View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@WPI

Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

October 2016

Trusted Execution Development: Designing a Secure, High-Performance Remote Attestation Protocol

Alexander David Titus
Worcester Polytechnic Institute

Seth Daniel Norton
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

Titus, A. D., & Norton, S. D. (2016). *Trusted Execution Development: Designing a Secure, High-Performance Remote Attestation Protocol*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1145

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WPI

MITRE

Trusted Execution Development

Designing a Secure, High-Performance Remote Attestation Protocol

October 13, 2016

Written by

Seth Norton

Alex Titus

Project Sponsored by the MITRE Corporation

Project Faculty Advisor - Craig Shue

Project Sponsor Advisor - Joshua Guttman

Project Sponsor Advisor - John Ramsdell

Abstract

Intel Software Guard Extensions (SGX) are a Trusted Execution Environment (TEE) technology that allow programs to protect execution process and data from other processes on the platform. We propose a method to combine SGX attestation with Transport Layer Security (TLS). Doing so will combine guarantees about the program, runtime environment, and machine identity into a normal TLS handshake. We implemented a basic server using SGX/TLS and provide performance details and lessons learned during development.

Contents

1 Introduction **1**

2 Background **4**
2.1 Network Security . . . 4
2.2 Protected Execution . . . 7

3 Objectives **9**

4 Related Work **11**
4.1 Intel Software Guard Extensions (SGX) . . . 11
4.1.1 SGX Architecture . . . 11
4.1.2 Key Derivation . . . 13
4.1.3 Enclave Measurement and Attestation . . . 14
4.1.4 Sealing . . . 18
4.1.5 Defining and Using an Enclave . . . 19
4.1.6 Development Considerations . . . 19
4.2 Trusted Hardware . . . 20
4.2.1 IBM 4765 Cryptographic Coprocessor Security Module . . . 20
4.2.2 ARM TrustZone . . . 20
4.2.3 XOM Architecture . . . 20
4.2.4 Trusted Platform Module (TPM) . . . 21
4.2.5 Intel Trusted Execution Technology (TXT) . . . 21
4.2.6 Aegis Secure Processor . . . 21
4.2.7 TrustLite and TyTAN . . . 22
4.2.8 Bastion . . . 22
4.3 Previous Benchmarking . . . 22

5 Threat Model **24**
5.1 Network Threats . . . 24
5.2 Application-Layer Threats . . . 25
5.2.1 Address Translation . . . 25
5.2.2 Memory Swapping . . . 26
5.2.3 Multicore Address Translation . . . 26
5.2.4 Cache Timing Attack . . . 26

6 Approach **27**
6.1 Enclave Architecture . . . 27
6.1.1 Secure Connections . . . 27
6.1.2 Secure Computation . . . 28
6.2 Enclave Attestation . . . 28
6.3 Quote Transparency . . . 33
6.3.1 Quote Verification . . . 33

6.4 Alternative Attestation Model . . . 34

7 Design and Implementation **37**

7.1 Two-Step Connection . . . 37

7.2 Certificate Chain . . . 38

8 Analysis **39**

8.1 Feature Availability . . . 39

8.2 Required Development Effort . . . 39

8.3 Enclave Restrictions . . . 40

8.4 Application Design . . . 42

8.5 TLS Enclave Design and Consequences . . . 45

8.6 Performance . . . 45

8.6.1 Individual Time Costs . . . 46

8.6.2 Standard TLS Handshake . . . 47

8.6.3 Two-Step Attestation . . . 48

8.6.4 Certificate Chain . . . 50

9 Conclusion **52**

9.1 Overall Evaluation . . . 52

9.2 Recommendations for Future Study . . . 54

References **55**

Acknowledgements **59**

List of Figures

1	Client-Server Web Service Architecture	4
2	Confidentiality Guarantee	5
3	Integrity Guarantee	5
4	Freshness Guarantee	6
5	Computer Abstraction Layout with Enclave using Trusted SGX CPU	11
6	Enclave Memory as Represented in Virtual Memory	12
7	Key derivation process.	14
8	Enclave Measurement Creation	14
9	Local attestation process.	15
10	Remote Attestation process.	17
11	Guarantees of a TLS Enclave.	27
12	Data Protection Through Multiple Enclaves.	28
13	Attempting to Resolve Mutual Attestation with Hard-coded Measurements.	29
14	Attesting a Fabric of Enclaves with Combined Attestation.	30
15	Handshake Process with Combined TLS and SGX Certificate.	32
16	Timeline of Data Signing on Server using Combined Certificate.	33
17	Client-side Timeline of Data Knowledge using Combined Certificate.	33
18	Timeline of Data Signing on Server using Two-Step Handshake.	35
19	Client-side Timeline of Data Knowledge using Two-Step Handshake.	35
20	Handshake Process with TLS Handshake and Standard Remote Attestation.	36
21	Pointers Crossing Boundary	41
22	Example code.	43
23	ECALL/OCALL Interfaces without Reduction	44
24	Reduced ECALL/OCALL Interfaces	44

1 Introduction

Cybercrime is rapidly becoming a major economic and security problem. Though accounting for all of the effects of the losses is extremely difficult, it has been estimated to cause global losses of between $375 and $575 billion [33]. This is between 15% and 20% of the total estimated income generated by the Internet economy. Beyond the raw value of the loss are the major effects of cybercrime: the loss of personal information of millions of people, the theft of intellectual property, and the damage done to the performance of companies and national economies.

The theft of personal information is often used for identity fraud. Identity fraud is often used to gain access other people's money. It involves stolen bank account numbers, routing numbers, usernames, passwords, credit card numbers, insurance policy numbers, and account numbers for national services, such as Social Security in the U.S. or the Canada Pension Plan. In some regions of the world, such as India, the use of ransomware has become common. These threaten to reveal personal information or stop important services unless the attacker is paid. In addition to the loss of money, the invasion of privacy constituted by this kind of theft can have substantial impacts on a person's records (criminal, financial, etc.), employment chances, and psychological well-being. There is also a looming potential for the theft and illegal use of medical information as the healthcare industry gradually digitizes records.

Companies can lose more than just financial information. In many industries, intellectual property (IP) is a crucial part of a company's ability to compete. IP includes everything from paint formulas to logos to rocket designs. Many of these are trade secrets, only protected by their confidentiality. IP theft damages a business's ability to compete against its peers by reducing the return on investment (ROI) a company had expected. The U.S. Department of Commerce estimates that IP theft (by any means) costs American businesses $200-250 billion per year [52] while the Organization for Economic Cooperation and Development (OECD) estimated that counterfeiting and piracy cost the world $638 billion in 2005 [49], a number that has likely increased since then. Damage occurs from lost revenue as off-brand products utilizing stolen IP are sold, loss of trade secrets that previously conferred a competitive advantage, and from the reduced ROI relative to competitors, especially those in possession of the stolen IP. IP theft is also damaging to national security, since disclosure or loss of military hardware designs and details can become known to a country's adversaries and competitors.

The effects of cybercrime on businesses vary from the immediate quantitative losses to long-term image and direction problems. Victims of cyberattacks can lose money in the initial breach, as when attackers siphon money out of bank accounts, during a temporary fall in stocks after the publication of an attack, or during recovery efforts following the attack. Recovery efforts can cost as high as ten times the initial losses [4], including reimbursement, legal fees, and services for customers whose data was stolen. The 2013 Target attack cost Target over $200 million [6]. The cost is also high for banks, as most cybercrime will involve one or more banks in the post-attack recovery efforts. Fraud alone is estimated to cost Mexican banks $93 million every year [56], while Japanese officials estimate cybercrime costs Japanese banks $110 million annually [33].

More difficult to value is the damage caused by the loss of sensitive information, such as investment plans, exploration data, trade secrets, negotiation data, and other insider information useful for manipulating the stock market. These can give a competitor an advantage during contract negotiations or in the market-place. Some cybercrimes involve the actual destruction of data as well as theft. This can have long-term

consequences for the company and its customers, from the legal to the sentimental. An attack against South Korean banks and media outlets in 2012 saw data erased from over 30,000 computers [44]. This level of destruction against company property could set a business back months, if not years.

In the long term, businesses and national economies can be significantly impacted by cybercrime. Companies that suffer from high-profile attacks can have their reputations as safe and valuable places to do business damaged. In diverting resources to cyber defenses, companies will often draw from their IP-producing divisions. This decreased investment in research and development slows the rate of innovation, which hurts people worldwide. Companies and organizations may also make more risk adverse decisions and limit Internet use. IP theft can also damage a nation's exports, which has been shown to increase unemployment. It has been estimated that cybercrime has cost the US up to 200,000 jobs due to decreased export growth [33].

Malware has become a major source of high-profile attacks related to cybercrime, such as denial-of-service attacks from botnets, direct attacks from computer worms, and other assorted computer viruses [19]. Unlike many other activities of cybercrime, malware is actively present on the machines it attacks, where it compromises the normal operation of programs by snooping on sensitive data and interfering with normal code execution. Malware has been estimated to cost $370 million worldwide [3], a number that is likely to rise.

Trusted execution environments (TEEs) have been developed to help combat cybercrime. A TEE allows a program to run without interference from other programs on the computer, including the operating system or hypervisor. Data in a TEE is typically protected by encryption [21], paging controls [34], and/or physical isolation in separate hardware modules [14, 55]. These two features, trusted runtime and protected data, make TEEs useful tools for impeding the ability of malware to steal user secrets.

TEEs are platform-centered solutions, and are able to attest their existence to other programs on the same machine. However, attesting to remote platforms is more difficult but also more beneficial. A successful remote attestation would provide a remote user with a guarantee that the program with which the user is interacting is the anticipated program and not a malicious one. It can additionally provide a user with the guarantee that the program's—and the user's—secret data is protected by a TEE. However, proving the identity of the other machine is not addressed by remote attestation.

Transport Layer Security (TLS) is a protocol for arranging a secure channel between a user and another machine that can attest to the identity of the peer. TLS incorporates a chain of verifiable certificates based on RSA cryptography to create an encryption key for the session while at the same time validating the identity of the machine and in some cases the user as well. Both the identity validation and session-key derivation processes are reliant on the secret status of the private keys matching the public keys presented in the certicates. Loss of these keys can allow attackers to impersonate trusted servers and steal user data [37, 13].

TEEs can provide security for the private keys through their data-protection capabilities and add an additional layer of security to the TLS session negotiation process through their trusted runtime capabilities. The TLS program could be trusted to perform all necessary checks and cryptographic operations without execution interference from malware. The data protection capabilities of TEEs can aid TLS in the medium to long term by protecting the private key. Malware would be unable to exfiltrate the private key from the program during runtime and the key can be encrypted by the TEE for storage in the long term. While TEEs will not solve all cybercrime problems, they can have a significant impact by protecting user data and program execution.

We propose a method of integrating the remote attestation process of Intel's Software Guard Extensions

(SGX), a TEE technology, with TLS. Our method combines a remote attestation quote containing verifiable data about the SGX TEE with a TLS certificate for a key cryptographically tied to the TEE. The two together allow a remote user to verify both the TEE and the identity of the server. Verification of the TEE can provide the user with assurances of how data submitted to the server will be handled in addition to the assurances about the TLS handshake process. We show that our method is not significantly slower than the TEE-less handshake.

2 Background

This section will cover the security background that is necessary for discussing trusted computing. This includes possible guarantees of secure design, key schemes, and a summary of protected execution.

2.1 Network Security

Our research is largely focused on networked systems. A simple network assumes both endpoints are inaccessible to an adversary so any protections are based on the assumption that the adversary can only view the messages in transit across the system. There are some common cryptographic primitives of protecting network traffic that are mirrored in SGX. This section will explain how the cryptographic primitives work in the context of networks. We later will discuss how they are used in SGX and the adversarial model SGX protects against.

While not the only structure for networking, our work focuses on the client-server architecture, shown in Figure 1. In a client-server relationship, there exist two parties: a client attempting to communicate with a service on a server, and that server hosting said service. Typically the client will supply some data that a specialized program on the server processes. Servers then send back any output, such as a processed form of the initial data. In the World Wide Web, most client data are in the form of search queries, text posts, and webpage requests. Server data returned are usually webpages and their embedded content, such as videos or games. If these two parties want their machines to communicate over a secure channel, they will typically use TLS which is the current standard for communications security.

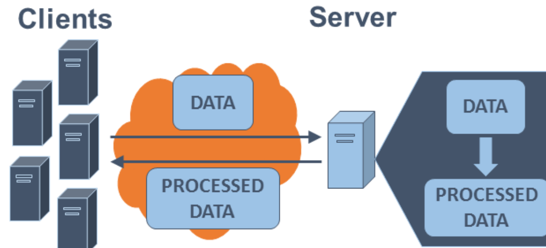

Figure 1: Client-Server Web Service Architecture

Guarantees We will be focusing on cryptographic primitives that provide confidentiality, integrity, and freshness. Confidentiality guarantees that no party other than the intended receiver can access a message. Integrity means that *either* the receiver will receive an unaltered message or the receiver will be able to notice any tampering. Freshness guarantees that a received message is the latest message coming from the sender.

When discussing cryptographic primitives, we will use three parties: Alice, Bob, and Eve. Alice and Bob are presumed to be innocent and are attempting to communicate securely. Eve is a malicious third party that can see, block, resend, and alter both the content and order of messages passed between Alice and Bob but nothing of the innocent parties' execution. Using secure methods that meet the desired confidentiality, integrity, and freshness guarantees, Alice and Bob should be able to communicate without Eve interfering with or gaining knowledge of the message contents.

To briefly cover some attacks that can occur if the aforementioned security guarantees are not met, we will consider the following scenario: Alice is attempting to communicate with her associate Bob about ordering a new batch of cars to be manufactured.

Guarantees Without confidentiality, Eve from a rival car company can see the number of cars ordered, how much it will cost, and possibly the designs of a new car.

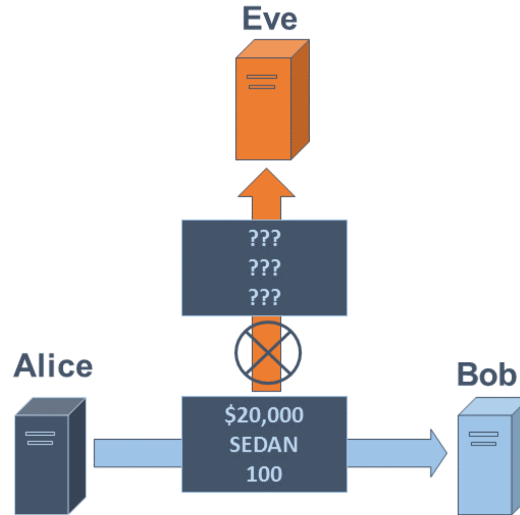

Figure 2: Confidentiality Guarantee

Eve might be able to gain a significant economic advantage over Alice with this information. As in Figure 2, it is imperitive that Eve not be able to read any data that is sent between Alice and Bob. Eve is allowed to know where the message is being sent, but not anything about the contents of the message. If integrity is not assured, Eve could tamper the order so that Alice orders half as many cars or change how much Alice is paying for each car.

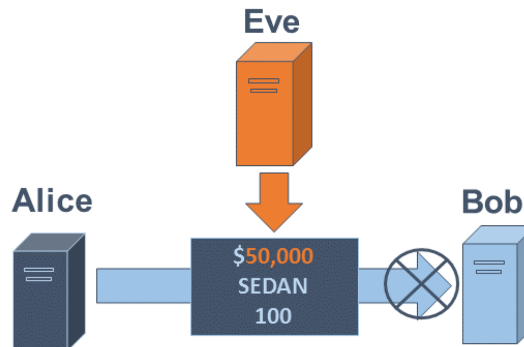

Figure 3: Integrity Guarantee

Integrity assures Bob that any message sent by Alice will arrive without change to Bob *or* Bob can detect that a change was made. Figure 3 shows Bob rejecting a message that was changed by Eve. Although the

system could not prevent the change, the guarantee of integrity remained sound because Bob was able to detect the change.

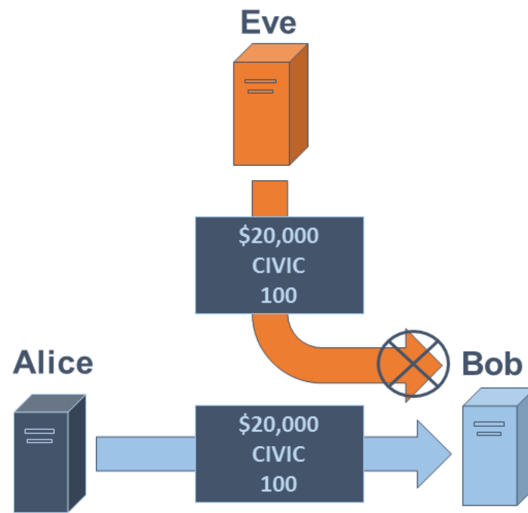

Figure 4: Freshness Guarantee

Finally, without freshness, Eve has the opportunity to record an order and, at a later date, send the exact same message to Bob who will believe Alice made the same order twice. Even if confidentiality and integrity are upheld, Eve can record messages without knowing what they contain. Bob should be able to determine if a message is the latest message Alice wanted to send, or if it is a duplicate of an old message. This is even helpful in non-sensitive networking as messages may need to be sent several times before they sucessfuly reach the target. Some of these duplicate messages may take longer to reach the target than others so Bob may receive several duplicate requests. He must then decide to only accept one of the requests and drop the rest. In Figure 4, Bob determines he has already accepted the duplicate message and therefore drops it.

Symmetric Keys The first method we will describe is symmetric key cryptography. Alice and Bob can share an identical key. Any data encrypted with this key can only be read by those who know the key. Methods for this kind of encryption have been known for a long time but it requires that the key must be securely shared between all communicating parties *before* sensitive messages can be passed. The most important concept is that any party with a symmetric key can access the data protected by the key. If only one party controls the key, then only that party can access the encrypted data. Symmetric keys can provide confidentiality as Eve does not know the key and therefor cannot read any encrypted messages. Integrity is provided when symmetric keys are combined with a hashing algorithm, such as the Secure Hashing Algorithm (SHA), that can produce a Message Authentication Code (MAC). The MAC is a hash of the message that was produced using the symmetric key. If Eve changed the message and not the MAC, Bob could tell that Eve tampered with the message which is one of the integrity goals. Eve cannot create an authentic MAC for her new message because she does not know the symmetric key.

