Trusted Execution Environments with Architectural Support: Foundations and Implementation

Barry Biletch

September 6, 2016

Advised by Professors Hugh Lauer and Daniel Dougherty

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see `http://www.wpi.edu/academics/ugradstudies/project-learning.html`

Abstract

In this project, we added support to the OCaml interpreter to use the protections afforded by Intel's Software Guard Extensions (SGX). In particular, this is applied to a cryptographic protocol generator to provide provably secure message exchange even in the face of a malicious operating system.

We argue from a theoretical and experimental perspective that the modifications presented do not alter program behavior and are not vulnerable to attacks on our use of cryptography or implementation issues. We also provide a set of guidelines for developers working with SGX to prevent security bugs.

This project was sponsored by the MITRE Corporation. We would especially like to thank Joshua Guttman and John Ramsdell for sponsoring and advising this project and for providing the environment we used to do the work. Your guidance and advice were invaluable. I couldn't have done it without you.

Contents

1 Problem Statement 5

2 Background 6
2.1 Software Guard Extensions . . . 6
2.1.1 Protection . . . 7
2.1.2 Secret Provisioning . . . 8
2.1.3 Prior Protection Mechanisms . . . 8
2.2 `OCaml` . . . 9
2.3 `CPPL` . . . 9
2.3.1 Example . . . 10

3 Goals and Results 11
3.1 Evaluation . . . 11
3.1.1 `CPPL` (and `OCaml`) in `SGX` . . . 12
3.1.2 No Theoretical Vulnerabilities . . . 12
3.1.3 No Implementation Bugs . . . 12
3.2 Roadmap . . . 12

4 Mathematical Basis for `SGX` 13
4.1 Requirements . . . 14
4.1.1 Enclave Memory Protection . . . 14
4.1.2 Attestation . . . 15
4.2 Symmetric vs Asymmetric Encryption . . . 15
4.3 Computational Complexity . . . 15
4.4 Group Theory . . . 16
4.4.1 Cyclic Groups . . . 17
4.4.2 Elliptic Curves . . . 17
4.4.3 Bilinear Pairings . . . 17
4.4.4 Complexity Assumptions . . . 18
4.5 Attribute-Based Encryption . . . 19
4.5.1 Keys . . . 20
4.6 Zero-Knowledge Proofs . . . 20
4.6.1 Example: Discrete Logarithms . . . 20
4.6.2 Fiat-Shamir Heuristic . . . 21

4.7 CPABE Implementation of Bethencourt *et al.* 21
4.7.1 Access Structures 22
4.7.2 Numerical Attributes 22
4.7.3 Algorithms 22
4.8 Shamir Secret Sharing 26
4.9 Hashing 26
4.10 Hash-Based MACs 26
4.11 BBS+ Signature Scheme 27
4.12 Enhanced Privacy ID (EPID) 28
4.12.1 Group Signatures 28
4.12.2 Algorithms 29
4.12.3 Correctness Proofs 31
4.12.4 Key Sizes 32

5 Implementation 34
5.1 Overview of Code 34
5.1.1 Architecture 34
5.1.2 Modifications to `OCaml` Interpreter 35
5.2 Correctness Argument 36
5.2.1 Functional Correctness 38
5.2.2 Implementation Correctness 38
5.2.3 Currently Viable Attacks 39
5.3 Modifications to `OpenSGX` 40

6 Comments about `SGX` 42
6.1 Early misconceptions about `SGX` 42
6.2 Limitations of `SGX` 43
6.3 Considerations for Writing `SGX`-Compatible Code 44
6.4 `SGX` in Other Architectures 45

A `SGX` Architecture 47
A.1 Enclave Creation 47
A.1.1 `EINIT` Access Control 49
A.2 Enclave Execution 49
A.3 Attestation 50
A.3.1 Measurement 50
A.3.2 Reporting 51
A.3.3 Quoting 51
A.4 Binding 52
A.4.1 Migration 52
A.5 Off-Chip Memory 53
A.6 EPC Management 54
A.6.1 Page Eviction 54
A.6.2 Page Loading 55
A.7 Adding Pages 55
A.8 Multithreading 56

B Associated Files **57**

C Glossary **58**

D Bibliography **61**

Chapter 1

Problem Statement

As cloud computing becomes more prevalent, it is becoming more and more important to ensure the confidentiality of proprietary data when it is stored offsite. In the current cloud model, it is possible for a disgruntled employee or a black hat with a VM exploit to inspect and modify anyone's code and data. In a world where even health care data is migrating to the cloud, this is clearly not an acceptable situation. We need a model in which nobody—not even the cloud providers themselves—can access their clients' sensitive data.

In theory, strict policies can help to keep these unauthorized intrusions at bay. However, people are fallible, falling prey to phishing, social engineering, and outright bribery. Furthermore, company policy can do little against an external attacker who exploits the system.

Another proposed solution is homomorphic encryption, a cryptosystem where it is possible to perform calculations on encrypted data, such that the result is also encrypted [20]. However, modern fully homomorphic encryption schemes are prohibitively expensive, requiring the use of custom-designed homomorphic encryption for each application, which is also impractical, especially for modern applications that can use data in several ways. Additionally, homomorphic encryption does not prevent disclosure of the code itself.

Intel has announced a set of processor extensions known as `SGX` that attempt to solve the problem of privileged access by isolating code and data in a so-called "reverse sandbox" that prevents other software, including the operating system and hypervisor, from looking inside. This project will leverage `SGX` to provide security for code generated by `CPPL`, a programming language for defining cryptographic protocols.

Chapter 2

Background

2.1 Software Guard Extensions

Intel's Software Guard Extensions (`SGX`) [1] are a set of processor extensions, released with the Skylake microarchitecture in August 2015, that allow userspace processes to run securely, even when running under an actively malicious operating system or hypervisor. It ensures that the process's memory is encrypted and integrity protected whenever it leaves the physical processor chip, so that privileged code, and even hardware attacks, cannot obtain meaningful information about a process's secrets or modify the data on which it operates. It also provides attestation capabilities: code running under `SGX` can prove to remote parties that it is "approved" code that has not been modified to leak secrets, and that it is running on a legitimate, non-emulated processor.

A process initially starts in an unprotected state, although it may subsequently request that parts of its address space be placed into a secure *enclave*[1] to protect the code and data stored there. Notably, this means that a malicious actor could modify a process before it entered the enclave, so simply observing that an enclave exists is not proof that it is running legitimate code. Therefore, `SGX` measures the created enclave by hashing its contents, signs the measurement with a key unique and known only to the processor itself, and makes the measurement and signature available to the enclave. By comparing this measurement against a known-good measurement produced locally, a remote party can verify whether the code is trustworthy.

`SGX` does not use a traditional signature scheme, such as RSA, to sign its measurements. Instead, it uses a scheme, called *Enhanced Privacy ID*, with some unusual properties, to preserve users' anonymity. Under EPID, every processor has a unique private key for signing, as expected. However, there is only a single public key, used for verification, shared by every processor produced by Intel. Due to the design of EPID, this means that it is impossible to determine which

[1] The first usage of each term in this paper that may be unfamiliar to readers is italicized, and brief definitions of all of these terms can be found in the glossary.

private key produced a signature, only that *some* private key, authorized by Intel, produced it. The mechanisms by which this is achieved are described in Section 4.12.

SGX also provides binding capabilities to allow enclaves to store secrets for future instances. Clearly, the data cannot simply be stored to disk, as the operating system can intercept the reads and writes or simply read the disk itself. Therefore, we would like to encrypt the data, but what key can we use? Any key that the enclave generates would itself need to be stored somewhere, which raises the original issue again. A remote party could provide the key, but this requires every instance of the enclave to establish a network connection, and requires the remote party to permanently store every key it sends, making it a single point of failure in two distinct ways. To prevent this situation, SGX allows an enclave to request a *binding key*, a symmetric key generated from a combination of the enclave's measurement and a secret value unique and known only to the processor. An enclave can then use the binding key to encrypt secrets that it has been given, allowing the ciphertext to be stored to disk. The key itself does not need to be stored because future instances of the enclave can simply retrieve it from the processor again to decrypt the secrets. Because the binding key is derived from the enclave's measurement, any modified enclaves will receive different binding keys, rendering them incapable of decrypting the secrets.

The binding key is also derived from the individual processor's secret value, so it varies across different physical processors. This prevents one compromised processor from revealing the binding keys for every other processor, but it does introduce logistical difficulties in distributing data. This is discussed further in Section 2.1.2.

2.1.1 Protection

To protect its enclaves, a processor implementing SGX must perform several checks on running code. To begin with, every thread has a flag, stored inside the processor, that indicates whether or not it is executing inside an enclave. If code is running without the enclave flag set, it is assumed to be malicious, so any access to virtual addresses inside an enclave results in a segmentation fault. Similarly, code with the enclave flag set that tries to access a different enclave's memory also receives a segmentation fault.[2] Finally, code executing inside an enclave cannot jump to addresses outside of an enclave, as such locations may contain malicious code that could hijack the enclave access.

To allow enclaves to safely interact with the outside world, SGX provides several instructions for managing control flow across enclave boundaries. These include EEXIT, which safely jumps out of an enclave, replacing the CPU state with a synthetic state that does not leak potentially sensitive information, and ERESUME, which allows untrusted code called from an enclave to return back

[2] However, it is allowed to access both memory within its own enclave and memory that is not in any enclave, as explained below.

into the enclave, restoring the CPU state. Note that ERESUME does not allow specification of the return address; it is stored per enclave thread to prevent malicious code from returning into regions of the enclave that were not expected to be running (e.g. returning to an address partway through an initialization routine, which would initialize only parts of the enclave's memory, likely causing unexpected behavior).

2.1.2 Secret Provisioning

In order to take advantage of SGX's ability to protect a process's data from snooping, it is necessary to provide input in an un-snoopable manner as well. It cannot simply be included in the initial enclave memory, as the data would be visible both in the executable image and in memory before the enclave is set up. The canonical solution to this problem is to have a trusted *provisioning server* that is responsible for sending input once the enclave has already been initialized. The flow of this setup is as follows:

1. The worker process sets up its enclave and randomly generates a keypair from inside the enclave. It sends the enclave's signature and the public key to the provisioning server.

2. The provisioning server verifies the enclave's hash and signature. If they are both acceptable, it encrypts the input to the public key, and sends the ciphertext.

3. Inside the enclave, the worker process decrypts the input and processes it.

The main cost of this solution is that it requires a trusted provisioning server, which must be accessible every time a new worker process is started. In a cloud environment, this means a high-reliability, off-site server, maintained by someone other than the cloud provider.

Another solution is to have each enclave request its input from another enclave instead of from the provisioning server. Doing this, the provisioning server only needs to be running when the first instance is set up (which does not require dedicated hardware; the machine that deploys the code can also serve the initial provisioning). After that, it can give the secret input to other enclaves, eliminating the single point of failure of a provisioning server. The disadvantage to this solution is that additional code needs to be included in the enclave, which increases the attack surface of exploitable bugs.

2.1.3 Prior Protection Mechanisms

To provide protections against emulated CPUs, Intel introduced a unique Processor Serial Number, accessible from software, for every physical processor starting with the Pentium III. This would help a secret provisioning server to identify emulated processors, as only physical processors would have valid, non-duplicate serial numbers. However, this was the only issue that it addressed; it did not provide any of the protections that SGX would come to provide. In particular, it could not prevent malicious snooping in processes' memory spaces or attest to the code that a process is running. The plan was abandoned, and the feature removed, before the Pentium 4, due to the privacy concerns it raised: any software could read the serial number, so it eliminated the possibility of running software anonymously if the vendor decided to track serial numbers.

2.2 OCaml

Caml is a programming language derived from ML[34] with a focus on efficiency. It is primarily functional, but it allows for imperative programming as well. OCaml [32] is an implementation that adds object-oriented concepts and syntax to vanilla Caml to further increase its flexibility.

There are two ways to run OCaml code: native compilation and bytecode interpretation. This paper will focus exclusively on the latter. In this mode of operation, source code is compiled into bytecode, a platform-independent representation that lies between source code and native binaries. Then, at runtime, an interpreter reads the bytecode and executes it, similarly to a high-level VM.

2.3 CPPL

CPPL [23] is a Cryptographic Protocol Programming Language for designing and implementing cryptographic protocols and defining the implications of their execution. It comprises a language that focuses on these semantics and a compiler that produces code to execute the protocols.

Protocols are described in the Dolev-Yao model [17], and every transmission and reception can have an associated formula representing a guarantee and reliance, respectively. Each principal maintains its own *theory* of the world, comprising known associations between principals and public keys, which principals it considers trustworthy, and other factual information relevant to the protocol. When a principal receives a message, it can rely on the sender having said the associated formula[3] to extend its theory, and when it sends a message, it is guaranteeing that the associated formula is true. Therefore, a principal must be sure to verify that the associated formula is in fact true according to its own theory before sending each message.

A protocol is considered *sound* if it is guaranteed that no principal can deduce an incorrect fact from the messages it receives. This includes executions of the protocol in which an attacker has full control over the network and can intercept, modify, and forge messages to and from any participating principals (the typical assumption for protocol analysis). It is assumed that all programs

[3] From this, it can be derived that the formula itself is true if the sender is trusted to tell the truth.

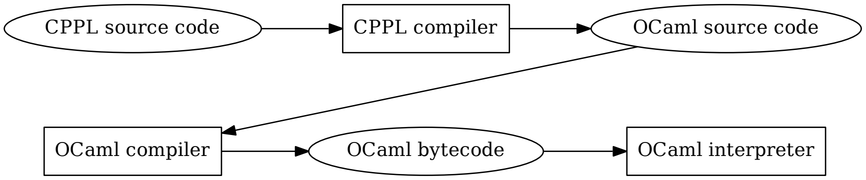

Figure 2.1: Progression from CPPL source to executing binary

running under SGX are sound; if they were not, the protections provided by SGX would become irrelevant.

The current implementation of CPPL is written in OCaml. The compiler takes a description of a cryptographic protocol and produces a working program that implements that protocol. More specifically, it produces OCaml bytecode, which can be run under the OCaml interpreter. This process can be seen in Figure 2.1. Although it seems to be unnecessarily roundabout, the process simplifies the implementation of CPPL and allows the generated programs to take advantage of the optimizations built into the OCaml compiler.

