface for a more secure cloud." In: *Proceedings of the 18th ACM conference on Computer and Communications Security*. ACM. 2011, pp. 401–412. DOI: 10.1145/2046707.2046754.

[61] *Tor Shared Random Subsystem Specification*. URL: https://gitweb.torproject.org/torspec.git/tree/srv-spec.txt (visited on 11/06/2017).

[62] Chia-Che Tsai et al. "Cooperation and Security Isolation of Library OSes for Multi-process Applications." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 9:1–9:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592812.

[63] N. Unger et al. "SoK: Secure Messaging." In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. May 2015, pp. 232–249. DOI: 10.1109/SP.2015.22.

[64] Alma Whitten and J. D. Tygar. "Why Johnny Can'T Encrypt: A Usability Evaluation of PGP 5.0." In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=1251421.1251435.

[65] Y. Xu, W. Cui, and M. Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[66] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems." In: *IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 640–656.

[67] Guy Zyskind, Oz Nathan, et al. "Decentralizing privacy: Using blockchain to protect personal data." In: *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE. 2015, pp. 180–184.