Symmetric keys are also useful in trusted computing if a program can solely control a key. In this case, only one program controls the key, allowing it to keep secrets from any other party. If a program wants to store data for itself on hardware that is considered malicious, the symmetric key can be used to encrypt data

for itself. If a storage disk is not trusted, symmetric keys provide the confidentiality needed for long-term storage.

Asymmetric Keys The second method used to communicate over an untrusted medium is asymmetric cryptography. In this method there is no need for key distribution. Alice and Bob each own a pair of keys, one key is kept secret while the other is freely shared. Bob, knowing Alice's publicly-shared key, can encrypt messages with the public key that only Alice can decrypt with her privately-kept key. Because Eve does not know Alice's private key, she cannot decrypt any messages meant for Alice. The conditional guarantee is that, as long as only Alice controls the private key, any message encrypted for Alice can only be read by Alice. Asymmetric cryptography is even more useful because the encrypting process can be reversed. If Alice encrypts something with her private key, anyone who knows her public key can verify that the message came from Alice because only her private key could be decrypted by the public key. Asymmetric cryptography provides both confidentiality and integrity in the same manner as symmetric keys.

When the client is first trying to contact the server, the server and client exchange their public keys so that each party can encrypt messages that only the other can decrypt. The encrypted messages are actually used to negotiate a symmetric session key which is then used for all further communications. A symmetric key is used for most communications for several reasons including speed and forward security. Forward security in this case refer to the inability of compromised keys to reveal any future conversations. Session keys are often used only once so that a compromised session key only compromises one conversation.

There is nothing in the previous scenario that lets a client know with whom he or she is talking. Even if a server has a key pair, the client should be able to tell that they are talking to the correct server. If Bob wants to access Alice's services, Eve can respond to his request with her own public key. For Bob to know whose key he just received, he relies on a Certificate Authority (CA) to check that the key does indeed belong to Alice. Again, Bob could encrypt messages using Eve's public key and have a secure channel with her, but he actually wants to talk to Alice instead. The CA's job is to keep track of who owns which key so Bob does not have to. The CA signs Alice's key with their own private key and marks that it belongs to her. Bob only has to keep the CA's public key to verify that the CA was the one to sign Alice's public key. Bob is trusting the CA to do its job so when he gets Eve's public key and sees that the CA marked it as belonging to Eve, Bob knows he is not talking to Alice and will drop the connection. TLS uses the CA infrastructure as a form of authentication for servers (and sometimes clients) [38].

2.2 Protected Execution

Similar to networks, we will be focusing on confidentiality and integrity as they relate to trusted computing. In this context, *confidentiality* means the execution of a protected program is not visible to the rest of the system. Only the inputs and outputs are observable [8]. *Integrity* on the other hand means the execution of a protected program cannot be altered. Therefore if the program completes, the output will be the same as a correct execution on a reference machine. The program can be affected by withholding resources or execution time, but the behaviour will remain unchanged [8]. Freshness is not a necessary guarantee because if the execution maintains confidentiality and integrity it is trusted to produce the same output each time. Protected execution does not extend to the freshness of the input or output data.

Past protection systems have provided these security guarantees, but in reverse. Sandboxing, process

isolation, managed code, etc. all provide confidentiality and integrity to the rest of the system *against* the running program. In certain cases, we would prefer to provide these guarantees to the program instead by protecting it from the rest of the system, no matter how privileged. This is the goal of a Trusted Execution Environment (TEE). TEEs can use either software or hardware to provide isolated and shielded execution of code. Software methods rely on trusted components such as hypervisors to separate programs from each other [8]. Hardware implementations typically use separate trusted hardware to shield the code from the rest of the system.

3 Objectives

The ideas of data isolation were implemented as early as 1969 and described by Saltzer, et al. [40, 41], Schroeder [45], and Bell, et al. [10]. The Multics team based their protection scheme on permission instead of exclusion. Their system was meant to provide isolation of files and memory on a per-user basis. Each user account was given any combination of read, write, or execute permissions to each file on the system. This hierarchical permissions structure is still found in modern systems but can be subverted by malicious operating systems and hypervisors. Program memory references are marked with similar read, write, and execute permissions to the file system as well as a designated subsystem that is allowed to access said reference. In many ways, these subsystems are a precursor to modern TEEs and their secure interfaces. Each subsystem included a set of procedures and databases with minimized interfaces. These interfaces were called *gates* and allowed other subsystems to use the functionality within the protected subsystem. Saltzer [40] mentions that, if the subsystem protections were hardware enforced, it would be possible to isolate programs based on their identity. Untrusted programs would be able to use the protected functionality without performance overhead. Most importantly, a user may develop his or her own proprietary program as a protected subsystem and the data is protected at the same level as the operating system. The subsystem design with small interfaces also allows for a *fabric* architecture that provides data isolation in addition to execution protections.

Our work is continuing the work of Joshua Guttman and John Ramsdell on TEE services. They aim to develop better methods of secure computing using recent TEE advances. TEEs provide a method for isolation so that compromised systems can only spill their own secrets. A *fabric architecture* of TEEs separates each functionality within a program, provisioning each subsystem with only the secrets needed to function. For TEEs, a *fabric* architecture has the same goals as Multics. Each TEE is responsible for a subsystem of the whole program. This way, if a TEE is compromised, the remainder of the program's protected subsystems can continue to function as normal. Any secrets protected by the uncompromised subsystems are also protected from the now-malicious compromised TEE. The whole program thus has no single point of failure, as each TEE would need to be compromised individually to gain its secrets.

The usability of trusted computing for networking relies on software attestation. Software attestation is a process that allows a program to present information about itself (code, runtime environment, etc.) to another program in a verifiable way. In the context of trusted computing it allows one TEE to prove to another TEE that it is actually running in a secure environment. Different implementation of TEEs have included varying methods of software attestation. Attesting software to a remote computer always requires sending additional data about the software over a network connection. This could be detrimental to the performance of large services that must offer a secure, TEE-based service. Each connection would require $AttestData$ amount of information to be processed and transmitted during the attestation process. Thus at-scale services that deal with N connections must be able to serve $N * AttestData$ in addition to all other traffic.

To develop a *fabric* architecture, our work began by developing a prototype service to illustrate its usability and discover possible issues with implementation. Our goal was to prove the efficacy and compatibility of our *fabric*-style framework with standard services. In addition, the efficacy must be analytically proven to function at the scale of a standard web service. To do so, our development expanded beyond a prototype to include the design of a protocol that would allow TEE web services to efficiently function at scale. The

protocol aimed to provide a usable and secure attestation model. The service was required to meet the following criteria:

- Use the Intel SGX TEE platform to implement a TEE-fabric architecture.
- Able to provide a client the standard network guarantees of TLS.
- Able to attest itself to a client in an efficient and cryptographically meaningful manner.

The rest of this paper will describe our findings from development and our contributions to the attestation model. Our prototype implements a server which uses an SGX enclave to handle the server's private key and perform the steps of a TLS handshake that meets these goals. This work will provide an example from which abstractions can be made for TEE-fabric libraries to help develop similar secure web services. However, we did not test on the performance of the prototype and protocol under heavy usage so further optimization may be required.

4 Related Work

Before developing under a TEE, it is important to understand the theory behind trusted computing and some generic security principles. TEEs are very powerful tools but one cannot use the available capabilities without comprehending why they exist. This section will explain SGX's main capabilities and cover work related to SGX.

4.1 Intel Software Guard Extensions (SGX)

In this section we describe the capabilities of SGX. The full references can further expand on the implementations of each capability [2, 23, 48, 28, 22, 25, 26, 7]. We chose to use the SGX architecture over the other options defined in §4.2 because of its ability to provide very small roots of trust and the potential for high performance. The threat model of SGX treats every layer of the software stack as malicious. The processor itself is assumed to be implemented correctly and not compromised. We will collectively refer to the OS, hypervisor, firmware, and non-processor hardware as the OS when discussing malicious privileged code.

4.1.1 SGX Architecture

SGX is fully implemented in the CPU, requiring new hardware and structures. Most of the newest Intel CPUs that are built on the Skylake architecture include the capability for SGX enclaves [58]. An SGX enclave is a new structure for execution that allows programs to protect their security critical components. When the processor runs in its enclave mode, the memory access semantics are changed to include additional checks, only allowing the code inside an enclave to access the rest of the enclave's memory. This effectively isolates the enclave from the rest of the system, only trusting in the CPU and the enclave code itself (Figure 5). The trusted computing base is in gray.

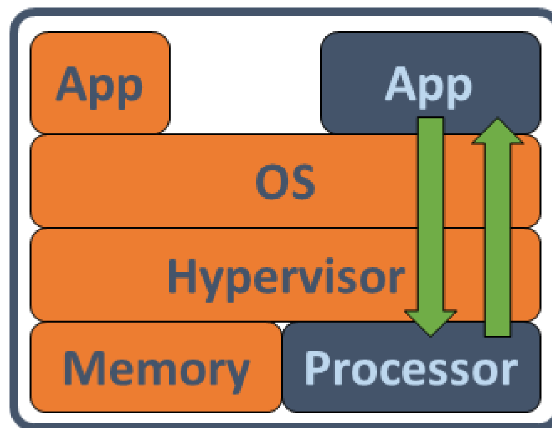

Figure 5: Computer Abstraction Layout with Enclave using Trusted SGX CPU

Guarantees When switching out of the enclave execution back to the untrusted code, all registers are cleared, the TLB is flushed, and other aspects of the processor's state are scrubbed [34]. In the event of an asynchronous exit due to interrupt or exception, enclave execution state is saved to a State Save Area (SSA)

and the processor's state is scrubbed before the event is handled [14]. SGX seeks to provides confidentiality, authenticity, and integrity violation detection for a subsection of an application's source code. Availability is not a goal for SGX enclaves, and it includes no protections against denial of service by the operating system.

SGX adds 17 instructions to the regular instruction set that handle loading memory into enclaves, entering enclaves, access to resources, and more [34]. Programs can use these instructions to create an enclave that holds an entry table, enclave code, and the enclave's heap and stack. When launching an enclave, the untrusted application will load the trusted code base into the enclave, then request the enclave's launch.

During the loading step, the code and any data on the heap within the enclave is measured. This measurement is used to determine if the enclave is approved for launch, and later for attesting the software [2].

Software that uses enclaves will generally follow a prescribed life cycle. The source code that is loaded into an enclave is unencrypted and readily available, so software within enclaves must be launched with no sensitive data. The code itself is stored in dynamically loaded libraries which makes enclave design similar to existing modular software structures.

The enclave exists inside of the process's virtual address space as shown in Figure 6. The enclave contains within it all of the same components a normal process has, such as an entry table, and heap, stack, and code segments [34]. Attempts by the untrusted process or even the operating system to access enclave memory recieve an Abort page [14]. The enclave developer sets the size of the enclave's stack and heap, which must be 4K-aligned [26], up to an implementation-specific maximum [34].

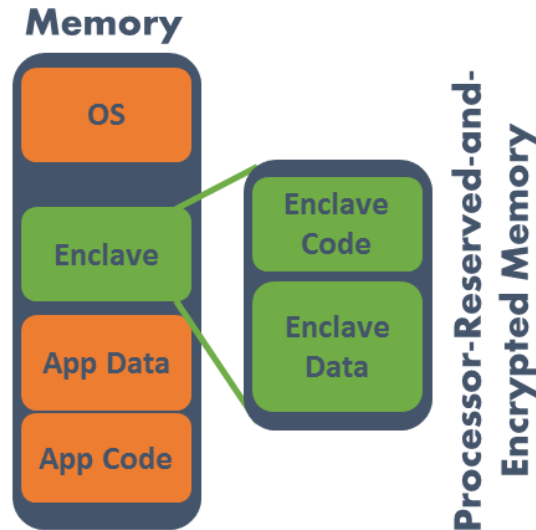

Figure 6: Enclave Memory as Represented in Virtual Memory

For the protected enclave to be useful, SGX allows software to be provisioned with secrets from a remote source. The enclave must first prove through a secure assertion that it is under a trusted hardware environment and using the expected code base. Then, once the remote party decides the enclave is safe to provision to, a secure channel is set up and secrets can be transferred. For the service's enclave to save those secrets securely, SGX permits the sealing (encryption) of data that only the same trusted environment can retrieve. Updated software can use derived keys that allow the new trusted environment to unseal the old data that would otherwise be rendered useless [2].

We will go into more details about sealing in §4.1.4 and attestation in §4.1.3. First, we need to examine the hardware components of SGX. To protect an enclave's memory, SGX uses an Enclave Page Cache (EPC). This extra cache is inaccessible to software and hardened against physical attacks. Every enclaves' pages, code and data, are stored in this cache. To prevent enclaves from reading each other's memory, SGX maintains an Enclave Page Cache Map (EPCM). This map structure is consulted by the Page Miss Handler to mediate memory accesses [34]. The operating system and hypervisor remain in control of the regular and enclave page tables. The EPC and EPCM are used to prevent memory-mapping attacks. When an enclave page is evicted from the EPC, the EPC page is marked as available in the EPCM after the OS has flushed all translation look-aside buffer entries that could translate to that page. EPC pages that are written to DRAM or disk are encrypted, transferred, then zeroed in the EPC [14]. Integrity is provided to these pages by the Memory Encryption Engine's integrity tree using a stateful MAC with nonces [21].

To guarantee freshness of the pages when retrieved from DRAM, SGX uses a Version Array (VA). This array contains a nonce for each page that gets updated each time the page is evicted. The array is also held in the EPC so that it cannot be changed [34]. The VA pages could be maliciously deallocated because they do not belong to a specific enclave, causing the subsequent denial of access to the referenced page. While this is another possible denial of service opportunity for a malicious OS, it maintains the security guarantees that are expected from SGX.

4.1.2 Key Derivation

SGX makes use of several keys: the EINIT token, provisioning, provisioning seal, report, and sealing keys [28]. The EINIT token key is used during the enclave creation process to finalize the enclave before runtime. The provision and provisioning seal keys are used in the Intel Provisioning and Attestation Services, detailed in §4.1.3. The report key is used during the local attestation process to validate reports. The sealing key is used to encrypt data for later use that can only be decrypted by certain enclaves enclave.

These keys are derived from two root keys created during the manufacturing process, the root sealing key (ROOT SEAL KEY in Figure 7) and the root provisioning key (ROOT PROVISIONING KEY). The root provisioning key is retained by Intel for use in the Intel Provisioning Service while the root sealing key is randomly generated and not retained. The provisioning key is run through a pseudo-random function (PRF) at boot time (before the first instructions are fetched) to generate the Trusted Computing Base key. The PRF is run as many times as the hardware has been updated. This number is kept as the CPU Security Version Number (SVN), a non-integer value determined by Intel for each update. This ensures that leaked keys do not compromise either the root key or future derived keys. This TCB key is combined with the root seal key to generate the other keys [28].

Two keys can be accessed by Independent Software Vendor (ISV) -authored enclaves: the report and seal keys. The EINIT token, provision, and provision seal keys require that the enclave be started in a mode with additional privileges in order to access them. The provision key is only derived from the TCB key so that it can be used to prove the identity of the processor during validation with the Intel Provision Service (explained in the next section). All keys are tied to the enclave for which they are produced, because the enclave measurement and metadata (represented by MRE in the figure) are used as additional data for the derivation [28].

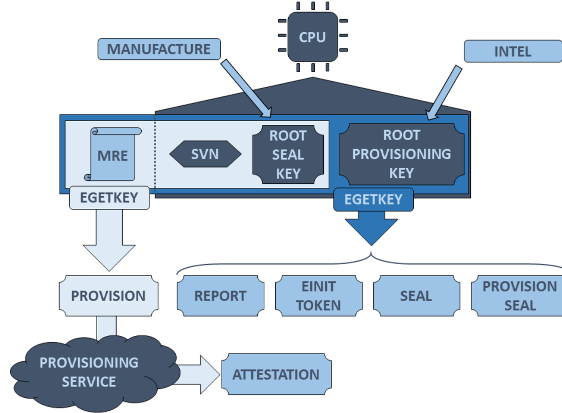

Figure 7: Key derivation process.

4.1.3 Enclave Measurement and Attestation

When the enclave is created, a secure hash of the initial state is created. This hash contains the code, data, stack, heap, pages, and SGX flags. The hash is called a *measurement* (styled as MRENCLAVE) and is used in part for verifying the enclave's identity and deriving keys. MRENCLAVE is created by the process in Figure 8 when the enclave is first created. The ECREATE instruction begins the digest creation process and provides the initial value. Each subsequent EEXTEND instruction adds the page added to the digest. Finally, the EINIT instruction finalizes the digest and locks it. MRENCLAVE can thus be used to identify a particular enclave. It is also used during enclave startup when it is compared to its respective field in SIGSTRUCT (a structure which maintains the identity of the enclave signer, usually the developer [34]). This check ensures that the enclave creation process was not modified by the OS or a malicious program.

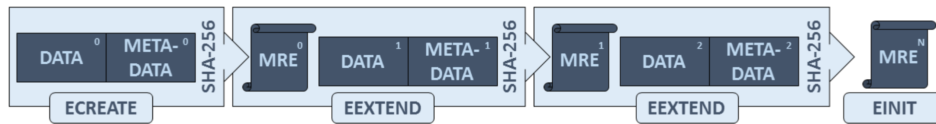

Figure 8: Enclave Measurement Creation

SGX extends the system beyond just measuring the enclave's contents to allow software updates to be recognized as the same software as their previous versions. In order to do this, SGX also uses a certificate-based identity scheme. Enclave developers act as a Certificate Authority that will sign a certificate for each of their enclaves. When initializing an enclave, the certificate is checked along with the Unique Product ID given by the developer. If the ID and signing RSA key are the same, then the software is assumed to represent a different version of the same software [14].

The report key and MRENCLAVE come together to allow for enclaves to perform software attestation between themselves. There are two types of attestation: local and remote. Local attestation is carried out between enclaves on the same machine, while remote attestation is performed between enclaves on different machines. An enclave wishing to attest its validity to another enclave (here called the target enclave) sends information about itself that can only be confirmed as valid by the target enclave.