2.3.1 Example

An example CPPL program, borrowed from [23], is provided below. It implements the server side of a Needham-Schroeder-like protocol that allows a client to request stock prices, but will only respond if it can determine that the client will actually pay for the results.

```
proc server (b:text, kb:key) [owns(b, kb)]
  let chan = accept in
  (chan recv {na:nonce, a:text, d:text} kb [true]
    let sk:symkey = new in
    (send [owns(a, ka)] chan {na, sk, b} ka
      (chan recv {sk} kb [says_requests(a, a, b, d)]
        (send [will_pay(a, d); curr_val(d, na, v:text)]
              chan {Data_is v} sk
          return [supplied(a, na, d, v)]))))
```

Chapter 3

Goals and Results

The aim of this project was to investigate the application of SGX to non-trivial, real-world applications, from both theoretical and applied perspectives, by modifying CPPL to produce code protected by SGX.

Originally, it was also intended to modify CPPL to include primitives for the generated programs to perform remote attestation, and to extend CPPL's trust model to include these formulae and to know how to handle their semantics. However, these goals proved to be beyond the scope of the work required for an MQP, and were dropped due to time constraints.

Due to the implementation of CPPL, we decided that it would be easier to modify the OCaml interpreter that executes the compiled CPPL code than to directly modify the code produced by CPPL. This had the added benefit of extending the protections provided by SGX from just CPPL programs to any programs written in OCaml.

Another goal was to approach SGX from a theoretical perspective, and explore its trust architecture was explored. This led to an analysis and review of its underlying primitives and their functions within the system.

3.1 Evaluation

To establish the validity of the project, it was necessary to demonstrate that each of the following goals was met:

- CPPL code (and any OCaml code) runs under SGX
- There are no vulnerabilities arising from the logic of the interpreter
- There are no vulnerabilities arising from the implementation of the interpreter

This was done by a series of tests and mathematical arguments:

3.1.1 CPPL (and OCaml) in SGX

To test that code generated by CPPL is protected by SGX, several protocols were compiled with CPPL, including the Needham-Schroeder-like protocol presented in Section 2.3.1. It was observed by inspection that an enclave was created, and that there was no sensitive data or code (aside from the loader and trampolines) outside of the enclave. Furthermore, it was observed that the protocol executed correctly, leading to the conclusion that the code and sensitive data must exist, and therefore are stored inside the enclave. This process was repeated with several non-CPPL OCaml programs as well, the largest of which was over 600 lines.

3.1.2 No Theoretical Vulnerabilities

The execution of a process was modeled so that its execution inside an enclave could be analyzed mathematically. Using this model, it was proven that it is impossible to mount an attack that causes unintended disclosure of secrets on a sound protocol running inside an enclave.

3.1.3 No Implementation Bugs

We also argued that the low-level implementation cannot be exploited; that is, an adversary cannot maliciously modify code or data outside of the enclave to cause undesired behavior. This analysis focused primarily on passing unexpected or invalid values (such as negative buffer sizes or pointers to sensitive data inside the enclave) across enclave boundaries.

3.2 Roadmap

The rest of this paper will include the results of this project:

- Chapter 4 explains the mathematical concepts and algorithms used by SGX and provides proofs of their security.
- Chapter 5 describes the structure of the OCaml interpreter and the modifications made to it to execute its bytecode under SGX. It also provides correctness proofs of the code, and discusses extant vulnerabilities and how to address them.
- Chapter 6 contains the author's comments on SGX. It addresses commonly held misconceptions, advice for using SGX, and the "rough edges" of the system.
- Appendix A explains in detail the operation and interfaces of SGX. It also describes the challenges faced by SGX, and how they are solved.
- Appendix B lists the files associated with this paper and their purposes.

Chapter 4

Mathematical Basis for SGX

In order to protect enclaves from privileged code and provide its security guarantees, SGX makes heavy use of cryptography. This section introduces the major concepts and cryptographic primitives used by SGX and proves that the latter meet their stated security goals.

There are two primary guarantees that SGX must make in order to meet its security goals. First, it must ensure that malicious code, regardless of permission level, cannot tamper with enclaves. Second, it must be able to demonstrate to users that their code is actually running inside an enclave, and that it is the same code that they produced. To provide these guarantees cryptographically, SGX makes use of a mechanism known as a *hash-based message authentication code* (HMAC) for the former, and a signature scheme called *Enhanced Privacy ID* (EPID) for the latter. This chapter describes these operations, as well as the cryptographic concepts, functions, and principles that they depend on:

- Section 4.1 describes how SGX uses HMACs and EPID to provide its guarantees.

- Section 4.2 briefly introduces *symmetric encryption*, in which encryption and decryption use the same key. In EPID, symmetric encryption is used to encrypt each page of an enclave's memory.

- Section 4.3 defines the notion of computational complexity and explains its importance in designing secure cryptography.

- Section 4.4 reviews the mathematical concept of a *group*, used heavily throughout many of the following sections. Section 4.4.3 defines the related concept of bilinear parings, which play a prominent role in EPID.

- Section 4.5 describes a class of encryption schemes known as *attribute based encryption*, or ABE, in which messages can be encrypted to many recipients, based on sets of attributes instead of individual keys. Section 4.7 introduces one such scheme, published by Bethencourt *et al.*, and proves it correct.

- Section 4.6 describes the concept of a *zero-knowledge proof*: a proof of a fact that does not reveal any additional information. ZKPs are used in EPID to prove that a private signing key is not on a list of revoked keys, without revealing the key itself.

- Section 4.8 introduces *Shamir secret sharing*, a primitive that allows a message to be split into multiple shares, such that no information can be gleaned about the original message without possession of a given number of shares. This is used in the ABE implementation of Bethencourt *et al.* to allow more flexibility in defining the authorized recipients of a message.

- Section 4.9 defines the fundamental cryptographic operation of a *hash*: a mapping from arbitrarily sized data to a fixed size digest. Hashes are used in many places throughout this paper, most notably in EPID during attestation of an enclave's contents.

- Section 4.10 introduces the concept of a hash-based message authentication code, used by `SGX` to ensure the integrity of an enclave's memory.

- Section 4.11 provides a brief introduction to the BBS+ signature scheme, which will be used in EPID to prove security guarantees.

- Finally, Section 4.12 describes the algorithms of EPID, used by `SGX` to attest to the contents of an enclave. It also contains a more direct proof of the correctness of EPID, one of the principal results of this MQP.

4.1 Requirements

4.1.1 Enclave Memory Protection

`SGX` was designed to interfere with the operating system as little as possible; in particular, this means allowing the OS to do its own memory management. As a result, `SGX` does not directly prevent the operating system from reading enclaves' memories.[1] Instead, it prevents the operating system from obtaining meaningful data from any such reads by encrypting each page of the enclave's memory, and only putting encrypted pages in places accessible to the OS. This process is explained in more detail in Appendix A.5.

Similarly, `SGX` does not directly prevent the OS from writing to an enclave's memory; instead, it detects when this has happened and aborts the execution in response. To do this, the processor includes a Message Authentication Code (or MAC) of every page in the enclave and its metadata, including the page's permissions, address (to prevent copying a valid page to an invalid location), and a monotonically increasing counter (to prevent copying an old page back into memory). If there is ever a mismatch between the memory and its MAC, the processor concludes that something has tampered with the enclave.

[1] Part of memory management involves swapping memory to and from disk. Therefore, the OS needs to be able to read (to swap out) and write (to swap in) every process's memory, including enclave memory.

4.1.2 Attestation

Running a process within an enclave for protection does little good if nobody is convinced that this is actually the case. Therefore, it is essential to be able to demonstrate that a process is in fact protected. Doing so is termed *attestation*, explained in detail in Appendix A.3. Briefly, this is achieved by producing a hash of the enclave's contents, and cryptographically signing it using a secret key known only to a legitimate processor. If a verifier trusts the hardware manufacturer to issue secret keys only to chips that function correctly (e.g. with no backdoors to peek inside enclaves or to export secret keys), then it can be deduced that a valid signature demonstrates the existence of a secure process. If the signed hash matches a hash of the enclave's expected contents, computed locally, then it can also be deduced that the secure process is running the correct, unmodified code.

4.2 Symmetric vs Asymmetric Encryption

All historical encryption algorithms, and even many modern ones (such as DES and AES), are *symmetric encryption* algorithms. This means that they use the same key for encryption and decryption (as opposed to *asymmetric* or *public key* algorithms, which have a separate public key for encryption and private key for decryption). Readers interested in the subject are referred to [5, 16, 26, 27] for more information.

4.3 Computational Complexity

When analyzing algorithms, it is desirable to have an objective measure of efficiency that is not dependent on processor speed, compiler optimizations, or other variable factors that do not affect the algorithms themselves. To do this, we define *complexity* as a measure of how quickly the number of operations required by an algorithm grows as the input increases in size. Formally, an algorithm that takes $f(x)$ operations, where x is the size of its input, is in complexity class $O(g(x))$ if

$$\exists x_0 \mid \forall x > x_0,\, \exists \alpha \mid f(x) < \alpha g(x),$$

that is, that beyond a point x_0, $f(x)$ is less than $g(x)$ up to a constant factor [28]. It is in complexity class $\Omega(g(x))$ if

$$\exists x_0 \mid \forall x > x_0,\, \exists \alpha \neq 0 \mid f(x) > \alpha g(x),$$

meaning that beyond a point x_0, $f(x)$ is greater than $g(x)$ up to a constant factor [28].

An important complexity class consists of algorithms that take exponential, or $\Omega(c^n)$, time, for a constant $c > 1$. We aim to make our adversaries' best-case attacks fit into this class because these algorithms can be made to take extraordinarily long amounts of time just by increasing the input size a moderate amount. For this reason, they are also referred to as *computationally infeasible* for sufficiently large inputs.

While it would be ideal to make an adversary's attack literally impossible, such a goal is itself impossible for most algorithms: if it is possible for the adversary to determine if a given solution (e.g. private key) is correct, then he can simply enumerate and check every possibility. Because there are only a finite number of possible solutions that fit in a finite storage size, this algorithm takes a finite amount of time. The next best option, therefore, is to make such an attack take prohibitively long.

4.4 Group Theory

Group theory is a branch of abstract algebra dealing with algebraic structures known as *groups*. A group is a set of values with a binary operator, which we will denote as multiplication (either by juxtaposition (ab) or with a multiplication dot ($a \cdot b$), when necessary for readability), with the following properties:

0. Closure: The binary operation is defined for all pairs of values in the group, and always yields a value in the group.
1. Associativity: $(ab)c = a(bc)$ for all a, b, and c in the group.
2. Identity: There is an element e, called the *identity element*, such that $ae = ea = a$ for all a in the group.
3. Inverse: For every element a in the group, there is an element a^{-1}, called the *inverse* of a, such that $aa^{-1} = a^{-1}a = e$.

Groups are not required to be commutative (that is, it is not necessarily true that $ab = ba$ for all a and b), but groups that are are called *abelian*. Groups may be finite or infinite. Some examples of groups include:

- Real numbers ($\mathbb{R}$) over addition
- Rational numbers without 0 ($\mathbb{Q}^*$) over multiplication
- Integers modulo n ($\mathbb{Z}_n$) over addition
- Integers modulo p without 0 ($\mathbb{Z}_p^*$) over multiplication
- Non-singular $n \times n$ square matrices (for a fixed n) over matrix multiplication
- Permutations over sequencing
- The Klein four-group: $\{e, a, b, ab\}$, where a and b are abstract elements such that $a^2 = b^2 = e$

If the operation on a group is obvious from context, it is often omitted.

4.4.1 Cyclic Groups

For some groups, there is an element g, called a *generator* of the group, such that every element can be represented as g^n for some integer n. These groups are called *cyclic groups*, and are often notated as $\langle g \rangle$.

For instance, the set of integers modulo 5, excluding 0 (normally written as $\mathbb{Z}_5^*$; containing the elements 1, 2, 3, and 4) is a group with respect to multiplication. In this group, 2 is a generator, because every element can be expressed as 2^n for some n:

- $1 = 2^0$
- $2 = 2^1$
- $3 = 2^3$ (because $2^3 = 8 \underset{\text{mod } 5}{\equiv} 3$)
- $4 = 2^2$

Therefore, we could also write this group as $\langle 2 \rangle$.

Cyclic groups are often used in cryptography, and are the basis of the cyclic redundancy checks often used for error detection in hardware devices such as disks and in communication protocols.

4.4.2 Elliptic Curves

Another example of a group can be found in elliptic curves: curves of the form $y^2 = x^3 + ax + b$. In this definition, x and y are often taken from $\mathbb{R}$ or $\mathbb{Q}$, but for cryptography, they are taken from $\mathbb{Z}_p$, for a prime p. For a fixed a and b, points on this curve (together with the point at infinity) form a group. The binary operation is defined as follows:

Definition 1 (Binary operation of elliptic curves)**.** *Find the line passing through both points and find its third intersection point with the curve. The result of the operation is this point mirrored over the x axis.*

Although it is not immediately apparent that this satisfies the group conditions, a proof can be found in [19].

4.4.3 Bilinear Pairings

A bilinear pairing is a function $e : G_1 \times G_2 \to G_T$ that takes two group elements (usually from the same group, although it is not necessary) and produces an element from another group, satisfying the following conditions:

1. Linearity: $\forall a \in G_1, b \in G_2, m, n \in \mathbb{Z} : e(a^m, b^n) = e(a, b)^{mn}$
2. Non-degeneracy: $\exists a, b \mid e(a, b) \neq e_{G_T}$ (where e_{G_T} is the identity element of G_T.

Informally, these conditions mean that exponents can be distributed to either argument and that $e(a, b)$ is not always the identity element. Many texts also require that e be efficiently computable; regardless of definition, it is necessary to be useful in practice.

From this definition, it can be derived that e must be *injective* in each argument; that is, with a fixed element from G_1, no two distinct elements from G_2 (and vice versa) will map to the same element. This result will be used in section 4.4.4.2.

4.4.4 Complexity Assumptions

Classical cryptography relies on the computational complexity of discovering secret keys from public information. To do so, it usually depends on the difficulty of one or more *problems*: constructions that constrain an output value based on the value of inputs. Ideally, cryptographic implementations would use problems that are proven to be computationally infeasible; however, because these are limited in number, problems that are strongly suspected by the community to be computationally infeasible are also often used [29].

4.4.4.1 Discrete Logarithm Problem

The discrete logarithm of a group element β is defined as the smallest positive integer n such that $g^n = \beta$. It is guaranteed to exist if g is a generator because cyclic groups are defined such that every element is expressible in such a form.