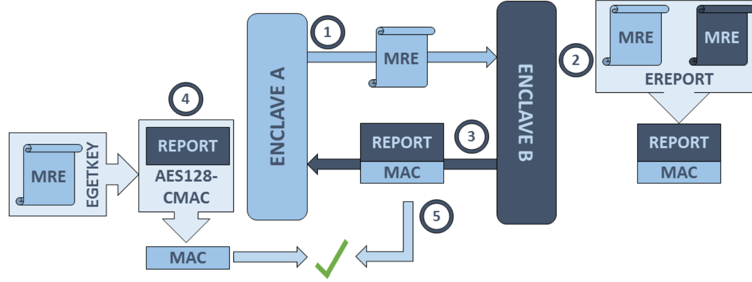

Figure 9: Local attestation process.

Local Attestation To perform local attestation, the software running inside SGX can ask the hardware to generate a *report* containing information about the enclave and processor's current security states using the EREPORT function (step 2 in Figure 9). This instruction also takes as input a small amount of data to be shared with the other enclave and a TARGETDATA struct used to identify the other enclave (obtained in step 1). As seen in Table 1, the information includes developer-configurable aspects of the enclave (MiscSelect, Attributes, ISVPRODID, ISVSVN), the Intel-set CPU firmware version (CPUSVN, a non-integer value), the enclave and signing key measurements (MRENCLAVE, MRSIGNER), data to be passed between enclaves (Report Data), and the signed MAC of the report with key-wearout protection (Key ID and MAC) [14, 28].

The MiscSelect, Attributes, and CPUSVN fields detail the current runtime security posture of the running enclave. An evaluator can use these fields to determine if the enclave is running with acceptable security configurations. ISVPRODID, ISVSVN, MRENCLAVE, and MRSIGNER contain information about the enclave code itself. ISVPRODID, ISVSVN, and MRSIGNER are called the Signing Identity, which can be used to determine the developer of the enclave [2].

The MAC field is a digest of the other fields, including the Key ID, signed by the Report Key of the target enclave. The target enclave's Report Key is derived by the hardware based on the information provided to the EREPORT instruction in TARGETDATA. When the target enclave recieves the report (step 3) it checks the report's authenticity by re-computing the MAC (step 4). Only the enclave has access to its own report key, which it can get through the EGETKEY instruction. If the MAC in the report matches the one produced by the target enclave (step 5), then the target enclave can be assured that the report is authentic [2]. It can then check MRENCLAVE and the Signing Identity to verify it is communicating with the correct program. Finally, the process can be reversed to make the attestation mutual. This has been combined in the SGX Library with a Diffie-Hellman key exchange to create a secure tunnel between the two enclaves for future communications [26].

Intel Provisioning Service and Enhanced Privacy ID The Intel Provisioning Service (IPS) allows SGX-capable processors to get an attestation key for remote attestation. The attestation key is called the Intel Enhanced Privacy ID (EPID). Enhanced Privacy ID is an extension of Intel's previous Direct Anonymous Attestation, a group signature scheme [12]. This allows each signer to be part of a group, which can be verified by the group's public certificate without allowing the individual signer to be identified. An EPID key has two signature modes: the Random Base mode and the Name-Based mode. The first generates

Field	Size (Bytes)	Description
CPUSVN	16	Security version number of CPU (as byte array)
MiscSelect	4	State Save Area (SSA) Frame extended feature set
Reserved	28	Currently used for spacing
Attributes	16	Attributes flags for enclave
MRENCLAVE	32	SHA256 digest of enclave creation process
Reserved	32	Currently used for spacing
MRSIGNER	32	SHA256 digest of RSA public key used to sign enclave code
Reserved	2	Currently used for spacing
ISVPRODID	2	Enclave product ID, assigned by enclave signer
ISVSVN	2	Security version of an enclave, assigned by enclave signer
Reserved	60	Currently used for spacing
Report Data	64	Information to be passed between enclaves, determined by calling program
Key ID	32	Key wear-out protection value
MAC	16	Cipher-based Message Authentication code of above signed by the Report Key of the target enclave

Table 1: Enclave Attestation Report [22].

a random base for each signature, while the second uses a name as the base (called the basename). EPID signatures are divided into linkable and unlinkable signatures. Linkable signatures use the same basename each time, while unlinkable signatures use a different one. It is thus easy to tell if a key made a given signature if that signature is linkable [28]. Additionally, pseudonymous EPIDs allow the verifier to determine if they have verified the platform previously [2]. Fully anonymous only determines the group and we will be focusing on this mode.

Revocation of EPID keys can happen at three levels. First, a member key can be revoked if it is leaked. Second, a signature can be revoked if suspicious behavior is detected. Finally the entire group can be revoked if the group master key is leaked. Revocation lists are maintained by a revocation authority [28].

The IPS is used when first generating an EPID key, when regenerating that key after a TCB upgrade, or to recover lost previous keys. During the provisioning process, an Intel-authored Provisioning Enclave (available in the SGX Software Development Kit) sends proofs of the processor's secure computing environment to the IPS. After successful validation, the EPID key is sent and the Provisioning Enclave executes a blind join

protocol to become a member of the EPID group [28]. The EPID key can then be transferred to the Quoting Enclave for use in remote attestation.

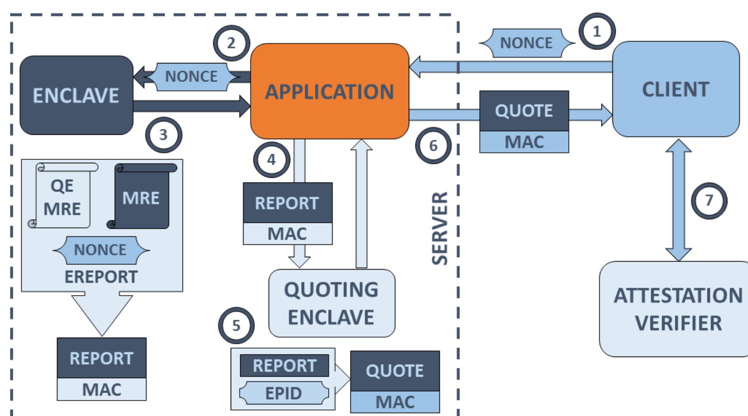

Figure 10: Remote Attestation process.

Remote Attestation Remote attestation builds on local attestation to extend those guarantees to another platform. It requires two additional constructs. The first of these is the EPID. The second in another Intel-authored enclave provided in the SGX SDK, the Quoting Enclave (QE). The QE has access to the EPID key provisioned through the IPS and uses it to sign augmented reports called quotes on behalf of ISV-authored enclaves [28].

Field	Size (Bytes)	Description
Version	2	Version of the quote structure
Signature Type	2	Whether signature is linkable (1) or unlinkable (0), others reserved
Group ID	4	The EPID Group of this platform
ISVSVN_QE	2	Security version of the Quoting Enclave
Reserved	6	Currently used for spacing
Baesname	32	EPID basename used in Quote
Report Body	394	All fields in Table 1 except for Key ID and MAC
Signature Length	4	Length of below signature
Signature	variable	EPID-signed digest of above (excluding signature length)

Table 2: Remote Attestation Quote [22].

Version is used when parsing the remainder of the quote. Signatures can be either linkable or unlinkable. Linkable signatures uses the same basename for the signature, making it easy to tell if a given key made a

signature, while unlinkable uses a different basename each time. The Group ID identifies which group this platform's EPID belongs to when verifying the signature. The security version of the Quoting Enclave can be used to determine the quoting enclave itself. The signature is created over a hash of the quote fields.

Remote attestation begins when a challenge is sent to the application (step 1 in Figure 10). The application then sends the relevant parts of the challenge to its enclave to begin the quote creation process (step 2). The QE performs a local attestation process with the enclave it will create a quote for (steps 3 and 4), only creating the quote if the enclave is legitimate (step 5).

The quote is then sent to the remote party (step 6). The remote party can then use the public group key of GID to verify the message (step 7) [2]. Because the member key of the platform is cryptographically tied to both the platform and the Quoting Enclave, verification of the signature guarantees that the quote is valid.

Intel Attestation Service Intel provides a service called the Intel Attestation Service (IAS) that allows developers to receive quotes from their client application, and confirm through Intel that the client can be trusted. What IAS offers beyond checking the CPU includes the capability to maintain revocation lists and link developers to specific software.

A use-case for the IAS includes a client that does not currently trust a server. To confirm the server's trustworthiness, the client asks the server to attest itself. The server must take the measurement of its software and sign it through the Quoting Enclave that uses an EPID member key ($CPUK^{-1}$). Intel is the only entity in this use-case system that can verify the $CPUK^{-1}$ signature so the client sends the quote to Intel. In return, the client receives verification on the trustworthiness of the server.

To use IAS, an ISV must first use the Intel Provision Service, which confirms the authenticity of the processor using a hard-wired Root Provisioning Key on the processor. This process results in the creation of $CPUK^{-1}$. The member key is cryptographically sealed for the Quoting Enclave's use only. Then the ISV registers the machine(s) that will be communicating with the service because IAS employs a mutual TLS connection during use and additionally receives a Service Provider ID (SPID). The registered machines can then use the service to retrieve the signature revocation list, determine if a given quote is valid, and retrieve the report of the verification process of a quote. A machine can use a regularly-updated signature revocation list in conjunction with previous saved signatures to speed up the attestation process.

Our approach aims to combine TLS certificates and SGX quotes to provide the same guarantees without the need for the IAS.

4.1.4 Sealing

The ability to securely save data outside the CPU is essential for services under the threat model of SGX. Not only are malicious programs expected to have access to all files on disk, but DRAM is untrusted as well. This means that any data outside of the CPU must be encrypted and only accessible by the appropriate enclave. We talked about the memory access controls put in place by SGX so now we can cover the encryption process and the necessary keys. SGX has a capability called sealing, in which data is encrypted by a derived sealing key before being written out to disk.

Each enclave has two identities it can use for deriving sealing keys [2]. The first is the Enclave Identity, based on MRENCLAVE, and the second is the Signing Identity, based on MRSIGNER and the enclave

version number (ISVSVN and ISVPRODID). Use of the Enclave Identity will seal the data to that specific enclave. Updates to the enclave will render the sealed data unusable, as the decryption key is no longer obtainable. The Signing Identity seals data to only be accessible to the signer, allowing multiple enclaves authored by the same entity to access it. This allows updates to the enclave without leaving sealed data unreadable but also allows other enclaves access to the data. The ISVSVN field allows a Signing Identity to deny access to older, insecure versions of the enclave to new sealed data produced by current versions.

The sealing key is symmetric and only ever known by the enclave and the CPU. When attempting to write data out to disk, the CPU will perform the encryption before initiating the I/O sequence. When the data is read back into the processor's EPC, the enclave can request the sealing key again from the hardware and decrypt the data when it is safely in the CPU. As long as the hardware and software remain the same, the same key will be generated [2]. Remote clients connecting to an SGX-secured service should be able to attest the software and ensure that their data is being securely stored by one specific program.

4.1.5 Defining and Using an Enclave

Enclave code is written, compiled, and linked in a manner similar to a dynamically linked library. Writing enclave functions is the same as writing normal C/C++ code, except that most system calls are unavailable and no dynamic library dependencies can exist. Untrusted code creates the enclave using an SGX library function and receives a handle for that enclave in return. This handle is supplied as the first argument an enclave function call [26]. The enclave has the ability to call untrusted functions, exiting the enclave in the process. A function call that crosses the enclave boundary from untrusted to trusted space is known as an ECALL while a function call crossing from trusted to untrusted space is known as an OCALL [14]. In both cases, buffers and structs referenced by pointer arguments are copied into temporary buffers on the other side of the enclave boundary.

To define the ECALLs and OCALLs, Intel developed a special Enclave Definition Language (EDL) [26]. In a separate EDL file, the developer writes the function declarations for first the trusted ECALLs and then the untrusted OCALLs, designating pointed-to data sizes and data-passing direction (input, output, or both) as appropriate. The SGX SDK provides a tool called the Edger8r, which reads the EDL file and creates function definitions and header files that both the untrusted application and the enclave will link against, one for each. These functions create, fill, and pass buffers for the pointer arguments (if any) and then execute a callback mechanism to change privilege levels to or from enclave mode [34].

4.1.6 Development Considerations

There are a few development restrictions when authoring enclaves for production. First, sealed data is only unsealable on the same machine on which it was sealed. For many current applications this would not present an issue until a database needs to be backed up or migrated. Simply copying the files onto another machine leaves them irretrievable because the key used to seal them could not be reproduced on the new machine. To accomplish the move, the software could be written so that it could unseal the data, encrypt it using a key known to the destination machine, and transfer the data to the destination machine. Then, for the software on the destination machine, it would receive the encrypted data, decrypt it, and seal it.

The data migration problem serves as a platform for the second development issue: SGX does not prevent insecure programs. A poorly designed database in the previous example may allow an attacker to initiate

the data migration to an insecure server, gaining access to all secrets within. Common coding errors that result in insecure memory accesses when reading from buffers could also cause security leaks.

Finally, an SGX enclave cannot make system calls without first exiting the enclave and asking the untrusted part of the program to make the call for it. The processor does not allow enclaves to call the untrusted system calls because the OS is untrusted for all SGX enclaves [24]. This creates work for a developer trying to port legacy software onto the SGX platform. Each part of the program put into an enclave will need to be checked and modified if system calls are present. We discuss these constraints in detail in the development analysis section §8.3.

4.2 Trusted Hardware

We describe a number of trusted hardware designs that relate to our work.

4.2.1 IBM 4765 Cryptographic Coprocessor Security Module

The idea of a coprocessor is to enclose the main computing hardware: CPU, caches, DRAM, and I/O controller, in a tamper-detecting environment [14]. The system uses Faraday cages and sensors to detect tampering attempts. If an attempt is detected, the processor will promptly destroy any secrets that are currently being stored. This type of system is very resistant to physical attacks, as demonstrated by the fact that the IBM 4765 achieved FIPS 140-2 Level 4 certification [17]. The 4765 also included the capability for software attestation based on a processor attestation key only available to the service processor [14]. While significantly more resistant to physical attacks than SGX, it requires additional complexity in the form of a second computer system in an expensive protective enclosure.

4.2.2 ARM TrustZone

The TrustZone design uses additional hardware to separate a *secure world* container from the normal software stack. During execution, the processor can switch between these worlds to match the needs of the application. Untrusted software in the normal world can only use well-defined jumps to enter the secure world. The processor will keep track of two page tables, one for each world [1]. The protected pages can be ejected by the untrusted world, opening a potential cache timing attack. Hardware manufacturers do not disclose whether or not they follow this part of the design. As long as the processor chip itself is not tampered with, the secure world is not vulnerable to physical attacks [1]. Protected memory is maintained in SRAM, which, while secure, is prohibitively expensive to match the DRAM capacity. The TrustZone design does not allow for software attestation, although a secure boot mode is possible with extensions [57]. TrustZone also possesses higher resistance to physical attacks than SGX, though not as high as the IBM 4765. The single secure world could also prove a liability if a function in the secure world is exploited. The lack of attestation capabilities and more limited memory space may limit what TrustZone can do compared to SGX.

4.2.3 XOM Architecture

XOM stands for eXecute-Only Memory, an architecture for isolated execution from untrusted host software. Isolated execution is achieved by using separate cache lines for each secured application and prevents other software, including operating systems, from accessing this memory. Each secured program has a symmetric

key that is used to seal data and serves as an identifier. That symmetric key is then signed with the CPU's public key. By sealing a secret in the secured program, one can authenticate the software but not the environment. XOM also seals away and clears register states before servicing interrupts, stopping other programs from reading the data. However, XOM does not support paging because even page faults have to be handled outside of the secure container. The processor's memory controller does encrypt and authenticate (with an HMAC) data in DRAM. XOM does not hide the memory access patterns leaving protected programs vulnerable to cache timing attacks [14]. SGX has a similar architecture, though it supports paging and has multiple methods of identifying code. SGX also shares the cache timing vulnerabilities.

4.2.4 Trusted Platform Module (TPM)

TPM is a separate chip from the CPU and its only purpose is to perform software attestation. The chip stores attestation keys and is tamper-resistant to protect them. This module did see widespread deployment; however, it provides weak security guarantees. The module holds an attestation signature that covers *all* the software on the machine. From the operating system down to the device drivers, updated software will affect the measurement of the user-level program [14]. Updating acceptable lists of attestation hashes is not scalable. SGX provides a finer-grained approach to trusted execution and data protection as well as a native capability on the main CPU. The attestation mechanisms in SGX provide more precise guarantees as less data is included in the trusted computing base. However, it is not as tamper-resistant as a TPM.

4.2.5 Intel Trusted Execution Technology (TXT)

TXT is a secure container model on top of the TPM hardware. This system creates a secure container with a virtual machine hosted by the CPU's hardware virtualisation features. When a protected piece of software is active it assumes full control over the entire computer which isolates it from untrusted sources. However, since there is no DRAM encryption, physical and memory-mapping attacks could defeat the protections.

Physical and memory-mapping attacks often target the memory controller to affect how a protected program accesses or deposits memory. Memory controllers can be compromised to allow another program to access the memory of an isolated program. Now that Intel integrates the memory controller onto the CPU die, it can work in conjunction with TXT to reject direct memory access to TXT memory. Unfortunately a compromised system management mode or physical attacks will still allow access to the TXT memory and this attack has been completed multiple times [14]. SGX is resistant to this kind of attack as it encrypts all memory not in the processor cache. Instead of controlling the entire computer during trusted execution, SGX makes certain operations illegal in the enclave, allowing I/O operations and signals to be handled by untrusted components.

4.2.6 Aegis Secure Processor

Aegis is a CPU that uses a security kernel in the operating system that can isolate containers. It does this by configuring the page tables used in address translation. This kernel is part of the trusted code base so when attesting software, it includes a cryptographic hash of the software it is running and itself. Aegis uses two areas in memory, one of which is encrypted while the other is integrity-checked. The integrity-checked memory is used to avoid the encryption overhead and the encrypted memory is for longer storage. The regular operating system is allowed to page out memory from protected systems. While the secure kernel

will check the correctness of the pages, it does not stop a malicious OS from learning memory access patterns with page resolution. Cache timing attacks will work against Aegis containers [14] like they do against SGX enclaves. SGX does not have the kernel in its trusted code base and does not use duplicate memory regions.