For some groups, such as $\mathbb{Z}_n$ with respect to addition, the discrete logarithm can be calculated efficiently. However, there is no known algorithm that works efficiently in arbitrary groups, and the problem is believed to be computationally infeasible in many groups without a known solution [35, 29].

4.4.4.2 Decisional Diffie-Hellman Problem

Another problem defined on groups is the decisional Diffie-Hellman (DDH) problem: given group elements g^a, g^b, and z for randomly selected integers a and b, does $z = g^{ab}$? More formally, the problem asks for a polynomial-time algorithm that can distinguish between (g^a, g^b, g^{ab}) and (g^a, g^b, g^c), where a, b, and c are randomly selected integers, with non-negligible probability. Note that this is stronger than the discrete logarithm problem; given a solution to that, b can be recovered from g^b, and $(g^a)^b = g^{ab}$ can be calculated directly. Therefore, the DDH problem can be solved efficiently for any group in which the discrete log problem can be solved efficiently; however, it is believed to be computationally infeasible for others [9, 29].

It can be easily shown that any group with an efficiently computable bilinear pairing has an efficient solution to the decisional Diffie-Hellman problem: if we can compute e, we can calculate $e(g^a, g^b) = e(g, g^{ab})$, and if this is equal to $e(g, z)$, then we know that $z = g^{ab}$ (because e is injective in each argument).

4.4.4.3 Gap Groups

Some groups, such as $\mathbb{Z}_p^*$, have an efficiently computable solution to the decisional Diffie-Hellman problem, but are believed to have a computationally infeasible discrete log problem. These are called *gap groups* and are useful when dealing with bilinear pairings [29].

4.5 Attribute-Based Encryption

Attribute-based encryption (ABE) [22, 7] is a cryptosystem in which encryption and decryption are performed using a set of attributes and policies, rather than the traditional public and private keys. This allows a user to encrypt to a group of people, without knowing the exact identity of each member. Critically, it is also collusion-resistant, meaning that multiple parties cannot pool their secret keys to decrypt a message that none of them could decrypt alone.

There are two main classes of ABE: key-policy (KPABE) and ciphertext-policy (CPABE). Only CPABE is used in this project, but both are discussed here for completeness and comparison.

In KPABE, messages are assigned a set of attributes at encryption time, and each user's key has an embedded policy that determines what messages it can decrypt based on the attributes with which the message was encrypted. This is well-suited for situations where the types of information that each user should have access to is known beforehand, such as television broadcasting or log sharing. For instance, consider a service that wants to share different parts of its logs with various entities, but needs to ensure that nefarious recipients cannot access unauthorized parts of the logs. Each log entry can be encrypted using a set of attributes (e.g. a sensitivity level, the originating component, and the date), and interested parties can be given keys with a policy that allows them to decrypt only entries that they are authorized to access (e.g. `(nonsensitive or (lowsensitivity and database)) and year2015`, to specify that a user can access any nonsensitive data and low sensitivity data related to the database, but that access expires after 2015).

In CPABE, these roles are reversed: a user's key has a fixed set of attributes, and each message is encrypted using a policy that specifies what attributes are required to decrypt it. This is better suited for situations where the classes of information are more dynamic, such as email. For instance, consider a large company that wants to be able to send emails to groups of employees. Each employee receives a key specifying, e.g., his department, position in the company, and hire date, and messages can be encrypted using arbitrary combinations of these attributes, which may not have been predicted beforehand. For instance, a message regarding benefits for newly-hired part-time employees can be encrypted with the policy `(parttime and hired2015) or hr` to ensure that it reaches only the affected employees and the HR department.

4.5.1 Keys

It is important to realize that ABE keys are not simply large numbers, as is the case for many other cryptographic algorithms. Instead, they are complex data structures with many component subkeys, arranged such that they can only be of use in decrypting messages that they are authorized to decrypt [22]. This means that the algorithm does not simply check the policy and refuse to continue if the given key is unauthorized; it is cryptographically unable to decrypt a message without an authorized key.

4.6 Zero-Knowledge Proofs

A zero-knowledge proof (ZKP) is a proof of a predicate that does not reveal any information beyond the truth of the predicate [21]. They will be used heavily in the formation of EPID signatures (explained in section 4.12), which are used for attestation by SGX.

In general, ZKPs are usually realized by constructing a game related to the predicate. To begin, the prover, called Peggy, commits to some randomly chosen r without revealing its value. Then the verifier, called Victor, requests either r itself or some function of r and the secret portion of the predicate. Thus, Peggy cannot make up an r and a value of the function that are consistent with each other without knowing the secret, although she can make up either one individually. This means that if she does not know the secret, she needs to guess which value to make up when she commits to r, although Victor has not yet decided which one to request. Therefore, if Victor plays this game several times, the odds of Peggy guessing correctly every time become vanishingly small, and she will eventually be unable to provide a valid response.

Zero-knowledge proofs are used in EPID when signing to prove that the signature was created by a key that has not been revoked, while preserving privacy by not revealing which particular key was used.

4.6.1 Example: Discrete Logarithms

Suppose that Peggy wants to prove to Victor that she knows x such that $g^x \equiv \beta$, where g is the generator of a group of prime order p in which the discrete log problem is computationally infeasible. However, she does not want to reveal the actual value of x. This can be achieved by following a protocol described in [13]:

1. Peggy randomly selects $r \in \langle g \rangle$ and sends Victor $C = g^r$.

2. Victor requests either r or $x+r \pmod{p-1}$ from Peggy, chosen randomly.

3. Peggy sends the requested value.

4. Victor verifies the result: r can be verified by recalculating $C = g^r$, and $x + r \pmod{p-1}$ by calculating $g^{x+r \pmod{p-1}} = g^x \cdot g^r = \beta \cdot C$.

Note that if Peggy is lying about knowing x, it is trivial for her to produce a C that will satisfy one of the two requested values. However, when she sends the value of C, she does not know which value Victor will request, so she must choose one of the cases to forge. Because Victor randomly chooses which value to request, there is a $\frac{1}{2}$ chance that Peggy will have guessed incorrectly, revealing herself as a liar. Therefore, if they execute the protocol N times, the chances of Peggy successfully lying by guessing correctly every time is $\frac{1}{2^N}$, which can be reduced very quickly to an arbitrarily small chance by increasing N.

It remains to show that this protocol reveals no information about the value of x. Recall that Peggy can produce a C that is consistent with either request without knowing x. Because this does not require any information unknown to Victor, it is also possible for him to produce such Cs. This lets him form a fake protocol execution, by simply deciding which value to request before producing a C. Clearly, Victor cannot extract any information about x from these fakes, given that he is the one that made them. However, the fake executions are indistinguishable from valid executions with Peggy, so he cannot extract any information about x from Peggy's executions either.

4.6.2 Fiat-Shamir Heuristic

An inconvenient aspect of many zero-knowledge proofs is that they are interactive. This has several implications: first, a proof cannot be published at one time by a prover and verified offline at another time by a verifier. Second, third-party observers will remain unconvinced; they have no assurance that the prover and verifier did not agree beforehand on a sequence of allegedly random numbers to use.

A technique, called the Fiat-Shamir Heuristic, solves this conundrum by allowing Victor to deterministically generate "randomness" to decide which request to make of Peggy [18]. Instead of generating a truly random request, he can take the hash value of Peggy's commitment (C in the above example). Under the random oracle model of hash functions [6], this is a random number, so it cannot be cheated by Peggy; however, it can be regenerated by third parties to verify a transcript. In fact, it allows Peggy to form a complete transcript by playing the part of Victor, as there is no longer any true randomness involved. To perform multiple requests to increase confidence, the ordinal of the request is included in the hashed value to prevent the same request from being issued each time.

The Fiat-Shamir Heuristic is used to make the ZKPs used in EPID noninteractive, so that they can be included as part of a signature.

4.7 CPABE Implementation of Bethencourt *et al.*

In [7], Bethencourt *et al.* describe an implementation of CPABE using bilinear pairings.

4.7.1 Access Structures

The authors begin by defining a tree-based access structure (for an example, see Figure 4.1) and a notion of satisfiability over it. This structure will be incorporated into the ciphertext; a principal will only be able to decrypt the message if his key can satisfy the access structure. The leaf nodes of the tree are individual attributes; they are satisfied for a given key if the key contains the associated attribute. Interior nodes of the access tree have an associated *threshold number*, a quantity representing the minimum number of children nodes that must be satisfied for the parent node to be satisfied (for instance, if the threshold number is 1, the node functions as an "or" gate, and if it is equal to the number of child nodes, the node functions as an "and" gate). Using these definitions, arbitrary access policies can be constructed, with the limitation that possession of an attribute cannot disqualify a key from satisfying a policy. The authors note that this limitation can be solved by doubling the number of attributes: each original attribute gets a positive and a negative variant, and all keys are given exactly one of the two new attributes.

4.7.2 Numerical Attributes

Attributes represent Boolean values, but it is often desirable to assign keys numerical values and restrict access using inequalities (e.g. `level > 5`). This can be achieved by using one Boolean attribute for each possible value and listing every permissible value in the policy, but this solution quickly becomes unwieldy for even moderately sized values. An alternate solution is proposed in [7]: define two attributes for each bit in the value and assign exactly one of the two to each key (e.g. a four bit numerical attribute uses Boolean attributes `0***`, `1***`, `*0**`, `*1**`, `**0*`, `**1*`, `***0`, and `***1`, and a key with the value 9 would contain attributes `1***`, `*0**`, `**0*`, and `***1`). Then, an access structure can be constructed that parallels the decision process for comparisons, as demonstrated in Figure 4.1.

4.7.3 Algorithms

CPABE comprises four algorithms (plus an optional fifth):

Setup This algorithm initializes the system, yielding a master key, used by the private key generator for generating keys for users, and the public parameters, used by anyone to encrypt messages.

GenerateKey This algorithm takes the master key and a set of attributes, and generates a new private key that contains the given attributes.

Encrypt This algorithm takes the public parameters, an access structure, and a message. It produces an encryption of the message that is only decryptable using a key that satisfies the given access structure.

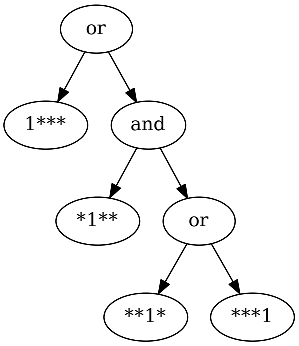

Figure 4.1: Example access tree for numerical attributes ($n \geq 5$)
This tree represents a comparison of a key's numerical attribute to 5. If the key contains the attribute `1***`, then its value has the fourth bit set, so it must be at least 8, which is a sufficient constraint for our ≥ 5 comparison (making it an "or" constraint). Otherwise, we check the third bit. If it is set (meaning it has the `*1**` attribute), then the value must be at least 4, making `*1**` a necessary ("and") constraint. Continuing, possessing a set bit 2 (`**1*`) makes the value at least 6 (sufficient; "or"), and otherwise, a set bit 1 (`***1`) makes the value at least 5 (which is our exact comparison, so there is no need for further checks).

Decrypt This algorithm takes the public parameters, a private key, and an encrypted message. It returns the original message if the private key satisfies the ciphertext's embedded access structure, and fails otherwise.

Delegate This algorithm is not necessary for a functional CPABE system. It takes a private key and a subset of its attributes and produces a new private key that contains the given attributes.

The implementations of these algorithms for the system described in [7] are summarized below.

4.7.3.1 Setup

Choose a bilinear group $\mathbb{G}_0$ with prime order p and generator g, randomly select α and β from $\mathbb{Z}_p$, and choose a hash function H. The public parameters are $\mathbb{G}_0$, g, $h = g^\beta$, $e(g,g)^\alpha$, H, and (for delegation) $f = g^{1/\beta}$. The master key is (β, g^α).

4.7.3.2 GenerateKey

Randomly select r and, for each j in the given set of attributes, r_j from $\mathbb{Z}_p$. The secret key comprises $D = g^{(\alpha+r)/\beta}$ and, for each j, $D_j = g^r \cdot H(j)^{r_j}$ and $D'_j = g^{r_j}$.

4.7.3.3 Encrypt

Define a polynomial q_x for each node x of the access tree as follows:

- For leaf nodes, use the constant polynomial whose value is the parent node's polynomial evaluated at the node's unique index within the parent.
- For interior nodes, set the constant term to the parent node's polynomial evaluated at the node's unique index within the parent (use a value s randomly selected from $\mathbb{Z}_p$ for the root node, which has no parent). Then, randomly select $k-1$ additional points, where k is the threshold number of the node, to obtain a unique $(k-1)$-degree polynomial.

The ciphertext comprises the access tree, $\tilde{C} = Me(g,g)^{\alpha s}$, $C = h^s$, and for each leaf node y in the access tree, $C_y = g^{q_y(0)}$ and $C'_y = H(\text{att}(y))^{q_y(0)}$.

4.7.3.4 Decrypt

Define the Lagrange coefficient $\Delta_{i,S} = \prod_{j \in S, j \neq i} \frac{x-j}{i-j}$ for $i \in \mathbb{Z}_p$ and $S \subseteq \mathbb{Z}_p$. Each node of the access tree can be decrypted (iff it is satisfiable) to obtain $e(g,g)^{rq_x(0)}$ as follows:

- For leaf nodes, if the given secret key does not contain the associated attribute, return an error. Otherwise, let $i = \mathrm{att}(x)$ and return

$$\begin{aligned}\frac{e(D_i, C_x)}{e(D'_i, C'_x)} &= \frac{e(g^r \cdot H(i)^{r_i}, g^{q_x(0)})}{e(g^{r_i}, H(i)^{q_x(0)})} \\ &= \frac{e(g^r, g^{q_x(0)}) \cdot e(H(i)^{r_i}, g^{q_x(0)})}{e(H(i)^{r_i}, g^{q_x(0)})} \\ &= e(g^r, g^{q_x(0)}) \\ &= e(g, g)^{rq_x(0)}.\end{aligned}$$

- For interior nodes, decryption requires the number of decrypted child nodes to meet or exceed k, the threshold number (note that this simply uses Shamir secret sharing (explained in Section 4.8) to share the decrypted value). If there are not enough child nodes that can be decrypted, return an error. Otherwise, decrypt a sufficient number of child nodes and return

$$\begin{aligned}\prod_{z \in S_x} F_z^{\Delta_{i,S'_x}(0)} &= \prod_{z \in S_x} (e(g, g)^{rq_x(0)})^{\Delta_{i,S'_x}(0)} \\ &= \prod_{z \in S_x} (e(g, g)^{rq_{\mathrm{parent}(z)}(i_z)})^{\Delta_{i,S'_x}(0)} \\ &= \prod_{z \in S_x} e(g, g)^{rq_x(i_z)\Delta_{i,S'_x}(0)} \\ &= e(g, g)^{rq_x(0)},\end{aligned}$$

  where S_x is the set of child nodes, S'_x is the set of child node indices, i_z is the index of child node z, and F_z is the decrypted value of node z.

Finally, decrypt the root node of the access tree as A. The message is

$$\begin{aligned}\frac{\tilde{C}A}{e(C, D)} &= \frac{\tilde{C}e(g, g)^{rs}}{e(h^s, g^{(\alpha+r)/\beta})} \\ &= \frac{(Me(g, g)^{\alpha s})e(g, g)^{rs}}{e((g^\beta)^s, g^{(\alpha+r)/\beta})} \\ &= \frac{Me(g, g)^{\alpha s + rs}}{e(g, g)^{(\beta s)((\alpha+r)/\beta)}} \\ &= M.\end{aligned}$$

4.7.3.5 Delegate

Randomly select $\tilde{r}$ and, for each k in the new set of attributes, $\tilde{r}_k$ from $\mathbb{Z}_p$. The new secret key comprises $\tilde{D} = Df^{\tilde{r}}$ and, for each k, $\tilde{D}_k = D_k g^{\tilde{r}} H(k)^{\tilde{r}_k}$ and $\tilde{D}'_k = D'_k g^{\tilde{r}_j}$.

Because this is the only algorithm that depends on the public value of f, delegation can be disallowed by omitting the value from the public parameters.

4.8 Shamir Secret Sharing

Secret sharing is a primitive that divides a secret value into n shares in such a way that at least k shares are required to reconstruct it, but having any fewer provides no information about the secret. One implementation of this is Shamir Secret Sharing [38]:

Construct a polynomial of degree $k-1$, where the constant term is equal to the secret, and the other terms are randomly selected. Evaluate the polynomial at points $\{1, 2, \ldots, n\}$ to form the n shares. To reconstruct the secret, perform an interpolation on k points to obtain the original $k-1$st degree polynomial, and extract the constant term. Note that with any fewer than k points, not only is there not a unique $k-1$st degree polynomial, but there is at least one $k-1$st degree polynomial that contains each constant term, making it impossible to glean any additional information about the secret.

Shamir Secret Sharing is used in Bethencourt *et al.*'s CPABE implementation to allow for nodes that are satisfied when k of their n children are satisfied.

4.9 Hashing

A cryptographic *hash* is a mapping $h : \{0,1\}^* \to \{0,1\}^k$ from arbitrary data to a finite-sized space, termed a *digest*, in an non-invertable way. It has several properties:

Avalanche effect Even a small change in the input should produce a large change in the output (approximately half of the bits).

Collision resistance It is computationally infeasible to find distinct m_1 and m_2 such that $h(m_1) = h(m_2)$; that is, to find two messages with the same hash value.

Preimage resistance Given a digest d, it is computationally infeasible to find an m such that $h(m) = d$; that is, to find a message that matches a given digest.

Second preimage resistance Given m_1, it is computationally infeasible to find an m_2 such that $h(m_1) = h(m_2)$; that is, to find a message that shares the same digest as a given message. (This is similar to collision resistance, but m_1 is fixed, making it a stronger condition.)

Hashes are used extensively for both cryptographic and non-cryptographic purposes, especially when dealing with arbitrarily-sized data.

4.10 Hash-Based MACs

A message authentication code (or *MAC*) is a code to verify the integrity of a message. It is analogous to a symmetric signature because it requires both the signer and verifier to share the same secret. MACs are usually implemented using hash functions; one particular implementation is called "hash-based MAC," or *HMAC*.

A naïve implementation of a MAC with a secret key k for a message m might look like $\mathrm{MAC}(k, m) = h(k||m)$, under the assumption that an attacker would need to know k or be able to find a hash collision in order to forge a MAC. However, this implementation is vulnerable to what is known as a "length extension attack." For many common hash functions, the digest is simply the hash's internal state when it reaches the end of a message. It is therefore possible to reverse a digest into an internal state, and continue hashing as though there were more data. This allows an attacker without knowledge of k to calculate $h(k||m||x) = \mathrm{MAC}(k, m||x)$; that is, to calculate a valid MAC for arbitrary data appended to a previously MAC-ed message. This may be acceptable for certain use cases, but for the vast majority, a better solution is needed.

The next obvious solution is $\mathrm{MAC}(k, m) = h(m||k)$ or even $\mathrm{MAC}(k, m) = h(k||m||k)$ in an attempt to avoid potential attacks on the former. However, there have been proposed vulnerabilities to these constructions as well [36]. To avoid these vulnerabilities, and potential future vulnerabilities, the standardized construction today, known as *HMAC*, is [30]:

$$\mathrm{HMAC}(k, m) = h((k \oplus \mathit{opad}) \mid\mid h((k \oplus \mathit{ipad}) \mid\mid m)),$$

where opad =0x5C5C …5C and ipad =0x3636 …36, both sized to match k. Not only is this construction not vulnerable to any known attacks, but it is resilient against future hash collision attacks that may be discovered against the underlying hash function [4].

HMACs are used in `SGX` for the processor to attest to other local enclaves and to verify that encrypted memory accessible to the operating system has not been tampered with.

4.11 BBS+ Signature Scheme

BBS+ is a signature scheme built on group theory and bilinear pairings [3]. It has three operations:

GenKey This is called beforehand by the signer to generate a private and associated public key.

Let G_1 and G_2 be cyclic groups of prime order p, with a bilinear map e to G_T. Randomly select g_1, h_1, and h_2 from G_1, g_2 from G_2, and γ from $\mathbb{Z}_p^*$. Let $w = g_2^\gamma$. The private key is γ, and the public key is (g_1, g_2, h_1, h_2, w).

Sign This is called by the signer to produce a signature, given the signer's private key, public key, and message $m \in \mathbb{Z}_p$.

Randomly select x and y from $\mathbb{Z}_p$, and let $A = (g_1 h_1^m h_2^y)^{1/x+\gamma}$. The signature is (A, x, y).

Verify This is called by the verifier to check a signature, given the signer's public key and message.

Determine whether $e(A, g_2^x w) = e(g_1 h_1^m h_2^y, g_2)$. The signature is valid if and only if the equality holds. The proof of this follows directly from the previous definitions.

It is shown in [3] that signatures in this scheme cannot be forged by an attacker without the private key.

4.12 Enhanced Privacy ID (EPID)

4.12.1 Group Signatures

A *group signature scheme* is one in which many principals, each with a unique private key for signing, share the same public key for verification. Informally, this means that each member of a group can sign a message on behalf of the entire group, in the same way that any spokesperson of a company can make a statement on behalf of the entire company. Additionally, the scheme is constructed in such a way that a signature verifier cannot determine which principal produced a given signature; nor can it distinguish between multiple signatures by the same principal and signatures produced by distinct principals. It also allows for a special principal, known as a *revocation manager*, to maintain a list of individual keys whose signatures should not be accepted, to deal with users who abuse their ability to sign on behalf of the group.

As used in SGX, each processor is a principal, with its own unique private key, and Intel is the issuer and revocation manager. This means that there is a single public key shared by all Intel processors. Verification requires no private information, so it can be performed by any processor, including those that do not support SGX (because they were produced before SGX was released or by a manufacturer other than Intel).

Although the term *group signature* shares a name with the mathematical concept of a group, it does not refer to the same definition. Instead, it uses the term in the colloquial sense: a collection or set (of principals, in this case). However, many schemes do use mathematical groups to implement group signatures.

Several group signature schemes have been published, each with its own strengths and deficiencies [2, 12, 14, 15]; the scheme used by SGX is Intel's *Enhanced Privacy ID* (EPID) [37]. Ordinarily, group signatures allow for the revocation manager to override anonymity and reveal which principal issued a given signature. A design decision of EPID is *not* to allow this; it does not allow for anyone, even the revocation manager, to do so [10]. Instead, it enhances the *Revoke* algorithm typically used in group signatures. This new *Revoke* algorithm allows the revocation manager to revoke a private key given only a signature that it produced, without knowing the identity of the private key. Technically, this means that EPID is not a group signature scheme. However, it is similar enough that we will continue to refer to it as one in this paper.

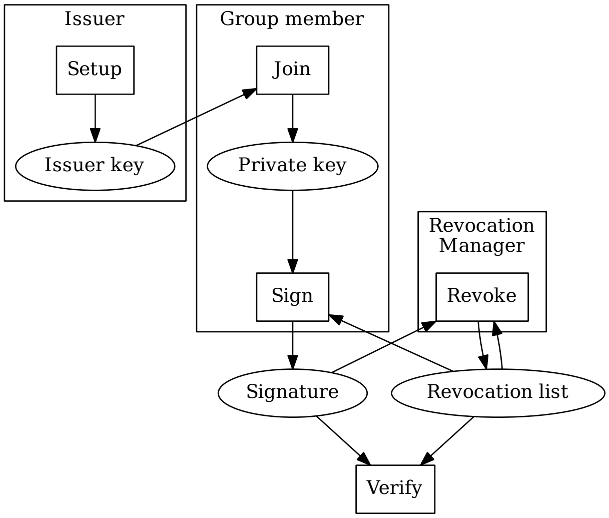

Figure 4.2: Algorithms and Data in EPID
Algorithms are denoted by rectangles, and data are denoted by ovals. Data within a principal's box is private to that principal, while data outside any box can be public.

EPID is used by `SGX` to attest to enclave hashes.

4.12.2 Algorithms

EPID comprises 5 algorithms [10], summarized in Figure 4.2 and explained in detail below:

Setup This is called by the issuer to construct a public key for the group and a private key for the issuer, which allows principals to join the group.

Let G_1, G_2, and G_3 be cyclic groups of prime order p (with generators g_1, g_2, and g_3, respectively), where G_1 and G_2 have a bilinear map e to G_3. Randomly select h_1 and h_2 from G_1 and γ from $\mathbb{Z}_p^*$, and let $w = g_2^\gamma$. The issuer's private key is γ, and the group's public key is

$$(p, G_1, G_2, G_3, g_1, g_2, g_3, h_1, h_2, w).$$

In the case of `SGX`, this was called once by Intel before manufacturing processors. The public key was published, and the issuer private key was stored securely, so that it can be used to generate each processor's private key.

Join This is an interactive protocol[2] between a principal $\mathcal{P}$ wishing to join the group and the group's issuer $\mathcal{I}$. It provides the joining principal with a private key that it can use to sign messages anonymously.

First, $\mathcal{P}$ randomly selects f and y' from $\mathbb{Z}_p$ and sends $T = h_1^f \cdot h_2^{y'}$ and a proof of knowledge of f and y' to $\mathcal{I}$. Next, $\mathcal{I}$ randomly selects x and y'' from $\mathbb{Z}_p$ and sends x, y'', and $A = (g_1 \cdot T \cdot h_2^{y''})^{1/(x+\gamma)}$ to $\mathcal{P}$. Finally, $\mathcal{P}$ takes $y \equiv y' + y'' \mod p$ and calculates $e(A, w \cdot g_2^x)$ and $e(g_1 \cdot h_1^f \cdot h_2^y, g_2)$. If the latter two results are not equal, an error has occurred. Otherwise, $\mathcal{P}$'s private key is (A, x, y, f). (Note that (A, x, y) is a BBS+ signature on f).

In the case of SGX, this is used by Intel to give each processor chip a unique private key at manufacture time.

Sign This algorithm produces a signature on a message, given a group's public key, an unrevoked private key, a revocation list, and the message m.

Randomly select B in G_3 and a in $\mathbb{Z}_p$ and calculate $K = B^f$, $b = y + ax \mod p$, and $T = A \cdot h_2^a$. Produce zero-knowledge proofs of a signature as described below, and, for each revocation in the revocation list, a ZKP that the private key is not the revoked key (Using the ZKP algorithm described in [11]).

To produce a zero-knowledge proof that $\mathcal{P}$ knows x, f, a, and b such that $K = B^f$ and $e(T, g_2)^{-x} \cdot e(h_1, g_2)^f \cdot e(h_2, g_2)^b \cdot e(h_2, w)^a = e(T, w)/e(g_1, g_2)$:

1. Randomly select r_x, r_f, r_a, and r_b from $\mathbb{Z}_p$ and compute $R_1 = B^{r_f}$ and $R_2 = e(T, g_2)^{-r_x} \cdot e(h_1, g_2)^{r_f} \cdot e(h_2, g_2)^{r_b} \cdot e(h_2, w)^{r_a}$.
2. Compute the Fiat-Shamir Heuristic $c = H(\text{pk}, B, K, T, R_1, R_2, m)$, where H is a publicly-known hash mapping into $\mathbb{Z}_p$ and pk is the group's public key.
3. Calculate $s_x = r_x + cx$, $s_f = r_f + cf$, $s_a = r_a + ca$, and $s_b = r_b + cb$, and produce $\sigma = (B, K, T, c, s_x, s_f, s_a, s_b)$.

Under SGX, this is used by the quoting enclave to attest to the contents of another enclave, without revealing which processor chip the enclave is running on.

Verify This algorithm determines whether a signature was produced by an unrevoked private key associated with a given public key. It does not require any private keying material.

Given a signature $\sigma = (B, K, T, c, s_x, s_f, s_a, s_b)$ and list of zero-knowledge proofs, compute $\hat{R}_1 = B^{s_f} \cdot K^{-c}$ and

$$\hat{R}_2 = e(T, g_2)^{-s_x} \cdot e(h_1, g_2)^{s_f} \cdot e(h_2, g_2)^{s_b} \cdot e(h_2, w)^{s_a} \cdot (e(g_1, g_2)/e(T, w))^c.$$

[2] Interactivity means that both the joining principal and the issuer can keep secrets from each other, so that even the issuer does not know the new signer's private key.

Verify that $c = H(\text{pk}, B, K, T, \hat{R}_1, \hat{R}_2, m)$, and that each zero-knowledge proof is valid. Note that the signature may have been made with an older revocation list, so it should also be confirmed that the ZKPs cover a sufficiently current revocation list.

In `SGX`, this is called by the user requesting the enclave to be run to verify an attestation. Note that it will, in general, not be used on the same computer that is running the enclave.