4.2.7 TrustLite and TyTAN

TrustLite is a security architecture for embedded systems. Instead of a memory management unit, TrustLite uses an Execution Aware Memory Protection Unit (EAMPU) that checks authorization on every memory access. As the name suggests, the EAMPU also checks the current execution state of a program before allowing memory access. This includes checking the current program counter value for fine-grained access control. Interrupts are handled by a secure exception engine that maintains memory isolation [36].

TyTAN was built on top of TrustLite to extend the security capabilities. Inter-process communication was made available through the secure exception handler. The normal and secure tasks exist in the same environment, isolated by memory access protection. The untrusted OS schedules all tasks and is assumed to schedule processes fairly and prevent starvation [36]. SGX has a page-based memory that does memory access checks and it encrypts memory outside of the CPU. It was also designed for much less constrained systems.

4.2.8 Bastion

Bastion uses a hypervisor as the base trusted code base. Their goal is to protect the execution and storage of programs running within malicious software stacks. This hypervisor will ensure that the OS and other processes cannot touch a program's protected memory. It keeps track of OS memory accesses with a reverse page map, mapping the physical memory addresses to the virtual address. This is possible on CPU architectures that use nested page tables. Bastion also uses memory encryption and integrity checks. A protected page will be encrypted when written to disk, and checked against a Merkle tree before swapping back into memory [15]. The OS still can observe memory access patterns through cache timing attacks [14]. SGX bases its trust on the processor instead of a hypervisor, but is still vulnerable to cache timing attacks.

4.3 Previous Benchmarking

Previous research from Georgia Tech [31] attempted to implement a secure network service using SGX. OpenSGX is a simulation of the actual SGX hardware and therefore has different performance characteristics than the full implementation. They measured the approximate cost of doing a remote attestation in the fashion Intel encourages in CPU cycles. Their results showed that the Diffie-Hellman key exchange took up 90% of the clock cycles involved in a full remote attestation. However, this full remote attestation only happens when two parties communicate for the first time. Consequently, the performance overhead of using Intel's IAS is relatively small. Our extended certificate does incur a small overhead every time a connection is made. However, our method does not have a vastly different worst-case scenario than the alternative method that uses IAS, reveals no information to Intel, and does not require our clients to be signed up for IAS just to verify our server.

The same researchers measured the cost to send a packet from within an enclave. OpenSGX handles system and I/O calls differently to actual SGX but either way an I/O call incurs a context switch and additional instructions. Under OpenSGX, the cost of a single I/O call is high, but not prohibitively so. The

cost can be reduced by batching I/O calls and should be considered as a possibility for true SGX optimization as well.

5 Threat Model

We combine and build on the threat models for SGX and Transport Layer Security (TLS) to produce the threat model for our method. The threat model considers threats at the network, application, and hardware levels.

5.1 Network Threats

Our work is based around network services that already face a number of security threats. Methods to mitigate network-based adversaries are constantly under development but we will be building upon current widely-adopted techniques.

Sniffer Attacks are prevalent when clients are using WiFi. All network traffic is broadcast and therefore allows a third party to eavesdrop on communications. Once a party can listen to traffic, data modification is the next logical step. These attacks are only possible if the communications are unencrypted so to ensure both confidentiality and integrity, services employ Transport Layer Security (TLS). TLS encrypts and provides Message Authentication Codes (MAC) for messages.

Request Redirection is another method of sniffing where, instead of listening passively to traffic being broadcast on WiFi, an attacker tricks the client into sending all the traffic to the attacker instead of the server. This can be done through control of the client's router or spoofing one of the following: IP address, ARP address, or DNS records. Unencrypted channels of communication are incredibly weak to these methods, requiring the router to assist against IP and ARP spoofing attacks. TLS helps against most of these attacks by ensuring the client that only the intended server can receive and decrypt traffic. Domain Name System Security Extensions (DNSSEC) is a protocol widely adopted to prevent DNS record spoofing by providing point-of-origin guarantees for DNS records.

Man-in-the-Middle (MitM) attacks typically follow a request redirection. Once receiving all the traffic from a client, an attacker can act as the server *and* the client, transparently monitoring and controlling the communication channel. This method defeats TLS by making the client believe that the attacker is the secure server, and contacting the server as a "secure" client. The client is still using TLS to encrypt its messages but it is encrypting them for the wrong destination. The attacker can then read the client's requests and forward them over TLS to the regular server.

The main means of protection is through the use of certificate chains. Entities known as Certificate Authorities (CAs) issue certificates, which contain identifying information and a public half of an asymmetric key pair, on behalf of service providers, certifying that the the provider can be trusted. Certain providers may issue their own certificates, creating the chain. These certificates use the asymmetric keys as a means of verification. Each certificate in a chain is signed by the private half of the key described in the next certificate. CAs release certificates for their signatures, allowing anyone to verify the certificates created by the CAs. This prevents MitM because the attacker must either know the private key of the server being impersonated in order to read messages or present a valid certificate chain to the client that identifies the intended destination server.

Certificate pinning is another means of protection against MitM once the attacker has established themselves between the client and the server. Browsers that support certificate pinning (Chrome and Firefox) come with lists of known TLS certificates for popular sites, preventing the attacker from supplying their own certificate and completing a TLS handshake with the client. This approach makes it hard for a server to change certificates as it requires an update to the every clients' browser. This becomes a problem if a service needs to issue a new certificate as clients would not be able to access the service until a browser update gets pushed.

Denial of Service (DoS) is a class of network attack that seeks to make the victim's service unavailable to clients. Simple methods of DoS attacks involve flooding a server with requests for connection but never completing the connection, forcing the server to use all of its memory to keep track of all the unfinished TCP streams. Servers will typically have conditions in place that only allow a small number of connections from any single source and will not use any extra memory on additional connections. The flooding of requests can also use all of the server's bandwidth, starving legitimate client requests. Typically, it is very hard to out-scale a service provider but in the cases where it is a risk, rate limiting upstream of the server can sufficiently mitigate the attack.

Distributed Denial of Service (DDoS) attempts to circumvent the previous mitigation by using many machines to flood the target server. The targeted server typically cannot keep track of the millions of connections and will not block connections because they are coming from different machines. Mitigations include distributing the load across multiple servers to out-scale the attack, as well as rate-limiting that can reduce the effect from each attacker. Either method requires the server to be able to efficiently deal with traffic. Given that SGX incurs a processing overhead, it could become easier to overwhelm a server.

5.2 Application-Layer Threats

The design of SGX was motivated by completely hostile software environments. Privileged and user-level software alike are considered to be malicious. There are also processes that are more privileged than a hypervisor or OS, specifically System Management Mode (SMM) code. SMM is the highest privilege level and is only used for very specific interrupts. There have been a number of reported exploits that can abuse SMM interrupts to gain unprecedented control over a CPU. Once SMM has been compromised an attacker has access to all software on the computer. An attacker does not always need the highest level of control over a CPU to tamper with a user's program. The following are methods that have been used by attackers to gain control of, or read secrets from, a running process.

5.2.1 Address Translation

On standard and isolation-augmented CPU architectures alike, software can be vulnerable to address translation attacks. System software can control the CPU's page swapping allowing for attacks against the virtual memory abstraction. The malicious system software can switch the page tables for an application, changing where virtual addresses are mapped to in DRAM. When the application calls the targeted function in virtual memory, the CPU will attempt to fetch the pages associated with that virtual address from DRAM. Tampered page tables will associate the wrong DRAM pages with the virtual address causing incorrect function

retrieval and execution. Even if the application is running in an isolated container, security checks on the page tables are necessary to prevent attacks from tampering with application calls. These attacks can be defeated by tracking the correct virtual address for each DRAM page associated with a protected container.

5.2.2 Memory Swapping

If the pages are being tracked, an attacker can ignore the page tables and do the swapping in memory instead. Using normal page swapping, the malicious system program can evict the function pages to disk and then swap the bits on the disk. The page table remains the same but the data it is pointing to changes. Aside from the extra actions to swap the data on the disk, this attack is identical to legitimate page swapping. The best protection against this attack is to cryptographically bind the contents of each page to the virtual address. This can guarantee integrity of the memory mappings, but it also should guarantee freshness to prevent system software from replaying an old page and undoing changes that the application made.

5.2.3 Multicore Address Translation

The last attack that uses remapping of memory is possible on multicore machines that share caches. While one core is executing the protected application, another core can run the system software that will evict the target pages out to memory, switch the bits in memory, and as long as the TLB entries in the protected core have not been invalidated, the protected core will read back in the swapped data without checking to revalidate the page. To avoid this attack, the trusted container must ensure that all cores invalidate their TLB entries when a page gets evicted.

5.2.4 Cache Timing Attack

The last software attack we will consider when evaluating applications protected by SGX is a cache timing attack. In this attack a malicious piece of software, privileged or non-privileged, attempts to garner information about the protected program's memory access patterns by measuring where its own memory is held. The attacker's program can fill the cache range that the target program would be using with its own memory. While the target program executes, the attacking program can continually try to access its own memory and by measuring the time it takes to access the page, it is possible to measure if that cache location had been used by the target program. Over time, the attacker can learn a subset of the target's memory accesses, and sometimes sensitive information if the data accesses are based said information. On a multicore CPU, the attacking program must be scheduled on the same core if the attacker wants to measure the L2 cache. The L3 cache can be used to attack any process on a CPU because that cache is shared but less access data can be measured. These attacks require the malicious program to be able to fill same cache as the target. Using a cache partitioning scheme will remove all threat, although implementing such a scheme requires a trusted party to enforce it. In the case of SGX, where all system software is untrusted, cache partitioning would have to be implemented in hardware. This introduces additional costs as the processor must be redesigned to incorporate the new partitioning elements.

6 Approach

Our method has several parts. Most prominently, we present a modified TLS certificate that can perform both the SGX attestation process and the TLS handshake. This method requires the server to seal the private key used in the handshake and for the source and measurement of the enclave that handles the handshake to be released. We first describe how the enclave should be designed for our method to be successful. Then we describe how the attestation will be done, followed by a discussion of "quote transparency", an idea which will allow enclave measurements to faithfully identify trusted software. Finally, we discuss a two-step method of achieving the same guarantees by simply running the remote attestation process after the TLS handshake completes.

6.1 Enclave Architecture

Other researchers have used the approach of putting every part of a program within an enclave, only providing I/O through the untrusted domain [9, 11, 46, 8]. While this is sometimes easier to develop, there is a decrease in the overall security of the software as argued in §3. The fabric architecture we follow splits a program into separate, self-contained systems. Ideally, each enclave deals with an individual secret or set of related secrets. To choose the division of systems, we broke down a standard web service by the secrets that are held.

6.1.1 Secure Connections

The first secret that should be held by any modern secure web service is a TLS private key. Holding the TLS private key is proof of a company's identity and losing exclusive possession is inimical to doing business. In order to isolate the private key, we needed to include every function that needed access to the key within our TLS enclave. When performing a TLS handshake, any operation that required a signature or decryption from the private key would be redirected to the enclave.

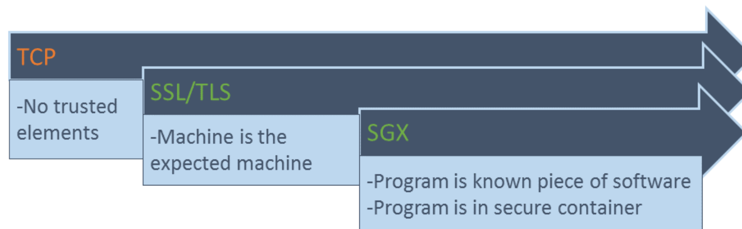

Figure 11: Guarantees of a TLS Enclave.

The enclave then uses the protected private key to perform signing or decryption as necessary and return the data to the unprotected application. The data itself is revealed to the untrusted application and could potentially be snooped by a malicious privileged software. However, the benefit of sealing the key behind an enclave means that even elevated software could not steal the private key. When the server is not running, the private key can be sealed on disk and only the TLS enclave could retrieve it when the service started up again. In doing so, a service can layer the trust of multiple protocols as visualized in Figure 11.

6.1.2 Secure Computation

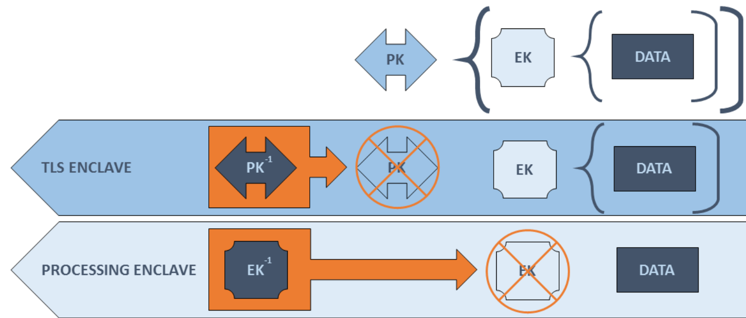

Figure 12: Data Protection Through Multiple Enclaves.

Assuming our service aims to provide a secure processing capability, we must treat and protect the user's data as a secret. In order to do so, we place data processing functions within another enclave so that the data cannot be observed by a malicious piece of software. This has the added benefit of protecting the user's data if the TLS enclave is compromised, and vice versa. The user can encrypt the data first for the processing enclave using `EK` ensuring only that enclave can act upon the data. Figure 12 shows how a client would establish a connection with the TLS enclave and use the generated master secrets to transfer messages to the server. Once the messages are transferred, the client assumes the TLS enclave will securely pass the messages to the processing enclave. Even if the TLS enclave does not perform as expected and attempts to reveal the message's contents, the message has been encrypted and no entity other than the intended enclave can change or act upon the data within the message.

The client will not always be using the service so the server should be able to securely store the user's data for later use. Using the sealing mechanism available, the user's data can be stored on disk and only retrieved by the computation enclave. Care must be taken by the developer to ensure that no malicious user input could cause the processing enclave to open another user's file. There are many methods of isolating files on a user-by-user basis and we will not cover that topic in this paper.

6.2 Enclave Attestation

To prove to a client that the server is running the correct software, a service can employ a central attesting enclave. Instead of asking each enclave in the fabric to attest itself the client only has to check the central attester's signature. The central attester would then be able to locally attest each other enclave in the fabric to provide a comprehensive report of the server's software. However, there is an issue with mutual attestation of enclaves that was also explored by Beekman, Manferdelli, and Wagner [9]. When writing an enclave, the developer knows which other enclaves it should be interacting with. In order to locally attest another enclave, the enclave must know what value to expect in the attestation report. Otherwise, any enclave can be mistaken for the intended companion enclave.

There is normally a circular dependency problem with two-way local attestation of enclaves, since adding the signature for the opposing enclave will change the signature for the current enclave, and vice versa. Figure 13 walks through several iterations of an attempt to preprogram an enclave with the attestation

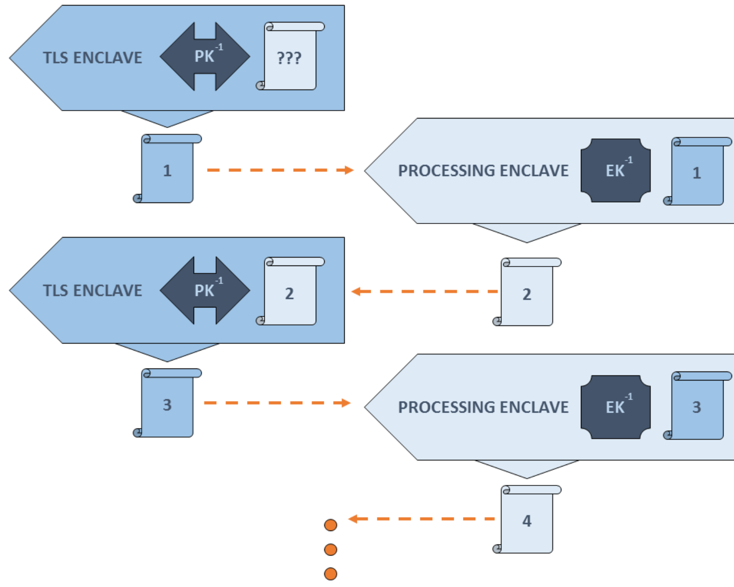

Figure 13: Attempting to Resolve Mutual Attestation with Hard-coded Measurements.

value to expect for another enclave. When the TLS enclave updates its expected report value, its own report value changes. The processing enclave needs to attest the TLS enclave and will attempt to include the expected values in its code. In doing so, its own attestation value changes and the TLS enclave must update. This goes on ad infinitum so instead we will use unidirectional attestation, where only one enclave has the expected measurement of the other.

For attesting multiple enclaves, it is straightforward to create an enclave the sole purpose of which is to attest the other enclaves. This Central Attestation Enclave will perform the local attestation process with each other enclave without reciprocation. In this manner it can easily provide a guarantee that all enclaves used by the program are valid.

The client can then trust this central enclave and still get a report of all the connected enclaves. Our prototype does not use such an enclave as we hope to eliminate the need with our attestation model. Our attestation model requires that we attest the validity of the TLS enclave used to set up the TLS session. Any additional attestations would add complexity to the model. Some services might find this structure beneficial if not implementing our attestation model. The approach shown in Figure 14 also serves to solve a weakness in the fabric architecture's security guarantees. The client is mainly communicating with the TLS enclave throughout our proposed interactions, but the TLS enclave is only a small part of the software and therefore can only attest to a small part of the program's authenticity.

If another service provider were running a different service that used the same TLS enclave, the client would receive matching attestations of the software and would only have the capacity to determine that the programs varied in company identity and hardware. Without combining the attestations of all the enclaves, the client has no guarantee about the actual software being run on its data. A possible solution would be to attest the Central Attestation Enclave after the session has been created. This would combine the benefits of our attestation model with the benefits of a Central Attestation Enclave while increasing the number of messages needed to form a trusted channel.