Revoke This algorithm allows the revocation manager to establish a set of keys (identified by signatures that they have produced) that are no longer allowed to issue signatures on behalf of the group. It produces (or extends) a revocation list that can be used in the Verify algorithm to check if a signature was produced with a revoked key.

Given a signature $\sigma = (B, K, T, c, s_x, s_f, s_a, s_b)$ and a (possibly empty) revocation list, add (B, K) to the revocation list.

Note that this algorithm does not require any private data, so it can be performed by anyone. Therefore, the true revocation manager must sign its revocation lists (using a traditional signature algorithm, such as RSA) to prevent attackers from producing rogue revocation lists. It also means that trusted third parties can produce their own, equally usable, revocation lists independently of the "real" revocation manager.[3]

For `SGX`, this will be run by Intel if the private key from a processor is ever compromised, to prevent attackers from using the key to falsely attest to the contents (or existence) of enclaves. The revocation list will be published; because it is not secret, there is no need for special treatment.

4.12.3 Correctness Proofs

For a signature to verify, it must have a valid zero-knowledge proof of a signature, which can only happen if all parameters to the hash function are the same. Therefore, we must show that $\hat{R}_1 = R_1$ and $\hat{R}_2 = R_2$ for a valid signature; all other parameters are given in σ. (An implicit proof of Theorem 1 appears in [10], but we provide an explicit proof here.)

Theorem 1. *Correct signatures will verify:* $\hat{R}_1 = R_1$ *and* $\hat{R}_2 = R_2$*.*

$$\hat{R}_1 = B^{s_f} \cdot K^{-c} = B^{r_f + cf} \cdot (B^f)^{-c} = B^{r_f + cf - cf} = B^{r_f} = R_1$$

[3] This could be useful if, for example, Intel stops supporting `SGX` in the future or refuses to acknowledge the compromise of a particular key. It could also be used to prevent enclaves from being created on particular processors for reasons other than key compromise.

$$
\begin{aligned}
\hat{R}_2 &= e(T,g_2)^{-s_x} \cdot e(h_1,g_2)^{s_f} \cdot e(h_2,g_2)^{s_b} \cdot e(h_2,w)^{s_a} \cdot (e(g_1,g_2)/e(T,w))^c \\
&= e(T,g_2)^{-r_x-cx} \cdot e(h_1,g_2)^{r_f+cf} \cdot e(h_2,g_2)^{r_b+cb} \cdot e(h_2,w)^{r_a+ca} \cdot (e(g_1,g_2)/e(T,w))^c \\
&= (e(T,g_2)^{-r_x} \cdot e(h_1,g_2)^{r_f} \cdot e(h_2,g_2)^{r_b} \cdot e(h_2,w)^{r_a}) \cdot \\
&\qquad (e(T,g_2)^{-x} \cdot e(h_1,g_2)^{f} \cdot e(h_2,g_2)^{b} \cdot e(h_2,w)^{a})^c \cdot e(g_1,g_2)^c/e(T,w)^c \\
&= R_2 \cdot (e(Ah_2^a,g_2)^{-x} \cdot e(h_1,g_2)^f \cdot e(h_2,g_2)^{y+ax} \cdot e(h_2,g_2^\gamma)^a \cdot e(g_1,g_2)/e(Ah_2^a,g_2^\gamma))^c \\
&= R_2 \cdot (e((Ah_2^a)^{-x},g_2) \cdot e(h_1^f,g_2) \cdot e(h_2^{y+ax},g_2) \cdot e(h_2^{a\gamma},g_2) \cdot e(g_1,g_2)/e((Ah_2^a)^\gamma,g_2))^c \\
&= R_2 \cdot e((Ah_2^a)^{-x} \cdot h_1^f \cdot h_2^{y+ax} \cdot h_2^{a\gamma} \cdot g_1/(Ah_2^a)^\gamma, g_2)^c \\
&= R_2 \cdot e((Ah_2^a)^{-x-\gamma} \cdot h_1^f \cdot h_2^y \cdot h_2^{ax+a\gamma} \cdot g_1, g_2)^c \\
&= R_2 \cdot e(A^{-x-\gamma} \cdot h_2^{-ax-a\gamma} \cdot h_1^f \cdot h_2^y \cdot h_2^{ax+a\gamma} \cdot g_1, g_2)^c \\
&= R_2 \cdot e(A^{-x-\gamma} \cdot h_1^f \cdot h_2^y \cdot g_1, g_2)^c \\
&= R_2 \cdot e((g_1 \cdot T \cdot h_2^{y''})^{(-x-\gamma)/(x+\gamma)} \cdot h_1^f \cdot h_2^y \cdot g_1, g_2)^c \\
&= R_2 \cdot e((g_1 \cdot h_1^f \cdot h_2^{y'} \cdot h_2^{y''})^{-1} \cdot h_1^f \cdot h_2^y \cdot g_1, g_2)^c \\
&= R_2 \cdot e((g_1 \cdot h_1^f \cdot h_2^y)^{-1} \cdot h_1^f \cdot h_2^y \cdot g_1, g_2)^c \\
&= R_2 \cdot e(e_{G_1}, g_2)^c \\
&= R_2 \cdot e_{G_3}^c = R_2 \cdot e_{G_3} = R_2
\end{aligned}
$$

□

This proves that signing with a private key will produce correct signatures; however, it must also be shown that a signature cannot be forged without a valid private key. This can be done as in [10]:

Theorem 2. *Signatures are unforgable under the strong Diffie-Hellman assumption.* Suppose there exists an algorithm that can forge an EPID signature in polynomial time. We use this algorithm to forge a BBS+ signature:

Given a BBS+ public key (g_1, g_2, h_1, h_2, w), choose a group G_3 of order p with generator g_3 to make an EPID public key $(p, G_1, G_2, G_3, g_1, g_2, g_3, h_1, h_2, w)$. Use the EPID forging algorithm to produce a signature, and extract A^*, x^*, y^*, and f^* from it. From the definition of the EPID signature, we know that

$$e(A^*, g_2^{x^*} w) = e(g_1 h_1^{f^*} h_2^{y^*}, g_2).$$

Therefore, (A^*, x^*, y^*) is a BBS+ signature on f^*. However, the BBS+ signature scheme is unforgable under the strong Diffie-Hellman assumption, so our initial assumption that there exists an algorithm that can forge an EPID signature must be wrong. □

4.12.4 Key Sizes

Even when a signature scheme is cryptographically secure, it needs to use keys that are sufficiently large to resist brute-force attacks. For EPID, "sufficiently large" means that p must be 170 bits to achieve 80-bit security, and 256 bits to achieve 128-bit security [10]. This means that private keys must be 681 bits (86 bytes) or 1025 bits (129 bytes), respectively, and that signatures are 1363 bits (171 bytes) or 2051 bits (257 bytes) respectively.

Chapter 5

Implementation

5.1 Overview of Code

5.1.1 Architecture

For this project, the `OCaml` interpreter was modified to execute under `SGX`. To do this, it transparently creates an enclave and runs the bytecode inside, as shown in Figure 5.1

Running a `CPPL` protocol under `SGX` requires several steps. First, the `CPPL` source must be compiled into `OCaml` code using the unmodified `CPPL` compiler. Next, the `OCaml` code must be compiled into `OCaml` bytecode, an intermediate representation used by the interpreter to maintain platform independence. Finally, this bytecode can be fed to the modified `OCaml` interpreter, which will load itself and the bytecode into an `SGX` enclave before executing.

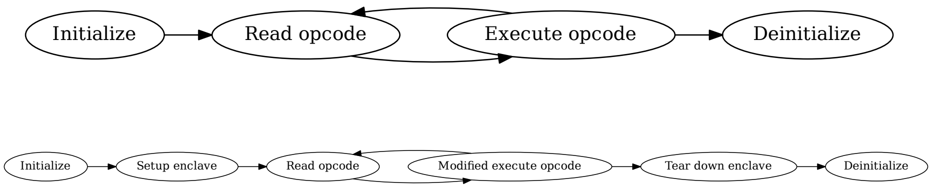

Figure 5.1: High-level operation of vanilla `OCaml` interpreter (above) and modified `OCaml` interpreter (below)

5.1.2 Modifications to `OCaml` Interpreter

Ordinarily, the `OCaml` interpreter follows a simple execution flow, as shown on the top of Figure 5.1. First, it initializes its memory and allocates its resources. Next, it enters its main loop, where it reads the next opcode, decodes and executes it (conceptually using a switch statement or lookup table), increments its program counter, and repeats. Finally, when an opcode terminates the loop, it releases its resources and exits.

Several modifications to the this flow were necessary in order to enable `CPPL` programs to run under `SGX`. First, the main function of the interpreter, which calls the initialization and deinitialization routines, was wrapped in an additional initialization/deinitialization layer for `SGX`. This allows it to safely load itself into an enclave before the program begins execution. Second, the code for the opcodes that performed system calls were replaced with a *trampoline* version (explained below) and a corresponding implementation outside of the enclave. Finally, the `unix` module, used for disk and network access, was linked statically into the interpreter. Under the vanilla interpreter, modules are loaded dynamically as needed; however, doing so under `SGX` would present a security concern. Therefore, expected modules (in this case, only `unix`, although others can be easily added) are linked statically, and dynamic linking is emulated.

5.1.2.1 Trampolines

Recall that code executing inside an enclave is not allowed to access external resources. If it attempts to execute system calls directly, the processor will force a segmentation fault to prevent the operating system from gaining access to the enclave's memory. To work around this, the interpreter running inside the enclave uses what is known as a *trampoline* [25]: unprotected code outside the enclave that coordinates with the protected code to perform the system call. It works as follows (shown in Figure 5.2):

- The code inside the enclave copies the necessary parameters to a fixed address in unprotected virtual memory.[1]

- The enclave code executes the `EEXIT` instruction to exit the enclave and transfer control flow to the trampoline located in unprotected memory.

- The trampoline reads the parameters from the same fixed address in unprotected memory (which it can do because it is outside the enclave), and uses them to execute the system call.

- When the system call returns, the trampoline copies the returned data to another fixed address and returns control flow back to the enclave using the `ERESUME` instruction.

[1] Recall that code executing in an enclave has normal read/write access to memory outside the enclave.

- Finally, the enclave resumes execution and copies the result from the second fixed address in unprotected memory into protected memory and continues its regular execution.

For instance, the interpreter makes use of the open system call to be able to read files. However, it cannot be called directly in an enclave. Thus, a stub version, called sgx_open, for use inside the enclave was written. This stub accesses a known memory location for stub communication located outside the enclave, where it stores a constant FUNC_OPEN to indicate the requested system call, the file path, and the flags used to open the file. It then exits the enclave to the trampoline address, which checks the requested system call and dispatches to a non-enclave sgx_open_tramp function. This function reads the parameters from the stub communication memory and performs the actual system call. It copies the result back into the stub communication memory when the system call finishes, and then returns to the main trampoline function. The trampoline function reenters the enclave where it left off inside sgx_open, which reads the system call result from the stub communication memory and returns it to the caller.

Note that this trampoline scheme closely resembles traditional function calls, and can be considered to be simply a different calling convention. Whereas traditional function calls store their parameters on the stack or in registers, trampoline calls store parameters in a fixed memory location, and while traditional function calls use CALL and RET to call and return from functions, trampoline calls use EEXIT and ERESUME, respectively. Aside from these differences in implementation, however, the underlying mechanisms remain the same.

5.2 Correctness Argument

Simply executing code under SGX is not sufficient to guarantee that it will operate correctly; we must also argue that it will act in the same manner as its unprotected counterparts. We must also establish that it cannot be manipulated by malicious modification of code or data outside of the enclave.

Fundamentally, the core logic of the interpreter is unchanged. The changes are in initialization and handling of system calls. The interpreter's initialization, which only sets up and executes the enclave, does not affect the interpreter's functionality; it is an augmentation, but it does not replace any logic. System calls, in the ordinary case, are unaffected. The changes add several layers of indirection, but the functionality is, by design, unchanged. Therefore, we can conclude that under benign circumstances, the modified interpreter functions identically to the original.

However, what about the adversarial circumstances where an attacker is trying to exploit the system? For our purposes, it is desirable to deviate from the unmodified interpreter's behavior; we want to avoid leaking information or misbehaving. We assume that SGX itself meets its stated security goals: that it will protect against unauthorized memory reads by properly encrypting all enclave memory and against unauthorized writes by correctly authenticating

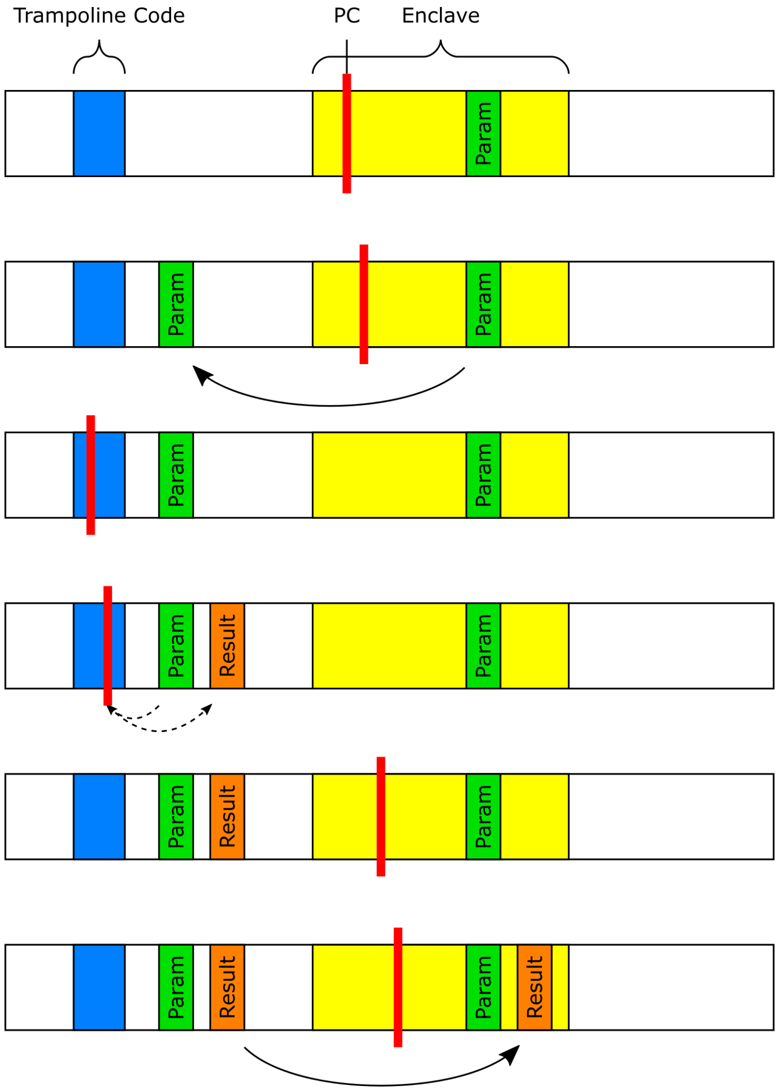

Figure 5.2: Trampoline execution
The first picture in this figure represents the enclave's state before using the trampoline; each of the following pictures directly matches the corresponding bullet point in Section 5.1.2.1.

the encrypted data (for details, see Appendix A). This means that while the program is executing inside the enclave, the attacker can do nothing to cause it to misbehave or to leak secrets. However, useful programs must perform some form of I/O (usually in the form of network access), and the execution flow is required to leave the enclave to perform system calls. Could this behavior be exploitable? There are two ways that this could be attacked: at the functional level, where the attacker exploits bad program logic, or at the implementation level, where the attacker exploits the low-level implementation.