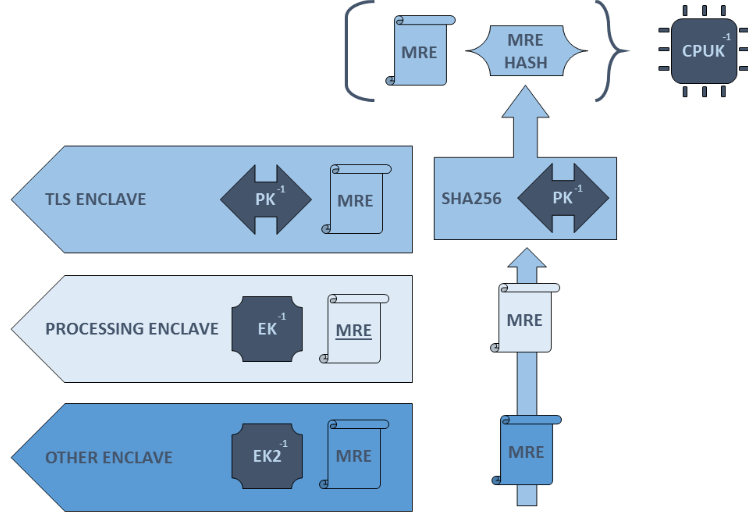

Figure 14: Attesting a Fabric of Enclaves with Combined Attestation.

Certificate Structure Our approach aims to combine TLS certificates and SGX quotes to provide the client with the necessary guarantees without the need for the IAS (covered in §4.1.3). In this section we will use the format $[\text{info}]_{signingkey}$ to describe any set of information (info) that is signed by a key (signingkey). The first set of information we deal with is the standard TLS certificate. A company owns a public key (PK) and a private key (PK^{-1}). There is also the trusted third party certificate authority with its own public key (CA) and private key (CA^{-1}). This information is provided as the first entry in a certificate chain:

$$\texttt{[PK]}_{CA^{-1}}$$

The SGX information includes a public Intel key (CPUK) and a group member key (CPUK^{-1}) owned by the attesting CPU. The CPU can use CPUK^{-1} to sign a quote from an enclave that it is running. The quote includes a measurement of the enclave software (MRE) that is a hash of the enclave's startup characteristics. The enclave also has its own public and private key pair (EK/EK^{-1}). Using the quote's user-data field we will include EK in the quote structure that is signed by CPUK^{-1}. We let each company produce its own certificate to include as the second entry in the certificate chain. The entry includes all of the SGX-related data and is signed by the company:

$$\texttt{[MRE, EK, [Quote(MRE, EK)]}_{CPUK^{-1}}\texttt{]}_{PK^{-1}}$$

Handshake Process In a simple TLS handshake, the client initiates a connection with a server by indicating it wants to make a connection and including a nonce and the supported cipher suites. A diagram of the handshake and the involved components can be found in Figure 15. So far, the only change our protocol would make would be that the client includes an appropriate entry in its cipher suite listing in the HELLO message. The server responds if it is available to make a connection (second HELLO) and includes its own nonce. After that, the server also sends its modified certificate, containing the public certificate for the Enclave Key (EK), the expected value of the TLS Enclave's measurement (first MRE), and the quote (second MRE and EK), and the remainder of the certificate chain (represented by the PK signed by CA^{-1}) to the

client for authentication. At this point the client needs to check the company's signature of the modified certificate with PK. The quote can then be verified with CPUK (publicly known and available) and equivalency checks between MRE, EK, and the same fields included in the quote. If all fields are authenticated correctly the client and server can carry out a Diffie-Hellman key exchange as in standard TLS and can begin secure communications.

Guarantees The company's PK is signed by CA^{-1} if and only if the company has been validated by the certificate authority. A signature created by CA^{-1} can be verified by CA which proves that the certificate authority has reviewed the company and that the company was issued PK^{-1}. If the client can verify the certificate authority's signature then they can be sure that the server they are contacting is controlled by the company listed on the certificate.

The MRE and EK are assumed to be tied by design. The enclave seals EK^{-1} to its Enclave Identity (at time T_0 in Figure 16), so any communication encrypted with EK can only be decrypted by an enclave with a measurement of MRE. Each SGX processor holds a $CPUK^{-1}$ (received via the Intel Provisioning Service) that can prove that it is within a group of processors that contain a correct SGX implementation. Each enclave quote produced is signed with $CPUK^{-1}$ (at time T_1 in Figure 16) so a client can verify that it came from a legitimate SGX-enabled CPU by verifying the signature with CPUK. Note that Intel's CPUK can only verify that the CPU is one of many CPUs that uses SGX, preserving the anonymity of the associated party. A quote that includes an MRE, EK in the user field, and signed by $CPUK^{-1}$ can guarantee a client that, at the moment the quote was created, a correctly implemented SGX CPU was running an enclave identified by MRE and that enclave controls EK^{-1}.

By signing the MRE and EK with PK^{-1}, the company is claiming to be running the service with a measurement MRE that has control of EK^{-1}. The quote including MRE and EK proves that, at one point in time, the service was being run on said company's server. EK^{-1} is assumed to be protected under the SGX data protection guarantees so if software with measurement MRE has controlled EK^{-1} at any point in the past, then EK^{-1} will never be accessible by any software that does not have a measurement equivalent to MRE. From a client's perspective, only software with measurement MRE on that one specific machine could possibly decrypt data encrypted with EK. This makes it possible to establish a secure connection with a specific enclave even if the quote is not fresh.

Figure 17 shows how the client processes the certificate chain. When the client is initiated it gains access to CPUK and CA, loading them in from external files. At time $T_{connect}$ the client receives the modified certificate chain from the server. The public key certificate part of the modified certificate containing EK and signed by $PK^{-}1$ is verfied as a normal TLS certificate would be, following the chain of signatures until a root certificate the client trusts is found. The quote part of the modified certificate is verified by using CPUK to confirm the signature is valid and comparing the expected values of EK and MRE contained in the certificate to the values contained in the quote. If all of these values match and are correct, then the client can be assured of the guarantees outlined above, namely that $EK^{-}1$ is solely held by an enclave with trusted measurement MRE on a platform with a genuine SGX processor which is a member of group CPUK and has a publicly known identity backed by a trusted CA.

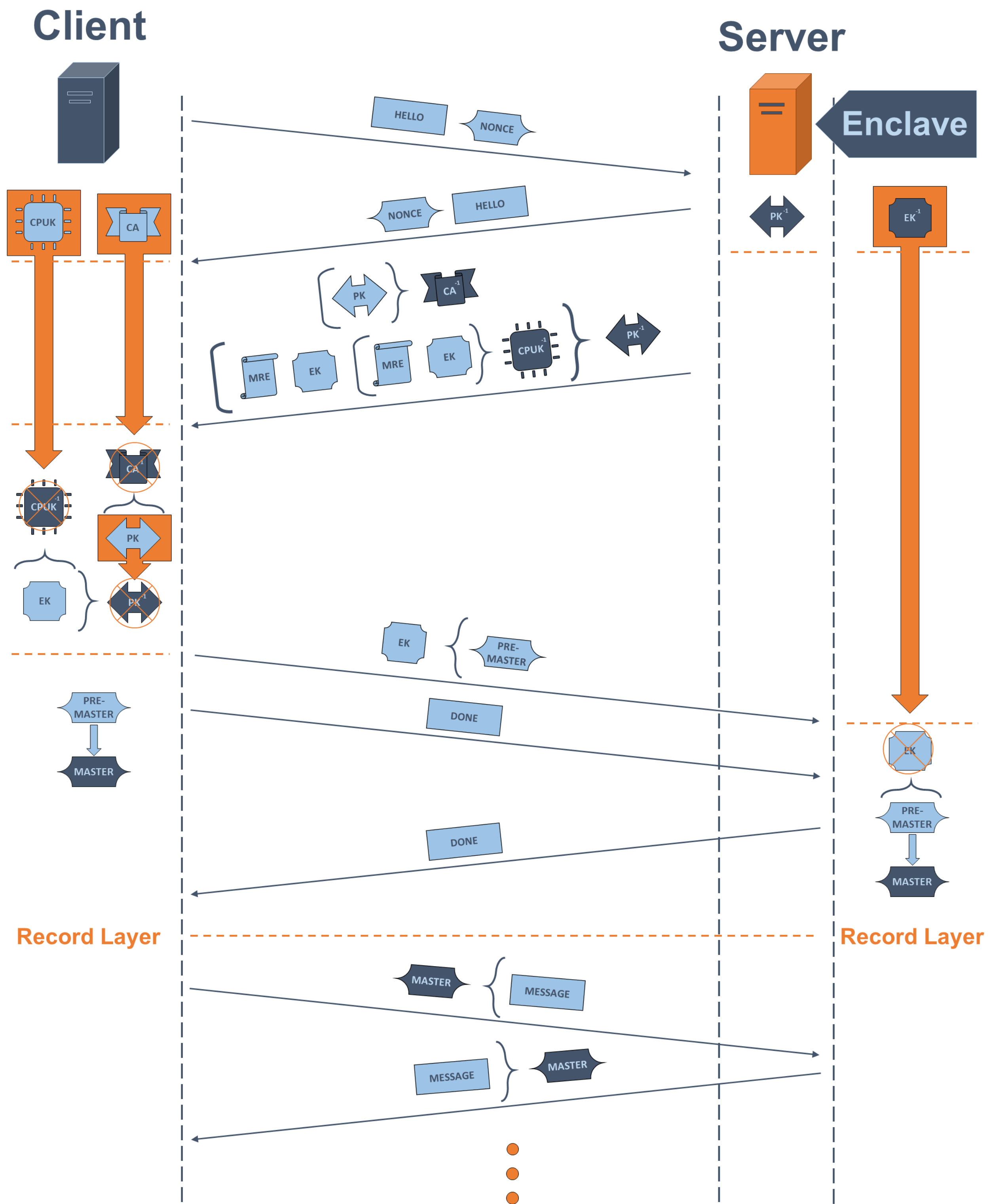

Figure 15: Handshake Process with Combined TLS and SGX Certificate.

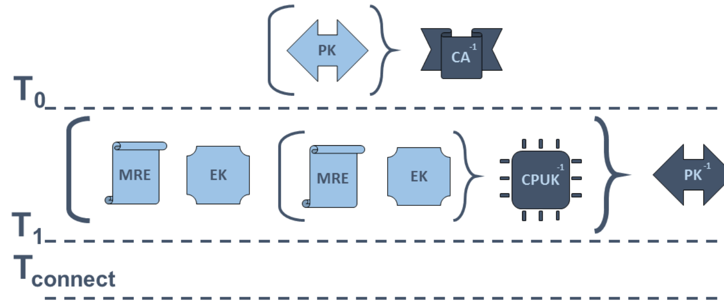

Figure 16: Timeline of Data Signing on Server using Combined Certificate.

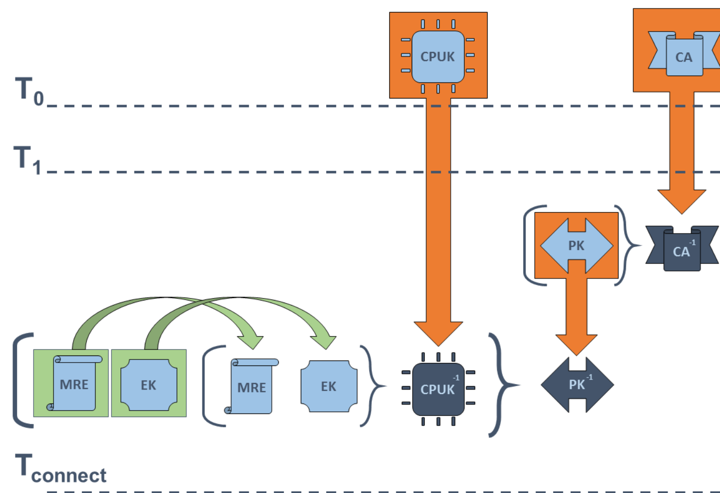

Figure 17: Client-side Timeline of Data Knowledge using Combined Certificate.

6.3 Quote Transparency

A company can claim to provide a service with the measurement of MRE but a client is only able to take on good faith that the service is truly secure and not leaking data despite being run in an enclave. For a client to have a guarantee that the service itself is secure there must be a way for the client to independently verify the software being running and what quote that software should produce. To understand what is required for this level of transparency we will examine the data that is included in a quote structure. The quote was detailed in Tables 1 and 2.

6.3.1 Quote Verification

To verify the connecting software, the client will use the data within the report part of the quote. The most important parts to verify are the enclave attributes, measurement, and report data. To do so, the signer measurement, product ID, and enclave security version can be used to find the expected values of the above data.

Finding those expected values requires an infrastructure to provide clients with not only expected values, but also knowledge of the associated program's security. A transparency infrastructure as described by Beekman, et al. [9] is one method of providing the clients with the required guarantees and information while securing corporate secrets. Making software open source provides the same transparency with an overhead of clients maintaining acceptable lists rather than a remote party. Under a fabric architecture of enclaves, a single TLS enclave could support incoming connections. An open sourced enclave could then be used in many programs and with a small enough code base within the enclave, the secure functionality could be easily maintained and checked by many developers.

For a client to check that the open source code is the same code running on the server, the client will need to be able to download the code and run it on their own SGX-enabled machine. There exist command-line tools that can be run to determine the report data values. These tools are useful for developers making the application [30] but provide little usability for an average user of a wide-reaching web service. There is an opportunity to close the gap between developer and user usability with a tool that accompanies every secure service. Such a tool would make checking enclave measurements as simple as checking a SHA measurement on an executable. The developer could also put a signed, precomputed report online for those that do not want to take the time to check it themselves.

There are certain fields within the quote that cannot be replicated as they are platform-dependent. A client cannot reproduce the Group ID, QE Security Version, Basename, CPU Security Version, Enclave Security Version, and of course the Signature. The above fields only provide information about the security of the hardware which can be verified using Intel's public group key. The fields that are related to the software itself can be reproduced on any machine. The Attributes, Enclave Measurement, Signer Measurement, Product ID, Enclave Security Version, and likely the Report Data should be the same in every instance. Once a client has determined for themselves what the measurements are supposed to be, they can store these results until the next update occurs. For widespread adoption, we need an infrastructure similar to the one in Beekman, et al. [9] but to get developers working towards fully secure software sooner, clients will need to accept the time cost of double checking the measurements against the source code. Upon first connection to the updated service, the client should be warned that the software is unexpected and a check of the source code is needed. A standardized tool for easily measuring the enclave would be useful for early adopters of the system.

6.4 Alternative Attestation Model

During our development of an attestation model, we considered a method of performing attestation that uses a two-step process to initiate a connection. In this method, a client would complete the TLS handshake with a server and, once a connection has been established, the client would then challenge the server to attest its software. The server thus receives its certificate before runtime from a CA (at time T_0 in Figure 18) and loads it. Then at connection time ($T_{connect}$) it creates the quote it will use to finalize the session setup. Compare to our method, where the quote is created once, after the private key has been provisioned to the enclave.

For each connection, the server must complete the attestation sequence in addition to the regular TLS sequence. If the TLS handshake can be completed using PK, the client can be assured that it is communicating securely with a server within the company that owns PK^{-1} and any quote provided must come from

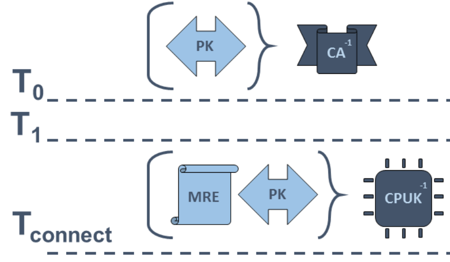

Figure 18: Timeline of Data Signing on Server using Two-Step Handshake.

one of the company's servers. This gives the same guarantee as signing the quote with PK^{-1}.

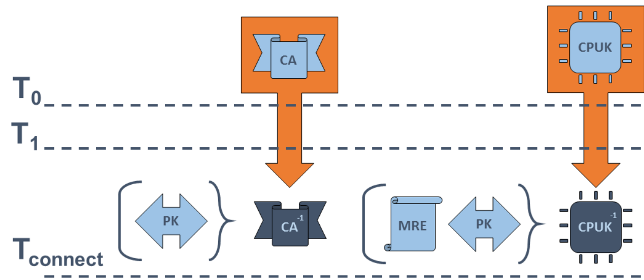

Figure 19: Client-side Timeline of Data Knowledge using Two-Step Handshake.

To validate the server, the client first loads its trusted root CA certificates (CA in Figure 19) and the EPID public keys (CPUK) from external files. Then, in order, it must validate the certificate chain beginning with PK and going until CA is reached before validating the quote with CPUK and ensuring that MRE is an acceptable enclave measurement.

Comparing Figure 20 to Figure 15, it is apparent that our proposed method is far more compact, fitting within a normal TLS handshake, while the two step method requires another set of messages. The two step method also requires that the quote be created at connection time because it does not have an Enclave Key to serve as proof it was set up properly and is still currently running. We have not tested the performance degradation caused by many consecutive attestations but there are other issues with a two step method.

As with regular TLS connections, the company's server must keep PK^{-1} readily available to complete each TLS handshake. This can be a security risk as a compromised server holding the private key can compromise the entire company's integrity. Using our protocol, PK^{-1} does not have to be used past creating the modified certificate. A company would have more options of how to securely store their private key using our protocol.