5.2.1 Functional Correctness

The only external I/O performed by CPPL programs is network access to communicate with other principals participating in the protocol. Therefore, this is the only avenue of attack on functional correctness for a malicious actor: modifying the data that the program sends or receives. This is equivalent to the attacker being able to interfere with the program's network connections. However, CPPL is already designed around a model where attackers have full control of the network. Thus, the weak points of the SGX implementation are covered by CPPL's cryptographic guarantees; there is no way for an attacker to extract secrets or cause the program to misbehave.

5.2.2 Implementation Correctness

Assuming that SGX is implemented properly, the only access that an attacker has to an enclave is to the unprotected memory that is mapped into the same address space as the enclave and the results of system calls executed from outside the enclave.

All processing for system calls occurs outside the enclave: either in the operating system or in the enclave's process when in non-enclave mode. Therefore, any exploits against the TCP stack, filesystem, the implementation of any other syscall, or any libc wrappers can only be used to attack resources that the attacker already has access to. Because these exploits, if they exist, do not gain an attacker any leverage, they do not need to be further considered in the security analysis of the enclave. This is by the design of SGX: it forces the programmer to partition the code to remove as much as possible from the code that needs to be verified to place trust in an enclave's correctness.

Inspection of the code reveals that the modified OCaml interpreter code does not directly access any memory outside the enclave, and that the only time the libsgx code, which is also included in the enclave, accesses unprotected memory is during trampoline calls. Therefore, it suffices to validate only this code, as any exploits against the modified interpreter would need to target either the trampoline code or an existing bug in the OCaml interpreter (which would be outside the scope of this project). Simple inspection reveals that the stub implementation, when copying the results of a system call into enclave memory, does read the length from outside the enclave for system calls that return such blocks of data. However, it compares this untrusted length to the known size of the buffer that it is copying into, and signals an error if it is exceeded. Therefore, if it copies data from outside the enclave, it is guaranteed to stay within the allocated buffer, preventing buffer overflow attacks.

5.2.3 Currently Viable Attacks

There are still three obvious attacks that are viable against most `CPPL` programs running under the current `SGX OCaml` interpreter:

- Influencing the source of randomness
- Modifying the executed bytecode
- Changing the theory files

Each of these can be solved with additional modifications to the interpreter, outlined below. Doing so is a possible direction for future work on this project.

5.2.3.1 Source of Randomness

All but the simplest `CPPL` programs require random data to operate (for creating keys, choosing nonces, etc.). An attacker that can influence the source of this data can violate the program's assumption that it is actually random, and cause it to misbehave. This could cause it to create keys known to the attacker, choose already used nonces, or even reveal already created keys [40]. `CPPL` uses a library called `cryptokit` [31] for its cryptographic operations, which currently uses randomness from `/dev/random` This is a resource provided by the operating system, which means that it can be controlled by an attacker who controls the operating system. This can be solved by modifying `cryptokit` to use an alternate, more secure source of randomness. Such a source can be found in the processor, in the form of the `RDRAND` instruction [33]. This instruction uses a hardware random number generator in the processor itself, based on thermal noise and compliant with NIST SP800-90A, B, and C, FIPS-140-2, and ANSI X9.82 [33], and can be used from inside an enclave, requiring no trust in the operating system.

5.2.3.2 Executed Bytecode

The `OCaml` interpreter, by nature of being an interpreter, does not contain the code that it will execute. Instead, it reads the bytecode for the program from a separate file. Because disk operations are controlled by the operating system, this is not secure: an attacker could modify the code that is executed, doing anything from simply revealing sensitive key material to introducing subtle bugs into the software. The correctness of the protocol is irrelevant in this case, because the protocol as the author intended it is never actually executed.

This can be solved in several ways:

- The code to be executed can be embedded in the enclave's memory space when the enclave is created, and the interpreter modified to read bytecode from memory, rather than from a file. Any modifications to the bytecode before the enclave is created would prevent the enclave from attesting to the expected value, so users would know that it had been tampered with.

- Similarly, a hash of the bytecode file can be embedded in the enclave's memory space, and the interpreter modified to verify the hash after reading in the file. Any modifications to the bytecode file would cause it to have a different hash, so the interpreter can reject the program.

- The interpreter can be modified to accept the bytecode (or a hash of it) over the network, encrypted to a keypair generated inside the enclave. In this case, the interpreter would attest to its contents, so that the user can reject it if it has been modified, and then the user would send the encrypted bytecode. Because the keypair was generated inside the enclave, the attacker cannot access the private key, and without the private key, he cannot undetectably modify the bytecode.

5.2.3.3 Theory Files

`CPPL` uses an inference engine at each step of the protocol to decide if it trusts the other principals involved enough to continue executing. This trust is ultimately rooted in a *theory file* containing a set of formulas that it is allowed to rely upon as axiomatically true. Similarly to above, if an attacker can intercept and modify read operations from that file, she can introduce false axioms (e.g. "Anything Mallory says is true") that cause the program to misplace its trust and thus misbehave.

The first two solutions from above can be similarly applied to this attack as well:

- The theory file can be included in the initial enclave image, and the interpreter modified to use the theory already present in memory instead of reading it from disk.

- A hash of the theory file can be included in the enclave's memory space, so that the interpreter can verify the integrity of the file that it reads (and abort execution if the hashes do not match).

Notably, the third solution from above cannot solve this problem: in order for the program to know that the provided theory file is correct, the sender would have to prove that the file is trustworthy, but in order to do that, the program already needs to have a theory file to root its trust in.

5.3 Modifications to `OpenSGX`

During the course of this project, we discovered several bugs with OpenSGX. Most of them were fixed during the course of its development, except for one bug, dealing with adding new pages to the enclave [8]. As this bug blocked development and was not addressed during this project, we modified OpenSGX to work around it. The patch is attached to this report and listed in Appendix B.

Chapter 6

Comments about SGX

6.1 Early misconceptions about SGX

Because it is such a new technology, there are many misconceptions about SGX and its capabilities. In this section we make some observations, each of which dispels some commonly held misconceptions about SGX.

SGX does not directly prevent the operating system from tampering with an enclave. As explained in Section 4.1, SGX needs to allow the operating system read/write access to enclaves' memories, so that it can swap pages to and from disk. To prevent malicious use of these permissions, SGX cryptographically protects each page, so that reads cannot obtain useful information, and writes can be detected by SGX, and the program aborted.

The operating system can modify the contents of an enclave before it is initialized. All enclaves are initialized with data originating outside the enclave. Therefore, it is possible for the operating system to alter the initial contents of the enclave (most importantly, this includes the code that it will execute). To address this, SGX allows for *attestation*, so that the end user can tell if such modifications have occurred, and refuse to use the results of the enclave.

SGX does not hide the code that runs inside an enclave. For similar reasons to the previous point, the operating system can see the code that goes into an enclave (although it can no longer read it after the enclave is fully initialized). This can be mitigated by only putting a *loader* in the initial enclave. Once the enclave is initialized, the loader can obtain the real code to be executed from a remote server, encrypted to a key generated inside the enclave.

SGX does not enable malware that cannot be removed or reverse engineered. Creating an enclave requires support from the operating system, so any attempts to put malware into an enclave can be rejected by the operating system. This can be implemented with anything from a simple user prompt or blacklist/whitelist to hooks that allow antivirus software to inspect the code being put into the enclave.

6.2 Limitations of SGX

Although SGX has been shown to be a cryptographically secure system, this is not the only consideration that should be taken into account. In particular, it has several design decisions that make it unnecessarily cumbersome to use, without increasing security. This section addresses some of the rough edges of SGX, and how they could be improved. Please note that these comments are solely the opinions of the author, and should not be interpreted as absolute facts or the opinions of Worcester Polytechnic Institute or the MITRE Corporation.

- There is currently no mechanism for an enclave to securely interact with a computer's peripherals. This is unnecessary for most servers, but critical for personal computers, where code running in an enclave needs to interact directly with the user. At present, the enclave needs to receive its input and send its output through the operating system, which it cannot trust. Consider the case of running a program to cryptographically sign email in an enclave: a malicious operating system could fake input to the program, asking it to sign a message never authorized by the user.

  To solve this, SGX would need to add a mechanism for the operating system to grant an enclave direct access to a particular peripheral, in a way that cannot be intercepted or modified (this would probably require non-trivial hardware changes as well). Then, each driver, usually run within the operating system, could be moved into its own enclave with access to only the associated peripheral. This allows programs running in an enclave to receive input from another enclave, which they can verify is actually receiving input directly from the user.

- There is currently no mechanism for enclaves to specify a "contract" for how they can interact with the rest of the system. Doing so could mitigate the effects of bugs in the code running in an enclave by aborting the program if it attempts to perform an unexpected operation. For instance, code running in an enclave should be able to voluntarily drop access permissions for unprotected memory and restrict the allowed targets for an EEXIT instruction.

- There is currently little compiler support for SGX. While this is not a problem with SGX itself, it is essential for widespread use and adoption of SGX. There are many possible ways in which compilers and toolchains can make working with SGX easier, the most important of which include:

  - Allow the programmer to annotate functions and variables that should be included in the enclave, and detect at compile time attempts to call code or access variables across an enclave boundary. Require an explicit declaration of intent to do so, to prevent accidentally accessing unprotected memory.

  - Automatically generate code for trampolines.

- Although SGX does not directly reveal what code is currently executing in an enclave, it can leak approximate memory access patterns to a cleverly malicious operating system. If the OS swaps every page of an enclave out of memory, it knows that if the enclave takes a page fault on page x, then it is currently executing an instruction somewhere in page x. Any further page faults are due to either memory accesses executing instructions on different pages. The operating system can then immediately swap the pages back out and repeat the process to get a fairly detailed picture of the enclave's memory access pattern. It can also distinguish between writes and reads/executes by comparing the ciphertext of the swapped pages before and after execution. This allows a side-channel attack on the program running inside the enclave, which could potentially reveal secret information from the enclave, given enough time.

  Unfortunately, there is no simple way to address this problem. (While it is technically possible to require all pages of an enclave to be swapped in and out as a single unit, and to increase the counter on each page regardless of if it was written to, this is prohibitively inefficient and effectively eliminates the operating system's ability to do effective memory management.) The best solution seems to be to ensure that developers are aware of the possibilities for side-channel leakage, and trust them to mitigate the issue (by ensuring that variables often accessed together are on the same page, designing side-channel resistant algorithms, deciding that such an attack would be too expensive relative to the information it would extract, etc).

- As explained in Appendix A.1, adding a page to an enclave during initialization requires one EADD for the 4 KiB page, followed by exactly 16 EEXTENDs, one for each 256 byte region of the page. Because it is incorrect to execute more or fewer instructions, or in any other order, there is no need to separate the logical operation into two (let alone 17) different invocations. Although compiler support will mean that the programmer does not have to interact with SGX at the instruction level, this is still an unnecessary wart on the implementation of SGX.

6.3 Considerations for Writing SGX-Compatible Code

When writing code that will be secured with SGX, special care must be taken to ensure that the program will satisfy the assumptions of SGX and be secure under its new threat model.

- Isolate all calls to resources outside the enclave (e.g. network access) with proper abstractions. Not only does this make testing easier, but it also reminds the programmer that these are no longer native calls, and cannot be trusted.

- Ensure that programs are executing sound protocols. At the very least, this means that all network (and disk) accesses must be encrypted and/or authenticated, but it may also impose additional restrictions based on the function of the code and malleability of the protocol. Use the appropriate key(s) for each access (communication with third parties should use a newly-generated symmetric key, storage should use the binding key, etc.).

- Design the software architecture to cleanly split into secure and non-secure portions. The secure portion should contain everything that needs to be kept secret, and nothing more: parsing and formatting, for instance, should be moved to the non-secure code when possible. Minimizing the code that is included in the enclave (as well as the interface between the secure and non-secure portions) reduces the amount of effort needed to audit the enclave, and decreases the surface area of attacks.

- If the overall program has multiple components, design the architecture with multiple enclaves. For instance, a server could have the database in one enclave, the TLS/SSL layer in another, and the program proper in a third. Done properly, this increases auditability, decreases surface area of attacks, and reduces the impact of a successful compromise.

- When analyzing an architecture, consider code outside the enclave not to exist, as it can be modified by an attacker. Assume that the attacker can interact directly with the enclave via any interface provided. If this bypasses any protections and allows the attacker to change the information that the enclave processes or that another principal sees, that functionality should be moved into the enclave, or the protocol should be changed to prevent such a scenario.

6.4 `SGX` in Other Architectures

Currently, `SGX` is an Intel-only technology. However, other manufacturers may decide to adopt it into their own processors if it gains a large enough developer base.

AMD, in addition to Intel, produces processors with the `x86_64` architecture. If AMD implements `SGX` in its own processors, it is exceedingly unlikely that Intel would give access to its issuer private key. This means that AMD would need to act as its own issuer and generate its own private key, resulting in a different public key. Users would then need to determine which manufacturer's processor their enclaves are running on, and check the attestations against the corresponding public key.

More interesting is SGX support in architectures other than x86_64. Because AMD's processors run the same code as Intel's, it would be likely to implement SGX in a compatible way. However, other architectures are not bound to being strictly compatible with Intel's SGX. This gives them the flexibility to change interfaces to suit their own use cases and to simplify usage for their users. They would be free, for instance, to implement any of the changes suggested in Section 6.2, without regard for compatibility with the current implementation of SGX.