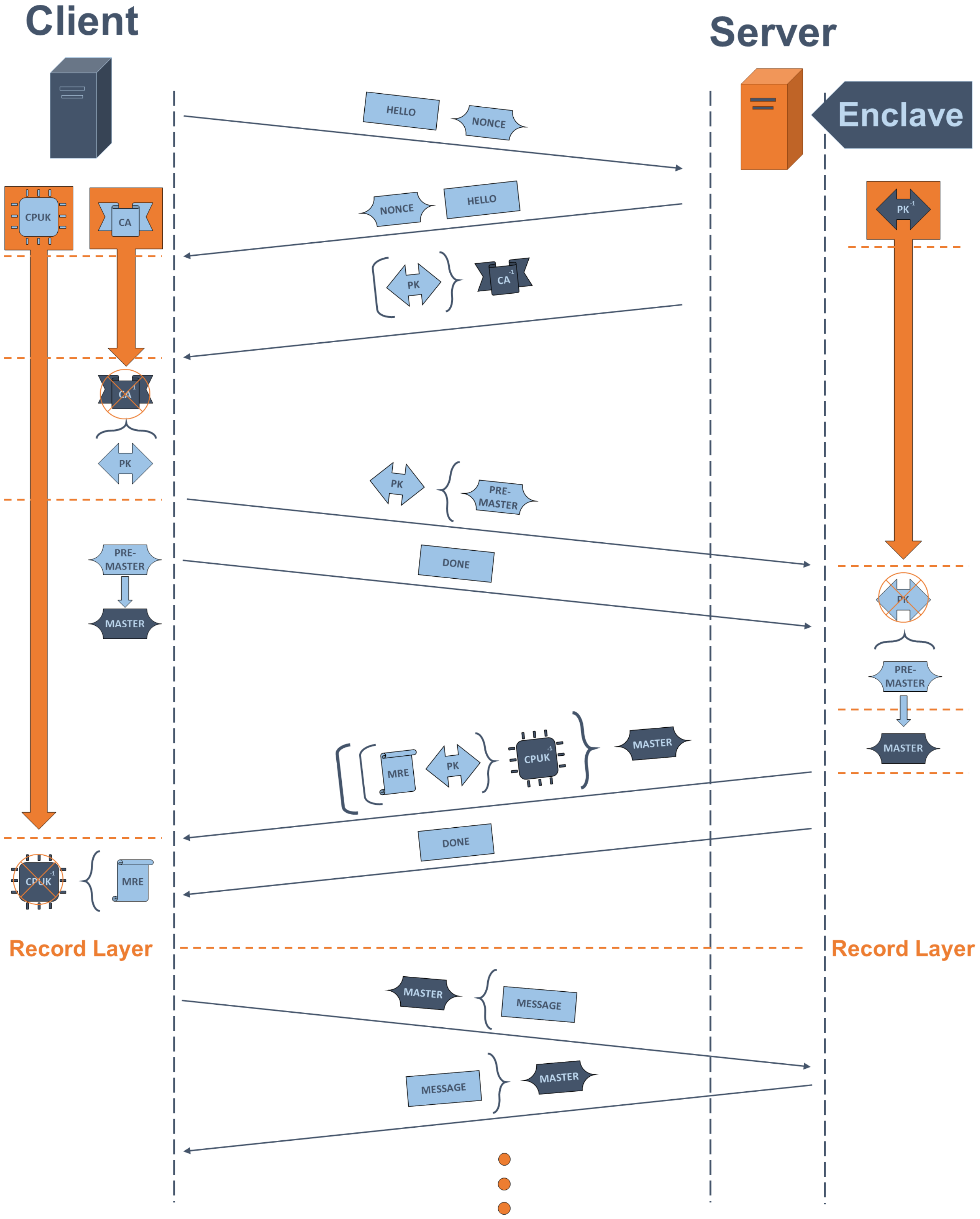

Figure 20: Handshake Process with TLS Handshake and Standard Remote Attestation.

7 Design and Implementation

In this section we cover the steps taken within our program to implement the protocols described in the Approach Section §6.2. The implementations described have certain security features cut so that we could measure specific performance metrics.

7.1 Two-Step Connection

To implement our naive approach for enclave connections, we used an echo server that would repeat messages sent to it back to the client. Setting up such a server using regular TLS is a trivial task but getting it to work as an enclave is much more difficult. We will discuss the difficulties associated with creating enclaves in §8.3.

Standard TLS connections require the setup of a context structure with which connections can be established. This context structure contains all the information needed to create a secure TLS connection such as if the agent is acting as a client or server, which protocols are supported, and what source of randomness to use. More importantly, a context structure includes the private key of the server and the certificate to send a connecting client. To handle the private key securely, we use our enclave for storing and loading the key. The private key is held in the context structure so the structure cannot be held outside the enclave. The structure or a pointer to the structure cannot be passed outside the enclave without revealing the private key so all functions to create connections are implemented within the enclave. The context structure also specifies the verification function and depth to which a received certificate needs to be verified.

The context structure is then given an input and output structure such as a BIO or file descriptor so that it can communicate over the network. After that, a server can block until data is available on the I/O structure. When a client attempts to connect to the service, the server will attempt the standard TLS handshake with the client connected to the I/O structure. This will return success or failure based on network integrity and on if the client was able to verify the server's certificate against a known CA's public key. If it returns success, the server and client will have a secure channel over which they can exchange messages.

The first message the server sends is the SGX enclave attestation quote. Normally a client must send a challenge message to a server to request an attestation. We are assuming that the client is requesting an attestation by initiating a secure connection. This can be made more robust by including a new protocol in the TLS cipher suite and, if the client and server agree to use that suite, then the client will be ready to receive a quote at the appropriate time. To send the message, we use the SGX-provided functions from within the enclave to gather a quote and get it signed by the quoting enclave. This function causes the enclave to create and pass a quote structure to the quoting enclave. The quote structure is verified, signed, and passed back to the enclave that can then send it over the secure connection. The quote is only verifying the code that is handling the TLS connections because that is the only part of the program that is running within an enclave. This could be extended using the fabric architecture's central attester method that would include attestations from all the other enclaves in the quote. This is not included as we were trying to measure the performance of a single quote. We expect a method using a fabric architecture to experience a linear degradation in performance in respect to the number of enclaves that need to be attested.

The client follows similar functionality. The client must set up a TLS context structure that specifies

that it is acting as the client. A protocol list and source of randomness must be provided as well as a CA certificate list. Any received TLS certificate will be checked against the appropriate CA certificate. In our simplified example the client does not use a public and private key although in some secure applications the client should also be authenticated to the server. The client also does not use an enclave to handle keys.

The client creates a similar BIO or other I/O structure to allow the TLS context structure to communicate over the network. Then the client will attempt to connect and initiate the handshake with the receiving server. The certificate that is received from the server must be checked against the CA list and the client will decide to continue the connection or send a close connection message depending on the results. If the connection is continued, the connection will enter the record layer where secure communications can occur.

The client will accept a signed quote structure from the server. The signature must be checked against Intel's public key to verify that the attesting enclave is running on an authorized CPU within a SGX-enabled group. The MRE should be checked against known and trusted enclave measurements from the transparency methods we describe above §6.3. We skip this step for simplicity and accept only the guarantee that whatever software we are communicating with is running in an SGX enclave. Checking what software the client is communicating with is left for future endeavors.

7.2 Certificate Chain

For our prototype that uses the certificate chain approach, we use the same echo server as before. The enclave is still securely handling the private key structure in the same manner. The difference comes from how the TLS certificate and handshake are handled.

To use the modified TLS certificate, we modified the `mbed TLS` library to read in and parse the new certificate structure. By replacing the functions that read a certificate file off of disk, the server can hold the modified certificate the same way it would a normal TLS certificate. We also had to modify the client functions that would parse and verify information from the received certificate. With these modifications complete, our server and client will be able to call the regular TLS handshake functions but complete the modified handshake process instead.

When a client attempts to connect to the server, the server will respond by starting the TLS handshake. When the server sends the modified certificate, the client will attempt to verify that the information is acceptable as described in our protocol (Figure 15).

The server must generate the new certificate before any connections are attempted. If the server attempts to load its certificate from a file but no file was found, it will generate a certificate signing request similar to how a company asks for a signature from a CA. The company must then use its private key to sign the request and give the program access to the new certificate.

8 Analysis

During development under SGX, our plans for the prototype service architecture and functionality fluctuated. In this section, we attempt to encourage the best practices we learned as well as share our results.

8.1 Feature Availability

Our development was completed entirely on a Linux platform, specifically Ubuntu 14.0.4, and using the Linux SDK for SGX. At the time of development, the Linux SDK was still under internal development so results found here may change as the technology matures. Here we have picked some areas of developer usability in which the Linux SDK had room for improvement and some areas in which it excelled.

An area that hindered our development process was the availability of standard trusted libraries. Intel provides trusted versions of the standard C and C++ libraries that have been modified to run within an enclave. Other libraries had not been ported to an enclave-ready state such as the OpenSSL library. Intel does provide a trusted crypto library but it is limited in functionality. It provides the standard RSA-3072, PKCS 1.5, AES (GCM, CTR), AES-CMAC, SHA-256, ECDH, ECDSA algorithms but only 128-bit security; the exception being RSA-3072 that provides about 112-bits. Developers of wolfSSL have been working on creating SGX-enabled libraries for their wolfCrypt operations but the results had not been made publicly available at the time of research. As we were attempting to build a web service, the lack of X509 certificate support and TLS handshake capabilities required us to adapt the `mbed TLS` library for enclave use. Future developers should have a range of trusted TLS libraries available for use and if special circumstances are required, adapting a library is a one-time development cost.

A quick test turnaround is almost a necessity when developing and SGX does accommodate for this. The SGX SDK includes a simulation mode so that developers can debug their software without using the true hardware. The simulation allows for greater debugging clarity and does not require a manual signature of the enclave with a specific developer key. The Intel-approved developer key is required for launching enclaves in their secure Release mode. Such a key can only be procured through direct interaction with Intel and signing an NDA. Enclaves can still be tested in Debug mode on the actual hardware but the container is ultimately insecure and cannot be used in production. Though requiring more work to release programs, the developer keys allow Intel to provide SGX protections only to legitimate services while still allowing for normal development.

The simulation mode has also appeared more reliable than the hardware it is based upon. Intel provides SGX wrapper functions that use a Diffie-Hellman key exchange and the Intel Attestation Service to remotely attest to another machine. In our experience under the Linux SDK version 1.5, the remote attestation sample code that uses the Intel-defined wrapper functions would fail upon execution on the hardware. Others in the community have experienced issues with the remote attestation sample code under both Windows and Linux alike. To complete our prototypes, we used the simulation mode which provided more consistent timing results.

8.2 Required Development Effort

The necessary modifications to `mbed TLS` were relatively minor in our Two-Step prototype as it used standard TLS methods and the built-in SGX remote attestation. The majority of modifications were made so that

the TLS private key would be protected by the enclave. Those changes alone took considerable effort to restructure the `mbed TLS` code base. While it does take a lot of consideration to develop boundaries and move functions, there was surprisingly little rewriting of code. Part of this can be attributed to TLS's relative lack of reliance on system calls which allowed for a strong reduction of the ECALL and OCALL interfaces. The effort needed for legacy code modifications was worrying for many researchers but from our experience, the application's structure and need for system calls may cause a large variance in actual effort required.

Extending the `mbed TLS` libraries to accommodate our extended certificate would reuse the same modifications of the architecture used for protecting the private key with the enclave. On top of the restructuring, the functions within `mbed TLS` that handle sending, reading, and parsing certificates need to understand the new certificate structure. It would be possible to do so by just replacing, not extending, the existing functionality of x509 certificate handling. The underlying data structures would be shifting, not the operations done on them so no extra system, or otherwise forbidden calls would be added. The absence of extra calls would hopefully lead to a relatively straightforward implementation.

8.3 Enclave Restrictions

Writing SGX-capable code imposes several limitations on the programmer. Enclaves are unable to execute functions outside of user mode (ring 3) and require explicit definitions of ECALLs, OCALLs, and their parameters. Enclaves also are necessarily opaque to the non-enclave program and necessarily cannot execute untrusted code, which can introduce other problems when adapting prior written code to use SGX.

Enclaves are unable to execute all of the 32-bit or 64-bit x86 instruction set. This is meant to protect the host computer by limiting the operations an enclave can perform [47]. These operations include I/O, system information gathering, system calls, and any other operation that could change the privilege level of the executing enclave. For the same reasons, an enclave cannot enter another enclave. A list of all illegal x86 instructions is included in Table 3.

Instructions	Reason
CPUID, GETSEC, RDPMC, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC	May cause a Virtual Machine exit. A Virtual Machine Manager cannot emulate enclaves.
IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD	I/O instructions that may cause a fault or VM exit.
Far call, Far jump, Far Ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/FS/GS, SYSCALL, SYSENTER	Enclaves should not rely on descriptor tables. Other instructions may cause enclave to change privilege levels.
LAR, VERR, VERW	Provides kernel information which may be used to aid exploits from within the enclave.
ENCLU[EENTER], ENCLU[ERESUME]	Cannot enter an enclave from within an enclave.

Table 3: Illegal Instructions in an Enclave [29]

These instructions are used by several common functions in the standard C library, such as `signal()`,

raise(), exit(), rand(), time(), and most of the functions in stdio.h. The wide character (wchar.h) counterparts to stdio.h functions are likewise prohibited. Finally, the string functions strcpy(), strcat(), and strstr() have been deprecated by Intel and are illegal functions in the enclave. SGX provides its own randomness instructions but they can be interrupted by a malicious hypervisor. The hypervisor can cause a VM to exit or force the use of weaker randomness by interrupting the random value return. This could cause a rollback attack if the program switches to an insecure method of generating randomness. If weaker randomness is prohibited the hypervisor is just performing a mild denial of service attack.

Pointer arguments are passed to and from the enclave by creating buffers containing copies of the original data and using pointers to those buffers. Figure 21 shows a the treatment of a pointer used as both input and output. At time T_0 the contents of the buffer pointed to are copied to the new location inside the enclave. The enclave performs its operations on the buffer at time T_1, and then returns. Upon returning at time T_2, the contents of the buffer are copied over the original data from the enclave. The reverse of this process, as well as either the first or second half alone can be done as well. This covers the range of possibilities for pointer argument to either an ECALL or an OCALL. The buffers must be created at the time of the ECALL or OCALL, so the maximum size, at least, of the intended input or output must be known at that time. The exception to this requirement are null-terminated strings, whose size can be calculated, when used as input [26].

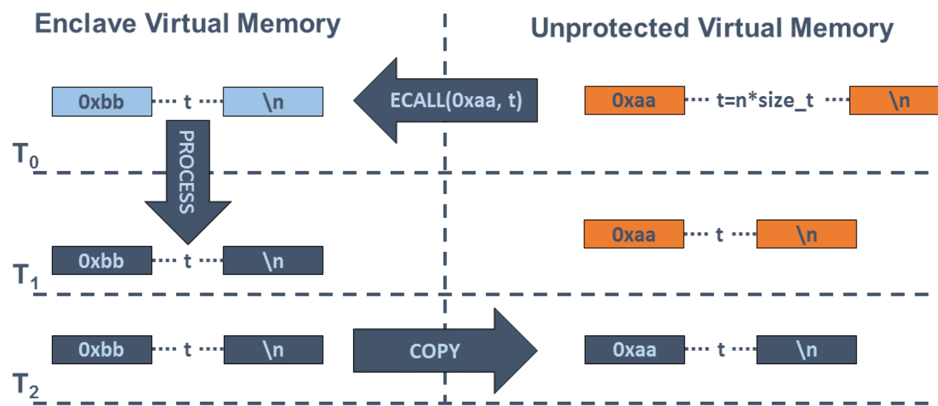

Figure 21: Pointers Crossing Boundary

The buffer size is determined by multiplying the pointed-to item's size by its count. The size is the number, in bytes, of an individual item, such as a char $\bar{1}$. The size of the item can be set manually, can be inferred from the pointer type, or can be determined using a "size function". A pointer to a char would be inferred to have a size of 1. The size function is called at the time the buffer size is calculated. It recieves the pointer and should determine the appropriate size of that item. For generic pointers (void *ctx), the size must be specified manually or using a size function. The count of a pointer is assumed to be 1 unless specified otherwise. Helpfully, count and size designations can reference other parameters. For example, a function foo(const char *buf, size_t buf_len) could reference buf_len as the count of buf.

This requirement complicates some of the usual assumed behaviors of functions when passing pointers as arguments. Enclaves will be using the pointers to the buffers instead of the actual pointers (though pointers copied into a buffer can be used to access that data), so it is more difficult to operate on pointers. This

is especially true with output pointers, as enclave functions can only work on the copied buffer, whose size must have been determined beforehand.

The above mentioned limitations have their greatest impact when attempting to adapt legacy code to use SGX. The first step is to determine what to protect with SGX enclaves, either particular data or a certain operation. The next step is typically determining which functions interact with the protected data or perform the protected operation and prepare to move them into the enclave. This is where the limitations become apparent. The legacy code was written without knowledge of the illegal instructions, so functions using those instructions need to be removed from the enclave code. If those removed functions are necessary for the enclave code to operate properly, then OCALLs must be created for that purpose. Then all pointer sizes and counts must be determined. Many functions write a certain number of items to a pointer, but that number is unknown at the time the function is called. Functions that rely on generic pointers may be troublesome if they serve as ECALLs or OCALLs, because the size function will have to determine the size of the pointed-to struct with little information about the data given. Finally, functions that rely on function pointers cannot call those functions while inside of the enclave. These limitations can seriously impact the general and polymorphic qualities of the legacy code if the enclave boundary (the location where ECALLs and OCALLs are made) does not take these requirements into account.

Some of these limitations can be overcome with additional mechanisms or by changing the enclave boundary. Generic pointers to structs can be dealt with by including the struct type as the first member of the struct, so that the size function can determine the actual size from the type. Callback structs to enclave functions can be stored in the enclave, so long as they contain no references to non-enclave functions and are never accessed outside of the enclave. Function pointers to non-enclave functions can be replaced by a table index into a table of OCALLs, with a similar result. Finally, the enclave boundary can be moved to a place where the sizes of the pointed-to items are known, or where the enclave is operating on the original pointed-to data directly, as it would in the normal program.

To better explain how moving the enclave boundary might work, we provide the following example. Consider a series of functions that call each other based on the polymorphic behaviors of their arguments (shown in Figure 22). The context struct `my_context` contains the necessary information for processing `in_buf`, such as the eventual size of `out_len`, as well as a context-specific function, `barX`, which is used to generate the output. Situating the enclave boundary at `foo` would be problematic, because the needed size of `out_buf` is unknown at that time. Moving the enclave boundary to `barX` may resolve the issue, if leaving the output length unprotected by the enclave is allowable. Alternatively, moving the enclave boundary up a level to `foo_parent`, which created both the input and output buffers as local variables, would protect the output length without the problems of `foo`. `foo` and `barX` can operate as normal on the buffers, without concerns over the necessary size of the output buffer. This will result in the addition of all possible functions for `barX` to the enclave, as well as a function to set `barX` with access to those possible functions in the enclave.