Appendix A

SGX Architecture

SGX adds two instructions to the Intel architecture: ENCLS and ENCLU, for supervisor (operating system) and usermode uses respectively. Each instruction has several leaves, representing distinct logical instructions, that are selected based on the value of the EAX register.

A.1 Enclave Creation

Enclaves, although used by unprivileged processes, can only be requested by the operating system. Therefore, a platform-specific API, which will not be described here, must be used for the application to request an enclave.

To begin creating an enclave, the operating system executes the ECREATE leaf of the ENCLS instruction. This initializes an SGX Enclave Control Structure (SECS), used to store the processor's information about the enclave. It then executes the EADD leaf to copy a 4 KiB page into the EPC, followed by 16 EEXTENDs, each of which cryptographically measures a 256 byte section of the enclave. Failure to properly measure the enclave with exactly 16 EEXTENDs means that the final measurement of a correctly constructed enclave will not match the expected value (if the attacker tries to incorrectly construct an enclave) or that an attacker can modify the unmeasured data without detection (if the expected value is constructed incorrectly). The operating system repeats the EADD and EEXTEND instructions until it has added all necessary pages to the enclave; then, it executes the EINIT leaf to finalize creation and prepare the enclave for use. This prevents additional pages from being added to the enclave and causes the processor to store the enclave's hash in the SECS, so that it can be retrieved and attested to later, as described in Appendix A.3. This process is shown pictorially in Figure A.1.

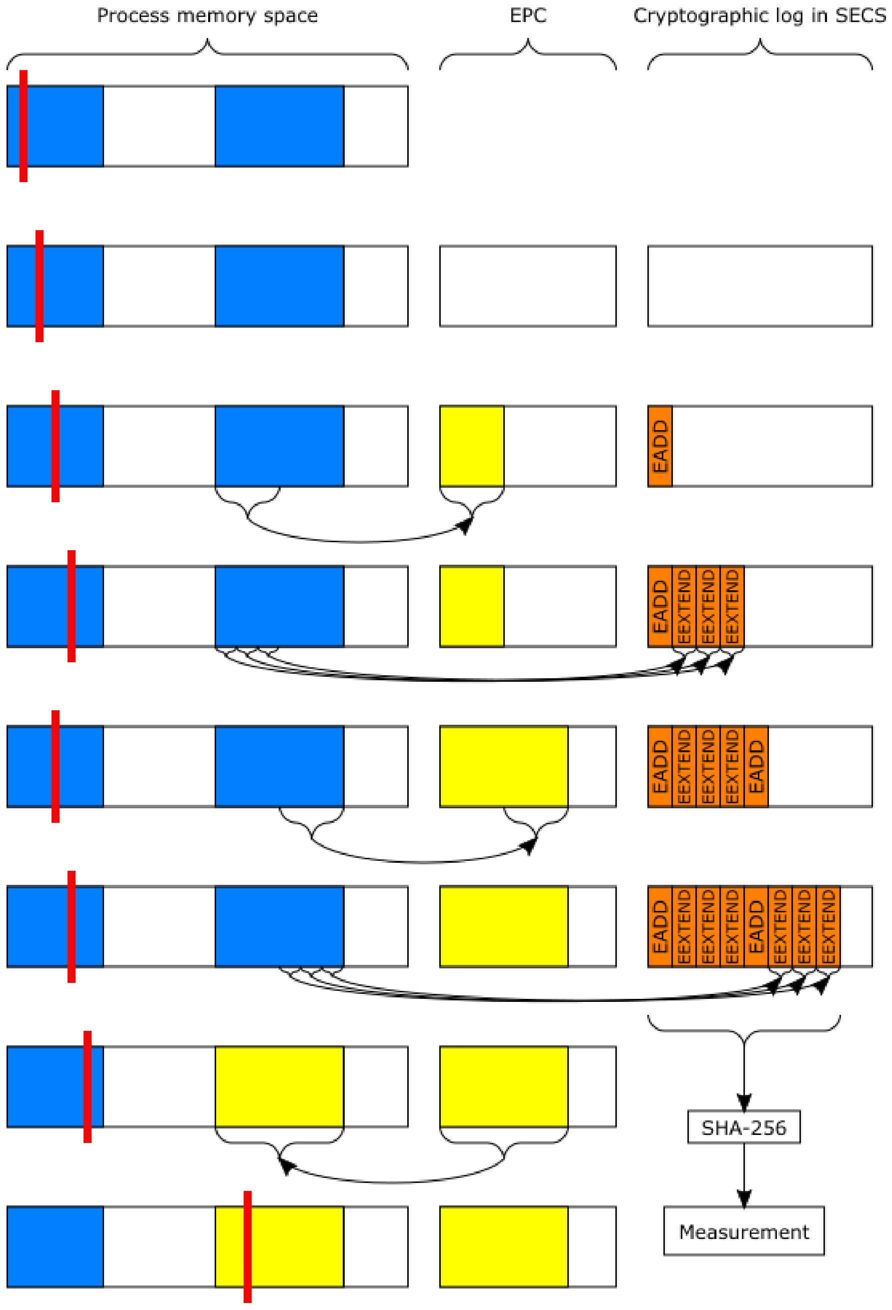

Figure A.1: Enclave creation
The first picture shows the process's memory space before creating the enclave. The next shows the results of the OS executing `ECREATE` to initialize the SECS. Next shows the `EADD` call to encrypt a page into the EPC. After that is the 16 `EEXTEND`s to measure the page. These two steps are repeated until the entire enclave has been measured. The next picture shows the result of the `EINIT` call: the measurement is finalized and the encrypted enclave pages are mapped into the process's memory space. Finally, the last picture shows the process after executing `EENTER` to jump into the enclave.

A.1.1 EINIT Access Control

There is one additional step to enclave creation that was not mentioned in the previous section. Intel has included a check in the EINIT instruction that allows a special enclave, called the *launch enclave*, to enforce arbitrary access controls on the EINIT instruction. To implement this check, the launch enclave is provided with the measurement, author, and other metadata about each enclave that is created.[1] If it determines that a given enclave should be allowed to proceed, it generates a MAC on the metadata using the processor's *launch key*, producing an *EINITTOKEN*.[2] When the enclave is being finalized, the EINIT instruction checks for a valid EINITTOKEN, and fails if one is not provided.[3]

Intel has been criticized for adding this element to SGX for a variety of reasons:

It provides no additional security. Because ECREATE, EADD, EEXTEND, and EINIT can only be executed by privileged software, the operating system already has the ability to perform access control for access to enclaves, by refusing to execute any of these instructions on behalf of the program. Furthermore, it has a superset of the information available to the launch enclave, so it can make equally, if not more, informed decisions.

It takes control away from the computer owner. The launch enclave is provided by Intel, and cannot be replaced. This wrests control from the actual owner of the computer and gives it to a third party, violating the idea that a computer's owner should have ultimate control over what runs on it.

It allows Intel to control enclaves. The launch enclave is written by Intel, and it cannot be overridden by the owner of the system. This allows Intel to restrict access to enclaves for non-technical (i.e. business) reasons, which violates SGX's apparent philosophy of not restricting what can be done to or with an enclave.

A.2 Enclave Execution

To execute code in the enclave, the usermode process uses the EENTER leaf of the ENCLU instruction. This marks the process as running in enclave mode and jumps to an enclave-specified address. When the enclave wants to call or return back to unprotected code (to perform system calls, for instance), it executes the

[1] The exact mechanism by which this information is transferred is left unspecified in the documentation.

[2] The launch key, like most other keys used by SGX, is requested from the processor using EGETKEY. However, the launch enclave, identified by its signature from Intel, is the only enclave given permission to access it. This prevents other enclaves from acting as their own launch enclaves, and allowing enclaves that would not otherwise be allowed.

[3] This check is bypassed for the launch enclave itself, as requiring an EINITTOKEN for the launch enclave would present a bootstrapping issue, and the launch enclave can be implicitly trusted, because it has been signed by Intel.

EEXIT leaf, which clears the enclave mode flag and jumps to a specified location in unprotected code. If permitted by the enclave, the process may return back into the enclave by calling the ERESUME leaf. If the processor encounters a fault or exception while inside an enclave, it performs an Asynchronous Enclave Exit (AEX), which stores the current registers on the stack, loads dummy registers known as *synthetic state* to avoid leaking information, drops the enclave flag, and jumps to a specified address in unprotected code. Once the event has been handled, the process can call ERESUME, which will additionally restore the enclave's original registers.

To prevent unauthorized access to the enclave, the processor enforces additional restrictions on memory access based on the access type, the location of the memory, and the enclave flag. If these are violated, a segmentation fault is produced. These restrictions are summarized in Table A.1.

	R/W Outside	X Outside	R/W Inside	X Inside
Non-enclave mode	Y	Y	N	N
Enclave mode	Y	N	Y	Y

Table A.1: Memory restrictions due to enclaves

A.3 Attestation

Running code in an enclave can sometimes be useful on its own, but most of the utility of SGX comes from being able to prove to third parties that their code is actually running in an enclave. This process is known as *attestation.* It takes place in three parts under SGX: first, measuring the enclave during initialization, and once the enclave is running, reporting (attestation to other enclaves running the same processor) and quoting (attestation to remote parties).

A.3.1 Measurement

As discussed previously, an enclave is initialized with the ECREATE instruction, and its contents are specified with several invocations of EADD and EEXTEND. The measurement of the enclave, finalized with the EINIT that starts protection of the enclave, is therefore specified to be a cryptographically secure log of these instructions, in execution order, specifying the exact arguments to each. Under SGX, this is realized with a SHA-256 hash. Each EADD instruction adds one block of data to the hash, specifying the instruction executed (EADD), the virtual address of the page added, and the permissions on the page. Each EEXTEND instruction adds 5 blocks of data to the hash: the first specifies the instruction executed (EEXTEND) and the virtual address of the data it is measuring, padded to the block length, and the remaining four contain the value of the data. Once the enclave is initialized, the EINIT instruction finalizes the computation of the hash, and stores the result in the enclave's SECS.

Because this process documents everything done to initialize the enclave (combined with the requirements on when EADD and EEXTEND can be executed), the calculated value can be considered to be a digest of the enclave itself, extending all of the properties of hashes to the measurement. In particular, this means that it is computationally infeasible to form a new enclave with the same measurement as an existing enclave.

A.3.2 Reporting

Reporting is the process of proving the contents of an enclave to a separate enclave on the same processor. This is useful for applications that are split among multiple enclaves (e.g. a server that has the database in one enclave, the TLS stack in another, and the business logic in a third), but is also used as a component of quoting, explained below.

To request a report, an enclave P executes the EREPORT instruction, specifying 64 bytes of arbitrary data[4] and the measurement of a target enclave V that wants to verify the result. In response, the CPU generates a MAC of the given data and P's measurement, using the signing key associated with V.

To verify the report, V can request a copy of its own signing key from the processor with the EGETKEY instruction, and use this key to verify the MAC on the report. If the MAC is valid, it knows that there is an instance of P running on the same processor, and that the public key given in the report came from within the enclave.

It is necessary for P's key to be included in the report for validation to prevent replay and man-in-the-middle attacks. Without this inclusion, an attacker could create two instances of P: one protected by an enclave, and one unprotected. Once the instance in the enclave produces a report, the attacker could proceed as though the report came from the unprotected instance. As V cannot directly observe P's contents, it is unable to distinguish between the two instances, and would proceed to communicate with the unprotected instance of P under the mistaken assumption that P is secure. Including a public key for P in the report prevents this attack, because the unprotected instance of P would generate a different key. This would make it unable to successfully decrypt messages from V, and thus unable to proceed normally.

A.3.3 Quoting

One of the main guarantees of SGX is to allow for remote attestation, or *quoting*. To do this, we need to introduce another enclave, called the *quoting enclave*. This enclave, provided by Intel, runs locally on the processor, so that any running enclave can attest to it, using a report as described above. It is unique in that it can request the EPID private key from the processor, but is otherwise a normal enclave, with no additional access to other enclaves. Its sole function

[4] Although there is no specified structure to this data, it is in practice necessary for it to be a public key generated within the enclave that can be used to communicate with P. This rationale will be explained in more detail below.

is to accept incoming reports, verify them, and then sign the measurement and public key from the report using EPID. This signature is passed back to the requesting enclave, which can then send it to a remote party to prove its identity. The security of this protocol derives from the same principles as reporting; the only difference is the type of signature, and thus who can verify it.

A.4 Binding

Almost all non-trivial programs need to store state between runs, and programs run in enclaves are no exception. However, enclaves cannot simply write their state to disk, as that would be vulnerable to modification by an attacker. Therefore, `SGX` provides a mechanism, known as *binding*, that allows a program running under `SGX` to securely encrypt secrets, using a key accessible only within the enclave. To obtain this key, the enclave uses the `EGETKEY` instruction, requesting its *binding key*. This key is unique to the enclave and processor, so that no other enclaves can decrypt stored information; not even another instance of the same enclave running on a compromised processor.

A.4.1 Migration

When the software running in an enclave is updated, the enclave's measurement changes. This means that if the binding key were derived directly from the measurement (in addition to other data), future versions of the enclave would be unable to access existing bound data after an update. To address this, `SGX` requires each enclave to have an *author*[5] specified at creation time. To do so, the author's public key and signature of the enclave's expected measurement are passed to `EINIT` when the enclave is initialized. At that point, the processor verifies that the signature is valid, and that it matches the measurement actually produced by the enclave. If either of these conditions is not met, the enclave is not created, and the call fails. Otherwise, the author's public key is included in the SECS as another attribute of the enclave (and thus included in any attestations), and this key, rather than the measurement, is used to derive the binding key.

This means that an enclave can decrypt secrets that were bound by a previous version (with a different measurement), as long as they were both signed by the same author. At first, this seems dangerous: there is no verification that the author key provided at enclave creation is the correct one, so an attacker can simply strip off the original author signature and replace it with his own. Then, if he modifies the enclave, he can produce a signature for the new enclave using the new key. Alternatively, he can leave the original enclave unmodified

[5] This usage of the term "author" in `SGX` is slightly incongruous with the traditional English meaning: if a single developer produces multiple independent enclaves, they should each have a separate author for the purposes of `SGX`. The only time two enclaves should be signed with the same author key is if one is an updated version of the other; a better term might therefore be "program identity."

but write another enclave, signed with the same key, that requests the binding key and uses it to decrypt the first enclave's secrets.