8.4 Application Design

When designing our prototypes, we aimed to follow the fabric architecture previously described. Reducing the trusted code base to protect certain secrets takes considerably more effort than putting the entire program within the enclave. An interface with the enclave must fully encompass all uses of the secret that the enclave

```
int foo_parent(struct my_context *ctx)
{
    char in_buf[16];
    char *out_buf;
    int out_len;

    /* Get input */

    foo(ctx, in_buf, 16, out_buf, &out_len);

    /* Use out_buf */
}

void foo(struct my_context *ctx, const char *in_buf, int in_len, out_buf,
         int *out_len)
{
    *out_len = ctx->output_len;

    ctx->barX(in_buf, in_len, out_buf);
}
```

Figure 22: Example code.

is protecting. Creating said interface requires considerable forethought about how to isolate the necessary components.

Our approach to isolation in a TLS enclave began by attempting to place only the functions that directly dealt with the private key inside the enclave. The original functions from the `mbed TLS` library were adaptable to the enclave but as described above, there are numerous issues when adapting legacy code to cross the boundary between untrusted and trusted environments. Consequently, the ECALL and OCALL interfaces are best minimized. This is doubly true as a general design principle when dealing with sensitive information. Providing fewer opportunities to accidentally reveal any data about contained secrets is generally safer and easier to maintain.

Figure 23 illustrates a program flow where the enclave boundary is crossed many times. Reducing the OCALL interface requires functions within the enclave to avoid forbidden instructions and calls. Minimizing the ECALL interface encourages the creation of fewer interfacing functions while encouraging the inclusion of many helper functions within the enclave. Creating fewer interfacing functions means the enclave boundary is crossed fewer times, increasing the simplicity of the interface. Each function within the enclave may have many helper functions and each of those functions either need an OCALL or need to be included in the enclave as well. As long as the included helper functions do not need to be called by any of the functions outside the enclave, no ECALL reference must be included. As long as the helper functions do not use any restricted instructions or functions, including them within the enclave helps to minimize both the ECALL and

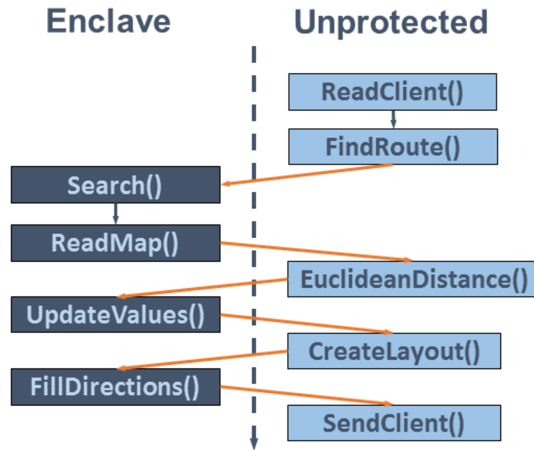

Figure 23: ECALL/OCALL Interfaces without Reduction

OCALL interfaces. Figure 24 illustrates how these principles changed the program flow from Figure 23. The colors have been maintained to highlight the difference. Note how there only a single ECALL in the reduced interface, where the unreduced interface had two OCALLs in addition to the ECALL. The developer of the program in Figure 24 no longer has to worry about whether the OCALLs may leak sensitive information being protected by the enclave or about the sizes or counts or arguments to those OCALLs.

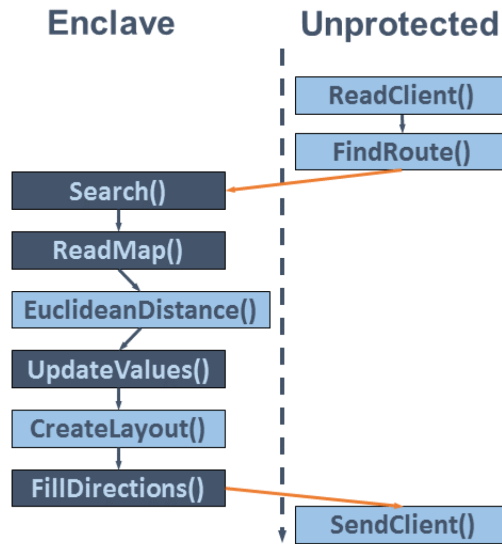

Figure 24: Reduced ECALL/OCALL Interfaces

As previously discussed in §6.1, developing a service as a monolithic architecture is another possibility that, while losing the security of better isolation, can be easier to develop for. Only making an ECALL to start the service and keeping all computation within the enclave would make a fast conversion from legacy code to an SGX-enabled service. This is only possible however, if the service requires no banned instructions or functions which is very unlikely. Some time gained by not changing the service's architecture can be lost

checking every part of the program for these calls.

8.5 TLS Enclave Design and Consequences

The `mbed TLS` library manages the TLS session through the use of a context structure. This structure contains all of the configuration and state data, input and output buffers, private keys, trusted certificates, and other attendant data. The handshake is managed as a state machine, where the context structure is run in a loop until it reaches an end state. On each iteration of the loop, a different function is called based on the current state.

The enclave encompasses the dispatcher "handshake step" function, with the goal of protecting the private key of the server. In the planned full implementation the private key would be sealed by the enclave while control of the program is outside of the enclave and would be unsealed when it is needed inside the enclave. The untrusted code runs the loop function as well as all of the setup functions, with the exception of a single function (`ssl_handshake_init`), which set certain callback functions that needed to be in the enclave. A full implementation will likely include more of these setup functions in the enclave, as some load or set sensitive data, such as the private key load function. The context structure is kept in untrusted space, because the input and output buffers needed to be accessed by the untrusted I/O functions.

These decisions carry certain consequences. The context structure's members are vulnerable to manipulation by privileged malware because the structure is kept outside of the enclave. There were only plans to seal the private key outside of the enclave, but because of the information contained in other members, those may need to be sealed as well. Possible candidates for sealing include the handshake state, root CA certificiates, certificate revocation lists, symmetric key exchange data (such as the Diffie-Hellman-Merkle exchange prime modulus and generator), flags for encrypt-then-mac, renegotiation, and anti-replay options, the socket descriptor, enclave ID, and the certificate verification function pointer. Each of these has the potential to derive the session key, interfere with verification processes, or weaken the security of the resulting session.

A first instinct may be to seal the threatened structure members in a similar manner to the private key. This decision would then lead to the creation of an ECALL to access the each member in untrusted code. Calls to these functions would rapidly grow the number of ECALLs and the associated complexity of the code. For example, a hypothetical "get state" ECALL would be called at least 4 times in the untrusted library, with potential to grow to 20-30 calls.

Taking the approach outlined in §8.4, it seems more appropriate to expand the enclave boundary again, attempting to encompass the entire context structure. For `mbed TLS`, this will likely involve rewriting the "read record" function so that it can read input without requiring the entire context structure to be passed out of the enclave and copied back in after the read is complete.

More positively, the enclave as it is currently constructed is capable of protecting the server's private key. Once the sealing aspect is added to the enclave, the private key will never be in an unsealed form outside of the enclave. The current size of the enclave also protects the symmetric session key negotiated during the handshake and the cipher operations used during message encryption.

8.6 Performance

Our protocol aimed to be a scalable solution for widespread web services. To determine the feasibility of the prototypes, we used a combination of theoretical analysis and benchmarking. Our results were compared

against a standard TLS handshake with no optimizations.

8.6.1 Individual Time Costs

Seeing as a TLS handshake involves several messages being passed between endpoints and computations being done on both sides, we can split the process into basic components to gather an estimated cost of performance. Expected costs include message travel time and cryptographic processing. The total cost can be expressed as:

$$Trips * Latency + \sum_{Function} NumKeyOp_{Function} * CryptoCost_{Function}$$

Where *Trips* is the total number of unidirectional messages that must be acknowledged by the remote party. If a message is sent immediately following another message and the listener only responds after both are received, only one trip is counted. The *Latency* is the average time in milliseconds for a message to reach from the sender to the listener. Finally, we can count the number of cryptographic operations that must be completed using each cryptographic function. There is a large variance in performance of each cryptographic function so the total cryptographic operation count must be separated by *Function*.

To determine the time needed for each cryptographic function, we used OpenSSL's Speed tool that finds the time required to complete every cryptographic operation available in OpenSSL. Tables 4 / 5 contain the results of the timing tests using an Intel quad-core i7-6700 Skylake processor with eight logical cores running at 3.40GHz.

Operation	Key Size (bits)	Sign Cost (ms)	Verify Cost (ms)	Signatures/s	Verifies/s
rsa	512	0.037	0.003	26862.3	368895.6
rsa	1024	0.109	0.007	9194.5	133779.9
rsa	2048	0.846	0.025	1181.5	39475.8
rsa	4096	6.378	0.095	156.8	10486.2
ecdsa	160	0.0	0.2	22896.3	6501.9
ecdsa	192	0.1	0.2	18785.6	5356.4
ecdsa	224	0.1	0.1	18463.2	8543.9
ecdsa	256	0.1	0.2	12606.0	5375.0
ecdsa	384	0.1	0.6	6714.9	1579.3
ecdsa	521	0.3	0.8	2924.6	1253.2

Table 4: Asymmetric Signature Performance

The operations used in a TLS handshake are negotiated between the client and server so we can only provide an average case performance for an expected connection. From the selected algorithms above, the average asymmetric key operation takes on average 0.797 ms to sign data, and 0.223 ms to verify a signature. The amount of data transferred for every connection that must be put into the cipher function is variable and would be difficult to predict. Even the slowest AES cipher in Table 5 can process 127,086,574 bytes per second, or 127 kilobytes per millisecond. Perhaps the largest chunk of data send during the handshake is the RSA or ECDSA key. The largest asymmetric key that is currently being used is a 4096 bit RSA key. Such

Operation	16 byte block	64 byte block	256 byte block	1024 byte block	8192 byte block
md5	80993.90k	230142.78k	491341.48k	695994.37k	789297.84k
hmac(md5)	67391.88k	200838.14k	457282.05k	674694.21k	785645.46k
aes-128 cbc	160581.35k	177055.28k	181814.61k	182897.97k	183457.25k
aes-192 cbc	135894.17k	148541.12k	150825.25k	152067.74k	153034.75k
aes-256 cbc	118075.53k	127437.67k	129172.07k	130391.38k	130356.22k
sha256	69140.92k	151098.84k	257955.38k	317063.85k	340931.44k
sha512	55585.60k	224306.92k	331526.91k	464465.31k	524200.91k

Table 5: Bytes-per-Second Cipher Performance

a key uses 3.1% of the AES-256 average throughput per millisecond and would take about 31 microseconds to process. This much data would be processed once by both the server and client when they send a hash of all the messages sent between them. A signature verification is an order of magnitude more expensive than these operations so we will not be considering them for our analysis.

We will use a *Latency* of 28ms as a hypothetical measurement of network latency. This delay happens to be an approximate time for light to travel from New York to London through an optical fiber cable. To count the *Trips* required to complete a regular TLS handshake, we step through the messages used to complete a handshake and count the number messages that are acknowledged by the receiving party and thus have a cost of *Latency* ms:

+2 A standard TLS connection occurs over a TCP connection, requiring one full round trip for the messages.
+1 With the TCP connection in place, the client sends its hello message.
+1 The server responds with its hello message and certificate.
+1 The client sends the Diffie-Hellman key exchange.
+1 The server confirms that the handshake has been completed with a finished message.

In total, *Trips* in a standard TLS handshake is six. It can be noted that the cryptographic functions use time in the order of tenths of milliseconds, while network operations last almost two orders of magnitude longer so the majority of the cost in a handshake comes from $Trips * Latency$.

8.6.2 Standard TLS Handshake

Our protocols are measured against a standard TLS handshake so we produce an expected time as well as measured tests of a handshake performed by unmodified `mbed TLS`.

Predicted Performance Using the standard TLS handshake *Trips* and our estimated *Latency*, our hypothetical network latency is $6*28ms = 168ms$. The cryptographic functions include two RSA verifications and one RSA encryption, a Diffie-Hellman key exchange, and two AES ciphers. Assuming the use of 2048

bit keys for the RSA operations, our machine would take 0.05 ms for the verifications and 0.85 ms for the signature. Using the worst case message size as described above with AES-256, the two AES ciphers would add about 0.06 ms to the procedure. All together the cryptographic functions contribute just shy of one millisecond to the total handshake. The final projected handshake would take about 169 ms.

Tested Performance When running an unmodified `mbed TLS` handshake, our total handshake time was only 40.1% of the estimated real-world performance. The full handshake took 67.8 ms from when the client sent the first TCP message to a secure channel being set up.

Trial	TLS (ms)
1	73.920
2	69.474
3	58.469
4	70.151
5	66.493
6	66.392
7	66.559
8	69.990
9	67.130
10	69.495
Avg	67.807

Table 6: Base mbed TLS Handshake Measured Performance

Our machine is trying to connect to itself so the *Latency* value is actually very small for our tested performance. The latency between the machine and itself is about 0.025 ms which contributes to the vast difference between our test results and our expected real-world scenario.

8.6.3 Two-Step Attestation

Recall that in this approach the server and client first perform a TLS handshake, then finish a standard remote attestation. A standard TLS handshake is considered the baseline performance for our services so the only additional computation required is in the Intel-specified remote attestation of the enclave. For each new connection created, the attestation requires the server's processor to enter the quoting enclave and perform a signing operation on the report. The client for each connection must also complete another set of verifications against the quote it receives. We have also changed the TLS functions to work within an enclave so any ECALL and OCALL overhead will be incurred here as well.

Predicted Performance In a realistic situation, *Latency* for the TLS handshake and attestation messages would be different as they are sent to different end-hosts. In our prototype the *Latency* values are both equal to the time it takes for our machine to connect to itself (`localhost`). Moving the TLS components into an enclave does not add any messages sent over the network so *Trips* remains at six. In addition to those six, the attestation requires the client to make another TLS connection with the Attestation Service and transfer the Quote. The additional trips are broken down as such:

+6 A standard TLS connection.

+1 With the TLS connection in place, the client sends the attestation quote.

+1 The server responds with a verification report.

+0 The client sends a close connection request but does not wait for response.

As noted with our hypothetical *Latency* of 28 ms, the time cost to send messages is orders of magnitude higher than performing cryptographic operations so this extra connection is not a trivial cost. The extra connection uses all the same cryptographic functions and messages as the first TLS connection so we can assume an approximate total performance of $2 * BaseTLS + 2 * Latency$. On top of the network costs, the client must verify the data within the Quote with a hash operation which we consider negligible for estimation and perform a Diffie-Hellman key agreement with with server. The server must also use a cryptographic function to produce a report for the Quoting Enclave which subsequently checks and uses another function to sign. An RSA function with a 2048 bit key took about 0.85 ms to sign and 0.025 ms to verify (Table 4. For estimation purposes, we will use 0.9 ms for the $CryptoCost_{RSA}$ and only count the pairs of these operations of which there is one between the Quoting Enclave and Attestation Service.

In our real-world estimation with *Latency* of 28 ms, the $BaseTLS$ handshake took 169 ms. Adding the $2 * Latency$ and 0.9 ms for extra computation brings the total expected handshake to 225.9 ms, or 33.7% longer. With our tests using `localhost`, the $BaseTLS$ was 67.8 ms with a *Latency* of 0.025 ms meaning our projected performance is 68.75 ms.

Tested Performance We looked to measure three areas of performance in our experimental testing: the effect of using an enclave to handle the TLS handshake, the cost of remotely attesting the enclave, and the overall time taken to establish a connection. The measurements were taken across ten trials using the elapsed wall-clock time. We are connecting to a program on the same machine so our latency is small compared to a real network connection. The elapsed time does not always directly correlate to the extra computation as both the client and server must spend relatively long periods waiting for the other's computation and network latency. However, in our tests the computation had a larger effect on the measurements due to the low latency.

The first performance consideration we observed was the context switching needed to enter and exit the enclave. There was no available timer accurate enough to measure the time lost to a single context switch so we could only observe the time difference across the entire handshake. Other developers had asked about the ECALL and OCALL context switch performance on the Intel forums but no answer has been provided publicly. We attempted to minimize our interface with the enclave and in the end we were able to reduce the overhead to sixteen context switches due to program logic during the handshake. More would be required throughout the course of a connection with a client. From our observations of the handshake process, there was no significant increase in time due to context switching. The average time to complete a TLS handshake without and enclave was 67.8 ms and with an enclave handling the private key it was 73.4 ms. The performance loss of 5.6 ms is an overhead of 8.3% to the basic TLS handshake. Given our minimal enclave interface, we cannot comment on whether ECALLS and OCALLS can truly be considered free when executing vast quantities of them.

The SGX attestation alone, from the time the program finished the TLS handshake to when the secure channel is set up, took 75.0 ms. On average, the attestation took longer than the TLS handshake, increasing the required time per connection made to the server by 102%. A lot of the time in the attestation is spent

while the is client verifying signatures and waiting for the attestation server to respond. While the extra time does affect the user's experience, the server does not need to designate a large amount of resources per connection meaning the overhead that can be used in a DoS attack is not the same 102% increase.

Trial	TLS (ms)	Attestation (ms)	Total (ms)
1	77.837	76.292	154.129
2	77.929	74.633	152.562
3	79.072	74.605	153.677
4	79.139	77.935	157.074
5	67.645	74.713	142.359
6	66.435	73.033	139.468
7	78.261	76.246	154.507
8	60.485	72.500	132.985
9	79.754	75.022	154.776
10	67.597	74.768	142.356
Avg	73.415	74.975	148.389

Table 7: Two-Step Attestation Measured Performance

To put the total 148.4ms in perspective, conventional human-computer interaction dictates that a tenth of a second be considered an instant response to a request. One second is within a human's flow of thought leading to a seamless experience. Ten seconds is the average attention span and clients will often navigate away from the page if a load time takes this long. The average load time for top-ranking websites on a solid internet connection can be upwards of seven seconds. Surveys have found that 40% of users will abandon their connection if the webpage takes longer than 3 seconds to load [18]. The two-step attestation is adding almost 150% of the detrimental load time. Websites that already take seven seconds to load have very little leeway for performance stutters or additional computation.

Some web services load slowly because they have a lot of content, but all services will start to lose performance under duress. Servers must be able to handle an immense number of connections to keep the service running. We hypothesised that the quoting enclave would become a bottleneck for performance when an SGX-enabled server needed to handle simultaneous connection handshakes. The enclave must attest itself every time a new connection is attempted. Typically services run on separate threads so that multiple processors can help scale the number of possible concurrent connections. In the SGX documentation it is unclear if the quoting enclave authored by Intel was a shared resource or microcode that could be run on any of the processors. We were unable to test the throughput of the quoting enclave but it might be a factor when considering SGX-enabled web service development.