The prevention for both of these attacks comes during attestation. If an enclave tries to attest to a provisioning server, and the server sees that it has an unexpected measurement or author, it should not provision any secret information to the enclave. Therefore, although the attacker can access any secrets that the enclave tries to bind, the enclave will never gain access to any sensitive information in the first place[6], making the attack pointless. This again aligns with SGX's philosophy of not using access controls to restrict what can be done, but rather using cryptographic means to render any attacks useless.

A.5 Off-Chip Memory

To prevent a malicious operating system, hypervisor, or even an attacker with physical access to the machine that runs an enclave from reading protected secrets, SGX encrypts every page of an enclave's memory. At system initialization time, a region of physical memory, known as the *Enclave Page Cache* (EPC), is set aside to be used exclusively for backing the virtual memory of enclaves. Ordinary software, including the operating system, is forbidden from accessing the EPC directly, although the operating system can map pages from it into the address space of enclaves.

Whenever enclave data is flushed from the processor's cache, it is not simply written back to a mapped page main memory (as non-enclave data would be). Instead, it is encrypted (with a key unique to the enclave and processor) and written to the mapped page in the EPC. This extra security measure appears unnecessary at first, as the operating system is prevented from accessing these pages, but it is useful in preventing physical attacks that go beyond the operating system. For instance, malicious peripherals with direct memory access [39] and cold boot attacks [24] can both retrieve data from RAM without any requirements from the operating system. There is no feasible way to prevent these attacks without crippling the computer for the common case, so instead, SGX includes the EPC to render these attacks unable to extract useful information.

Accordingly, when the processor takes a cache miss on data in an enclave, the processor must decrypt the data that it obtains from the EPC.

Because an enclave's memory is encrypted with a key unique to a specific processor, it is not possible to migrate an enclave from one processor to another. This is most relevant in a cloud computing environment, in which processes are often moved between physical machines to address shifting load and hardware failures. SGX interferes with this type of management, which will force cloud providers to change management techniques. It is likely that programs using SGX will be required to be designed to be shorter-lived and composable, or to be able to be interrupted at any time and restarted.

[6] Note that this guarantee, along with several others provided by SGX, requires the enclave to be running a sound protocol.

A.6 EPC Management

There are four types of pages in the EPC:

Regular Most pages in the EPC are regular pages, used for backing the virtual memory of enclaves.

SECS There is one SGX Enclave Control Structure page per enclave instance, used to store information about the enclave (including its size, memory location, measurement, etc.).

TCS Each enclave instance has one or more Thread Control Structure pages, one for each thread allowed in the enclave simultaneously.

VA Each Version Array page holds information on up to 512 other pages that have been evicted from the EPC, in order to ensure their integrity when reloaded. Unlike the other page types, VA pages are not associated with an enclave.

A.6.1 Page Eviction

A design goal of SGX was to allow the operating system to continue managing the memory of all processes, even those with enclaves. To swap out a page from an enclave's memory, the operating system must first inform the processor that the page is not to be used. It does this with the EBLOCK instruction, which sets the page as blocked to disallow the mapping from being added to a TLB. However, this alone does not remove the mapping from a TLB if it is already cached. To do so, it must send an inter-process interrupt (IPI) to each logical processor executing in the enclave, forcing it to perform an asynchronous exit from the enclave and flush its TLB. Once the page's mapping has been removed from all TLBs, the page itself can be evicted to main memory with the EWB instruction. This instruction takes the address of the enclave page to be evicted, the address of another page to store the encrypted data, the address of a buffer for protected metadata about the page, and a VA slot for secure metadata. Finally, now that the page has been evicted from the EPC, the operating system can swap the encrypted page to disk, just like it would any other page.

A.6.1.1 Additional Requirements per Page Type

Regular pages can be evicted as described above with no additional restrictions.

SECS pages are required to run an enclave, so each can only be evicted when all pages belonging to its associated enclave have been evicted first.

TCS pages are required for a thread to be in an enclave, so they can only be evicted when they are not in use.

VA pages can be evicted without additional requirements, but still require their own version numbers to be stored in another VA page. This leads to a hierarchical structure of VAs when many pages are evicted.

A.6.2 Page Loading

When an enclave tries to access a page that has been evicted, it will take a page fault, just like any other process. To continue its execution, the operating system must load the evicted page back into the EPC. To do this, it must first swap the encrypted page back into main memory, as it would a non-enclave page. It must also ensure that the enclave's SECS page[7][8] and the VA page containing the slot used during eviction are already in the EPC, which may require loading those back first. It then executes either the `ELDB` or `ELDU` instruction[9] to load the encrypted page back into the EPC, and recreates the mapping in the enclave's memory space.

A.7 Adding Pages

Although the original `SGX` specification did not provide a mechanism to add pages to an enclave after initialization, Intel has since released `SGX2`, an extension to the original `SGX` specification, which includes support for doing so.

To add additional pages, the enclave makes a request to the operating system through a platform-specific API (note that this requires the enclave to exit to unprotected code). The operating system then calls `EAUG` to allocate a page from the EPC and associate it with a given address in the enclave. Finally, when the enclave resumes execution, it calls `EACCEPT` to acknowledge the addition of the page. Until it does so, the page is inaccessible, and the enclave functions identically to how it would have if the page had not been allocated.

Although adding pages to the enclave after initialization may seem like it requires the enclave to re-attest to its validity, the necessity of the `EACCEPT` call renders this unnecessary. Because the enclave will be unchanged if it does not explicitly acknowledge the new page, any malicious additions to the enclave by an untrusted operating system cannot affect the enclave's execution. Therefore, any additions that modify the enclave must have been acknowledged, which could only have been done by previously authenticated code. Thus, the `EACCEPT` call allows the augmented enclave to extend a chain of trust as to its own validity, rooted in the initial measurement.

[7] If the page is being restored because the enclave took a page fault, the SECS will already be in the EPC, as it is needed for the enclave to execute. However, if the operating system is reloading the page for other reasons, this may not always be the case.

[8] As VA pages do not belong to an enclave and thus do not have an associated SECS, this requirement does not apply when restoring VA pages.

[9] Both of these instructions perform the same task, but `ELDU` unblocks the page after loading, while `ELDB` leaves it blocked, requiring the operating system to unblock it before it can be used in an enclave.

A.8 Multithreading

Another feature added in SGX2 is the ability to have multiple threads executing concurrently in an enclave. To use this functionality, the process must explicitly allow each thread by setting aside one page of memory to be used as a *Thread Control Structure* (or TCS) for the thread's metadata.[10] These pages are loaded into the enclave normally, as described in Appendix A.1, with the PT_TCS page type flag to indicate that they are not to be treated as regular enclave memory.

After the enclave is created, it can be entered normally, as described above. When doing so, one of the parameters to the EENTER leaf is the index of the TCS to use. Upon entry, the processor marks the TCS as "busy" to prevent another thread from being able to use the same TCS concurrently. This ensures that there will never be more threads than expected, and that each thread will be executing the expected code.

Note that this does not prevent against denial-of-service attacks from the operating system. It is still possible, for instance, for the operating system to refuse to schedule one of an enclave's threads; recall that SGX guarantees that if the process runs to completion, it will do so correctly, but makes no guarantees as to whether it will complete. This makes multithreaded SGX programming even more difficult than regular multithreaded programming already is: there can be no assumption that threads will execute at even approximately the same rates.

[10] The most important piece of metadata in the TCS is OENTRY, the location in the enclave where the thread should resume (or begin) execution.

Appendix B

Associated Files

The following files were submitted with this report, and are a part of this project:

ocaml-sgx.tar.gz The code for the `OCaml` interpreter modified to run under `SGX`. Based on `OCaml` version 4.02.3.

MD5:	`53778b9c2e79255065f3904488c6720f`
SHA1:	`d1bbc2f9117ffc456d33dc6acd3ec95fb857868e`
SHA256:	`834637505f4300d31e7ea0080175c68c0248148cf679e49dd3e95a12d1fbf6ec`

opensgx.diff A patch to OpenSGX that increases the size of the EPC and initial enclave heap to work around its bugs. Produced from commit `8feb343`.

MD5:	`227507cb7c37e3b8ba194787ef70f736`
SHA1:	`3b1379336790a595dcc0fa3170ff99f97655a2f1`
SHA256:	`9468027a2ba520b0df61f31612c140a970c932dd3e9f0b4bf6f3e01deffd1ccf`

Appendix C

Glossary

ABE Attribute-based encryption; a cryptosystem that uses attributes and policies in place of public and private keys

Abelian group A commutative group, that is, a group where the order in which terms are multiplied does not matter

Attestation A cryptographic signature of a measurement

Binding key A key produced by `SGX` from an enclave's measurement and the processor's secrets; used for persistent data storage

Computational complexity A measure of how quickly the number of operations taken by an algorithm grows as the input size increases

Computationally infeasible Taking at least $O(c^n)$ time, for a constant $c > 1$

`CPPL` Cryptographic Protocol Programming Language; a tool that verifies security properties of and generates code for cryptographic protocols annotated with trust constraints

Cyclic group A group in which all elements can be expressed as g^n, for a fixed g, called a generator of the group

Digest The result of a cryptographic hash

Gap group A group in which the decisional Diffie-Hellman problem can be solved efficiently, but calculating discrete logarithms is computationally infeasible

Generator See *cyclic group*

Dolev-Yao model A model for analyzing cryptographic protocols where only the usage of cryptographic primitives is considered, not their implementations

Enclave A region of memory protected by `SGX`

EPC Enclave Page Cache; a region of off-chip memory used to back enclaves' virtual memories

EPID Enhanced Privacy ID; a cryptographic signature scheme where each of a set of principals can sign messages anonymously

Group A set of elements with a binary operator, subject to constraints described in Section 4.4

Group signature scheme A signature scheme where any member of a group can sign a message using its own private key, but an ordinary verifier cannot determine which member produced a given signature

HMAC A MAC implemented with hashes

Identity element The unique element of a group that, when multiplied by another term, produces that same term

Group inverse

MAC Message Authentication Code; a cryptographic value that verifies the integrity of given data

Measurement A cryptographic hash of an enclave used to identify the code that it is running

Penetrator A malicious actor in a cryptographic protocol who can arbitrarily insert, remove, and process any message

Principal A participant in a cryptographic protocol

Provision Give secrets to a program running in an enclave that is already running, so that they are not visible in the enclave's initial state

Provisioning Server An external server that provisions necessary secrets only to properly attested enclaves

Random oracle model A model for analyzing cryptographic protocols in which hash functions are modeled as producing a truly random output for each input

Revocation manager The principal in EPID that can mark keys as untrustworthy

SECS `SGX` Enclave Control Structure; a structure used by the processor to hold information about an enclave

`SGX` Software Guard Extensions; a set of instructions from Intel to allow trusted computing under untrusted privileged code

TCS Thread Control Structure; a page of metadata allocated for each thread to be allowed in an enclave

Theory The set of facts that a particular principal knows to be true

Trampoline Unprotected code used by an enclave to perform system calls

Quoting enclave An enclave provided by Intel that uses local attestations to produce remote attestations

Appendix D

Bibliography

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[2] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Annual International Cryptology Conference*, pages 255–270. Springer, 2000.

[3] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-taa. In *International Conference on Security and Cryptography for Networks*, pages 111–125. Springer, 2006.

[4] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.

[5] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403. IEEE, 1997.

[6] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[7] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE, 2007.

[8] Barry Biletch. malloc broken with small allocations. `https://github.com/sslab-gatech/opensgx/issues/48`, 2016.

[9] Dan Boneh. The decision diffie-hellman problem. In *Algorithmic number theory*, pages 48–63. Springer, 1998.

[10] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID from bilinear pairing. Cryptology ePrint Archive, Report 2009/095, 2009. `http://eprint.iacr.org/`.

[11] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Advances in Cryptology-CRYPTO 2003*, pages 126–144. Springer, 2003.

[12] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*, pages 410–424. Springer, 1997.

[13] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. An improved protocol for demonstrating possession of discrete logarithms and some generalizations. In David Chaum and WynL. Price, editors, *Advances in Cryptology EUROCRYPT 87*, volume 304 of *Lecture Notes in Computer Science*, pages 127–141. Springer Berlin Heidelberg, 1988.

[14] David Chaum and Eugène Van Heyst. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 257–265. Springer, 1991.

[15] Lidong Chen and Torben P Pedersen. New group signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 171–181. Springer, 1994.

[16] Hans Delfs and Helmut Knebl. *Introduction to Cryptography: Principles and Applications (Information Security and Cryptography)*. Springer, 2007.

[17] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[18] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology-CRYPTO86*, pages 186–194. Springer, 1986.

[19] Stefan Friedl. An elementary proof of the group law for elliptic curves. *The Group Law on Elliptic Curves*, 1998.

[20] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[21] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 291–304. ACM, 1985.

[22] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. Acm, 2006.

[23] Joshua D Guttman, Jonathan C Herzog, John D Ramsdell, and Brian T Sniffen. Programming cryptographic protocols. In *Trustworthy global computing*, pages 116–145. Springer, 2005.

[24] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[25] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2016.

[26] Lars R Knudsen. Practically secure feistel ciphers. In *International Workshop on Fast Software Encryption*, pages 211–221. Springer, 1993.

[27] Lars R Knudsen and Matthew Robshaw. *The block cipher companion.* Springer Science & Business Media, 2011.

[28] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April 1976.

[29] Neal Koblitz and Alfred Menezes. Intractable problems in cryptography. In *Proceedings of the 9th Conference on Finite Fields and Their Applications. Contemporary Mathematics*, volume 518, pages 279–300, 2010.

[30] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[31] Xavier Leroy. Cryptokit. `https://forge.ocamlcore.org/projects/cryptokit/`, 2014.

[32] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 4.03.* 2016.

[33] John M. Intel digital random number generator (drng) software implementation guide. `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`, 2014.

[34] Robin Milner. *The definition of standard ML: revised.* MIT press, 1997.

[35] Andrew M Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in cryptology*, pages 224–314. Springer, 1984.

[36] Bart Preneel and Paul C Van Oorschot. On the security of two mac algorithms. In *Advances in CryptologyEUROCRYPT96*, pages 19–32. Springer, 1996.

[37] Carlos Rozas. Intel software guard extensions (Intel SGX). `http://www.pdl.cmu.edu/SDI/2013/slides/rozas-SGX.pdf`.

[38] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[39] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.

[40] Serge Vaudenay. The security of dsa and ecdsa. In *International Workshop on Public Key Cryptography*, pages 309–323. Springer, 2003.