8.6.4 Certificate Chain

This method uses a modified TLS certificate in a TLS handshake with minor changes to how the messages are processed. The number of messages exchanged are the same but for each connection, the server must send an extra long certificate to the client while the client is required to do extra computation to verify the modified certificate.

Predicted Performance Using a modified certificate requires no additional network connections as the quote is verified by the company above it in the certificate chain. Our *Trips* and *Latency* network costs will be the same as the *BaseTLS* networking costs. The added costs within this protocol include two RSA verifications and several equivalency checks that wont be considered as they do not require cryptographic functionality. Each RSA verification using a 2048 bit key takes about 0.05 ms on our machine. The total time to complete this handshake would be 169.1 ms. That is a 0.06% increase in time required to complete a handshake. We were unable to produce a prototype for this attestation model due to time constraints but we believe it is possible to develop under the current SGX environment.

9 Conclusion

Trusted execution environments can play an important role in the evolving battle against malware. By allowing programs to run critical sections of code in an environment protected against outside interference and to encrypt user data when it is not being actively used, TEEs allow programs to defend themselves against attacks originating from their host platform. Intel SGX is one of the latest TEE technologies, which we used to develop the ideas and implementation detailed in this paper.

We built a working demonstration of a client-server program using an SGX enclave. We created a TLS Enclave, which managed the private key of the server and handled the steps of the handshake with an unchanged client. ARM's `mbed TLS` library was chosen for its simple API. Development largely revolved around adapting this TEE-unaware TLS library while learning about the SGX API. Ultimately the library was adapted with minimal changes to the non-enclave code.

We developed a method of combining the Intel SGX remote attestation process using EPID group keys with TLS. The method introduced a new certificate into the bottom of the original certificate chain. This new certificate contains the public key for a new RSA key pair called the Enclave Key, and is signed by the CA-issued key issued to the machine owner. The Enclave Key is then sealed by the TLS Enclave that will use it using the Enclave Identity. This ensures that only that enclave on only that machine has access to the private key.

The remainder of the new certificate contains an Quoting Enclave-authored quote, which was created at the time the Enclave Key was sealed to the TLS Enclave. This quote is thus created once and stored rather than dynamically generated for each new session. It can be updated on a regular basis if additional liveness needs to be shown. The quote should also be updated each time the CPU or ISV security version number changes, so that it reflects what is actually running on the server.

Finally, we ended with a recommendation for the administration of the the various keys, quotes, enclaves, and certificates. We recommend generating an Enclave Key for each TLS-using program on each public-facing machine and storing CA-issued keys in an offline server. The sole purpose of the CA-issued keys in our model is to sign the Enclave Key certificate, extending the chain of trust a single certificate further. This allows the CA-issued key to be kept in a secure, offline server. For quotes to truly provide any guarantees about the enclave code's behavior, the enclave measurement and data management guarantees must be independently verifiable. We call this quote transparency. The enclave code itself must be available for evaluation and in a manner in which any interested party can independently produce the same measurement they will recieve upon connection with the enclave. This will probably involve the location and possible development of program measuring tools and the public display of enclave code at least.

9.1 Overall Evaluation

Our method combines the traditional guarantees of TLS, which confirms the identity and trustworthiness of the server to which the certificate was issued while creating a secure communications channel, with the guarantees of SGX attestation, which confirms that the given enclave is running in a given SGX environment on a genuine SGX-capable Intel processor. A successful verification of the new certificate thus affirms that a given piece of software is running in a enclave on a genuine SGX-capable processor on a particular machine representing a certified entity. Creating and using a static image of the quote rather than creating a new one

for each session will likely improve runtime performance without compromising the security guarantees of the quote. If the MRENCLAVE and source code of the TLS Enclave are publicly known as we recommend, then a validated quote and MRENCLAVE field would be enough to prove that the key used in the TLS handshake was the Enclave Key and that the TLS Enclave is thus currently running.

The remote attestation aspect of our method, when combined with the quote transparency and program measurement schemes, enables users to verify the actual program with which they are communicating. Through matching measurements of the enclave binary, clients can be assured that the key used in the TLS handshake is tied only to that enclave, and that any client data will be handled in a known manner.

These guarantees come with some additional cost to performance. Using OpenSSL's Speed tool, we calculated the time cost for two cryptographic operations (sign, verify) for both RSA and ECDSA with varying key sizes. Then we calculated the throughput of the various ciphers (md5, aes-128/192/256cbc, sha256/512). The time costs and throughput then allowed us to create an estimate of the time to complete a normal TLS handshake. This was estimated to be about 169 ms with a hypothetical one-way latency of 28 ms. Ten trials were conducted using the elapsed wall-clock time. The average time to establish a connection was 67.8 ms. The network latency was much smaller (0.025 ms) on our trial machine because it was connecting to itself, partly resulting in the significantly lower trial result. For the SGX-managed Using SGX to handle the handshake, without our certificate changes, increased the total time by 5.6 ms to 73.4 ms. This result, an 8% increase in time, indicates that the enclave context switch overhead is significant.

Then we estimated the total time for the alternative two-step method. This method required the same operations of TLS (the signing and verifying) to be performed again during the post-handshake attestation. The estimated time for the two-step method to complete the handshake was nearly 226 ms, 33.7% longer than the normal handshake. The bulk of the increase came from the additional time spent waiting for the attestation messages to travel. Ten trials were conducted using the elapsed wall-clock time. The average time to establish a connection was 148.4 ms.

For our final performance evaluation, we estimated the total time for the certificate chain method. This method has the same number of messages sent as TLS, with some extra processing of the certificate chain. The estimated time to establish a connection was 169.1 ms with the assumed latency of 28 ms. However, because of the length of the project, we were unable to complete the prototype to test its actual performance.

Furthermore our method allows the more important CA-certified key issued to a company to remain secure in an offline server, rather than kept on the public Internet-facing servers used to conduct day-to-day business. This not only keeps the often-expensive certified private key safe, but also discourages key reuse on multiple servers by allowing companies to easily and inexpensively create unique keys for each enclave on each server.

While the security guarantees of Intel's SGX are questionable against hardware and side-channel attacks, our protocol can be extended to similar architectures that provide attestation capabilities. SGX is still developing and there are opportunities for extending existing networking with the guarantees provided by TEEs. Hopefully, our exploration of SGX development and legacy code adaptation will assist the security community in evaluating the potential usability of SGX and similar TEEs.

9.2 Recommendations for Future Study

We were unable to fully implement the new certificate parsing or the Enclave Key sealing features into our TLS Enclave due to the length of the project. This suggests an immediate future project. Finishing the implementation can confirm or revise our vision of enclave development. Future developers may find a way to tighten the enclave boundaries beyond their current form, reducing the enclave's TCB, or they may find the need to expand them further in order to incorporate the loading of the Enclave Key and its respective certificate. Regardless of the changes to the enclave boundary, the ability to evaluate the performance of the full implementation, provide additional lessons learned with sealing data, and to test the enclave's robustness under attack would be invaluable to further research on SGX in particular and TEEs in general.

Future work can explore a higher-performance SGX/TLS connection implementation, the idea of an Attestation Enclave, and the applicability of our method to other TEE technologies.

Currently our implementation is a simple server capable of handling a single client connection at a time, but an actual production implementation would require the capacity to handle possibly hundreds of connections concurrently. There are several ways to do this, including event-based I/O, multi-threading, and multi-process implementations. SGX supports threading in the enclave, but the performance of this would help highlight possible strengths or areas of future improvement for SGX.

We only implemented a single enclave in our simple server, but a more complicated program could possibly require several different enclaves, one for each behavioral aspect of the overall program. We had proposed the idea of a centralized Attestation Enclave in this scenario, which would confirm the statuses of all other enclaves in the program before creating a quote in conjunction with the Quoting Enclave. Whether this is the most effective way of attesting for multiple enclaves given the single quote detailed in our method is not known at this time. Development along this line may discover some other, more effective means if the Attestation Enclave proves insufficient. This would lead to the development of a method for attesting all of the enclaves a program may use, hopefully as an extension of our method.

Finally, there are several other TEE models currently available. These include ARM's TrustZone, IBM's 4765 Secure Coprocessor, and various other TEEs. Each one of these has its own strengths, weaknesses, and capabilities compared to SGX. Finding a way to use our method for those models would help the TEE standardization effort as well as research on those technologies. Overall, we feel that our work has helped open the way for future exploration of SGX and TEEs in general.

References

[1] Alves, Tiago, and Don Felton. "TrustZone: Integrated hardware and software security." *ARM white paper* 3.4 (2004): 18-24.

[2] Anati, Ittai, et al. "Innovative technology for CPU based attestation and sealing." *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy* Vol. 13, 2013.

[3] Anderson, Ross, et al. "Measuring the cost of cybercrime." *The economics of information security and privacy.* Springer Berlin Heidelberg, 2013. 265-300. Web. 9 Oct. 2016.

[4] Armin, Jart. "The World's Community and the War on Cybercrime - What About Italy?" slide 12. *Italian Security Summit, CyberDefcon 2012* 22 Mar. 2012. Web. 12 Sept. 2016. https://www.securitysummit.it/archivio/2012/milano/upload/file/Atti/22.03.12_JART%20ARMIN.pdf

[5] Arnold, Todd W., et al. "IBM 4765 cryptographic coprocessor." *IBM Journal of Research and Development* 56.1.2 (2012): 10-1.

[6] Associated Press. "Target Data Breach Cost for Banks Tops $200M." *NBC News.* NBC, 18 Feb. 2014. Web. 12 Sept. 2016.

[7] B. (Intel), Alexander. "Introduction to SGX Sealing." Web log post. *Intel® Software.* 4 May 2016. Web. 16 June 2016.

[8] Baumann, Andrew, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with Haven." *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015): 8.

[9] Beekman, Jethro G., John L. Manferdelli, and David Wagner. "Attestation Transparency: Building secure Internet services for legacy clients." *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security.* ACM, 2016.

[10] Bell, D. Elliott, and Leonard J. La Padula. *Secure computer system: Unified exposition and multics interpretation.* No. MTR-2997-REV-1. MITRE CORP BEDFORD MA, 1976.

[11] Bhargav-Spantzel, Abhilasha. "TRUSTED EXECUTION ENVIRONMENT FOR PRIVACY PRESERVING BIOMETRIC AUTHENTICATION." *Intel® Technology Journal* 18.4 (2014).

[12] Brickell, Ernie, Jan Camenisch and Liqun Chen. "Direct Anonymous Attestation." *Proceedings of the 11th ACM Conference on Computing and Communications Security - CCM '04* 2004: 132–145. Web. 9 Oct. 2016.

[13] Bright, Perter. "Independent Iranian Hacker Claims Responsibility for Comodo Hack". *WIRED.* 28 Mar. 2011. Web. 7 Oct. 2016.

[14] Costan, Victor, and Srinivas Devadas. *Intel sgx explained.* Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086

[15] Champagne, David, and Ruby B. Lee. "Scalable architectural support for trusted software." *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture.* IEEE, 2010.

[16] Durumeric, Zakir, et al. "The matter of heartbleed." *Proceedings of the 2014 Conference on Internet Measurement Conference.* ACM, 2014.

[17] *FIPS 140-2 Consolidated Validation Certificate No. 0003.* 2011.

[18] Gardner, Brett S. "Responsive Web Design: Enriching the User Experience." *Sigma Journal: Inside the Digital Ecosystem.* Vol. 11 Num. 1 Noblis, Inc. Oct. 2011. Web. 13 Oct. 2016.

[19] Gu, Guofei, et al. "Bothunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation." *Proceedings of the 16th USENIX Security Symposium.* 2007. 167-182. Web. 11 Oct. 2016.

[20] Gu, Liang, et al. "Remote attestation on program execution." *Proceedings of the 3rd ACM workshop on Scalable trusted computing* ACM, 2008.

[21] Gueron, Shay. "A Memory Encryption Engine Suitable for General Purpose Processors." 2016.

[22] "Intel® Software Guard Extensions: Intel® Attestation Server API." *Intel Whitepaper.* Intel Corporation. N.d.

[23] "Enclave Access Control and Data Structures." *Intel 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide* Vol. 3D. Intel Corporation. 2016

[24] Hoekstra, Matthew, et al. "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA* 2013.

[25] "Intel® Software Guard Extensions." *Presentation.* ISCA. 2015.

[26] *Intel® Software Guard Extensions SDK Developer Reference for Linux* OS.* 1.5 ed. Intel Corporation, 2016.

[27] Jain, Prerit, et al. "OpenSGX: An Open Platform for SGX Research." *Presentation.*

[28] Johnson, Simon, et al. "Intel® Software Guard Extensions: EPID Provision and Attestation Services." *Intel Whitepaper.* Intel Corporation. 2016

[29] *Intel® Software Guard Extensions Programming Reference.* Intel Corporation, 2014.

[30] Kudelski Security. *sgxfun.* Nagravision S.A. 2016. github.com/kudelskisecurity/sgxfun/

[31] Kim, Seongmin, et al. "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications." *Proceedings of the 14th ACM Workshop on Hot Topics in Networks.* ACM, 2015.

[32] Lal, Reshma, and Pradeep M. Pappachan. "An architecture methodology for secure video conferencing." *Technologies for Homeland Security (HST), 2013 IEEE Conference on,* IEEE, 2013.

[33] McAfee. *Net losses: Estimating the Global Cost of Cybercrime.* 2014. PDF.

[34] McKeen, Franck, et al. "Innovative instructions and software model for isolated execution." *HASP@ ISCA.* 2013.

[35] Murdoch, Steven J. "Introduction to Trusted Execution Environments (TEE)-IY5606."

[36] Nyman, Thomas, Brian McGillion, and N. Asokan. "On Making Emerging Trusted Execution Environments Accessible to Developers." *International Conference on Trust and Trustworthy Computing.* Springer International Publishing, 2015.

[37] Prins, J. R. "DigiNotar Certificate Authority breach 'Operation Black Tulip'". *Fox-IT*, 5 Sept. 2011. Web. 3 Oct. 2016.

[38] Rescorla, Eric. "Http over tls." (2000).

[39] Rozas, Carlos. "Intel® Software Guard Extensions (Intel® SGX)." (2013).

[40] Saltzer, Jerome H. "Protection and the control of information sharing in Multics." *Communications of the ACM* 17.7 (1974): 388-402.

[41] Saltzer, Jerome H., and Michael D. Schroeder. "The protection of information in computer systems." *Proceedings of the IEEE* 63.9 (1975): 1278-1308.

[42] Sandblom, Derek, William Vine, and Benjamin Vowell. *Secure Enclaves-Enabled Technologies.* AIR FORCE ACADEMY COLORADO SPRINGS CO, 2014.

[43] Schellekens, Dries, Brecht Wyseur, and Bart Preneel "Remote attestation on legacy operating systems with trusted platform modules." *Science of Computer Programming* 74.1 (2008): 13-22.

[44] Sherstobitoff, Ryan. "Dissecting Operation Troy: Cyberespionage in South Korea." *McAfee*, 2013. http://www.mcafee.com/us/resources/white-papers/wp-dissecting-operation-troy.pdf

[45] Schroeder, Michael D. "Engineering a security kernel for Multics." *ACM SIGOPS Operating Systems Review.* Vol. 9 No. 5. ACM, 1975.

[46] Schuster, Felix, et al. "VC3: trustworthy data analytics in the cloud using SGX." *2015 IEEE Symposium on Security and Privacy* IEEE, 2015.

[47] Selvaraj, Surenthar. "Overview of Intel Software Guard Extensions Instructions and Data Structures." Web log post. *Intel® Software.* 10 June 2016. Web. 16 June 2016.

[48] "SGX Instruction References." *Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide* Vol. 3D. Intel Corporation. 2016

[49] Shapiro, Robert, and Kevin Hassett. "The Economic Value of Intellectual Property." *Sonecon.* http://www.sonecon.com/docs/studies/IntellectualPropertyReport-October2005.pdf

[50] Sinha, Rohit, et al. "Moat: Verifying confidentiality of enclave programs." *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* ACM, 2015.

[51] Smith, Sean W., and Steve Weingart. "Building a high-performance, programmable secure coprocessor." *Computer Networks* 31.8 (1999): 831-860.

[52] "Stolen Intellectual Property Harms American Businesses Says Acting Deputy Secretary Blank." *The Commerce Blog, U.S. Department of Commerce*, 29 Nov. 2011. Web. 12 Sept. 2016. http://www.commerce.gov/blog/2011/11/29/stolen-intellectual-property-harms-american-businesses-says-acting-deputy-secretary-

[53] Strackx, Raoul, Pieter Philippaerts, and Frederic Vogels. "Idea: Towards an Inverted Cloud." *International Symposium on Engineering Secure Software and Systems.* Springer International Publishing, 2015.

[54] Torrey, Jacob I., and Brent M. Sherman. "Enclaves for operating system protection." *2016 Annual IEEE Systems Conference (SysCon).* IEEE, 2016.

[55] Trusted Computing Group. TPM main specification. 1 Oct. 2014. Web. 7 Oct. 2016. http://www.trustedcomputinggroup.org/tpm-library-specification/

[56] University of Pennsylvania Wharton School of Business. "Latin America Reaches a Crossroads for Guarding against Cybercrime." *Wharton School of Business*, 24 Jul. 2013. Web. 12 Sept. 2016.

[57] Winter, Johannes. "Trusted computing building blocks for embedded linux-based ARM trustzone platforms." *Proceedings of the 3rd ACM workshop on Scalable trusted computing.* ACM, 2008.

[58] "6th Generation Intel® Core™ i7 & i5 Desktop and Intel® Xeon® E3-1200 v5 Family Processors" *Intel® Product Change Notification.* PCN 114074-00. 2015.

Acknowledgements

We would like to thank Joshua Guttman, John D. Ramsdell, John Butterworth, and the wonderful people of department J83K and the MITRE Corporation for their guidance, advice, and assistance. We would like to thank Craig Shue for his guidance and support. Worcester Polytechnic Institute for this opportunity. The drivers at AA Transportation, for getting us safely to MITRE every day.