A Practical TLS Enabled Web Server Without Trusting The Operating System

A. Awad C. Chaiphet I. Gakos W. Grajkowski
B. Jacob

August 23, 2016

Abstract

Web Servers are now often deployed in the cloud environment. To secure communications in the middle of the network between clients and web servers, the Internet community has come to rely on the SSL/TLS protocol which requires a strong secret (private key) to be stored with the server. However, the server administrator in this model is forced to trust the cloud provider to not disclose their private key. We propose a modification to legacy web servers and cryptographic libraries which secures the sensitive key material in the face of a malicious provider, or a compromised operating system, without the need to trust the cloud provider and/or operating system.

Contents

1 Introduction

With the recent surge in privacy concerns, employing SSL/TLS to secure communications in the middle of the network has become common place. SSL/TLS offers guarantees of confidentiality and integrity, provided that a private key's secrecy is maintained. Yet, SSL/TLS was designed assuming that its user trusts the hardware and OS of the machine on which the key is held.

While this assumption is perfectly valid in the case where a person is running an SSL/TLS-enabled service on their own machines, many web applications are now hosted by third party cloud service providers such as Amazon Web Services, Heroku, Digital Ocean &c. Moreover, to offer SSL/TLS, the private key must also be stored with the web application on these service providers' machines. This implies that a server administrator using the aforementioned services is trusting the cloud-provider, including any personnel with physical/administrative access to the machines, and the underlying OS to maintain the secrecy of the sensitive key material. Such a wide trust surface makes it difficult to maintain the privacy of critical secrets.

Consider a case where the cloud provider is not malicious; a vulnerability within their platform could lead to leaking the private key if exploited by an adversary. Moreover, if the cloud provider is indeed malicious, they could simply read your private key from the hard disk, assuming the key is not encrypted, or mount some form of memory sniffing attack to read the key from the web server's memory since data in RAM is not encrypted. A compromised private key allows an adversary to do the following:

- Decrypt past, stored communication between the web server and a client (assuming a cipher that does not provide perfect forward secrecy is in use)
- Decrypt any ongoing communication between the web server and a client
- Masquerade as the server and fool a client into disclosing sensitive information such as passwords

In all cases, a compromised key voids the confidentially and integrity guarantees of SSL/TLS.

Clearly, implicitly trusting a cloud-proivder and the OS with sensitive key material poses a considerable security risk. Yet, this assumption of trust is usually the case when server administrators deploy web applications via cloud based services. Our project aims to break this assumption by refactoring legacy web servers to secure the long term private key in the face of: (1) An adversary who is capable of exploiting the server application, (2) a malicious cloud provider, and (3) an adversary with an exploit for the underlying operating system. We utilise the principle of least privilege and along with a recently released technology from Intel® called Software Guard Extensions, both of which are described in Section 2, in designing our system.

The remainder of this report is divided as follows: Section 2 provides a detailed overview of previous solutions to the problem outlined here, and technologies underlying our project, Section 3 discusses the design of the system that we implemented to meet the above-stated goals, Section 4 highlights a few implementation considerations in realising the system that we designed, Section 5 presents the evaluation we have conducted of the system, Section 6 details the managerial aspects of this project, and, finally, Section 7 offers areas where this project could be improved, and concludes this report.

2 Background

2.1 An Overview of SSL/TLS

Before discussing previously proposed solutions to the problem we identified in the Introdcution, we present here a generic overview of SSL/TLS. In general, regardless of the cipher used, SSL/TLS utilises the long-term private key to negotiate ephemeral keys for use in the current session; hence, to secure the long-term private key, we need to examine the session establishment mechanism. Roughly, SSL/TLS's handshake can be split into four steps:

1. **Contacting the server, and establishing parameters for the session**. Specifically, the exchange of hello messages that include: the server's certificate(s) a list of ciphers supported by each side (to determine which cipher is to be used for this session), `ServerRandom`, and `ClientRandom`. The random variables are used to derive the secret, used to secure communications.

2. **Asymmetric key exchange**: based on the cipher selected, the client and server determine the asymmetric keys that are to be used to exchange the symmetric keys for the session. This step is necessary in two scenarios:

   (a) The client and server both possess public key certificates, and both entities wish to verify each other's identity during the handshake. We do not consider this case due to its rarity. Generally, the client verifies the server as part of the SSL/TLS handshake, and the server verifies the identity of the client via some other means, such as a username & password.

   (b) The client and server selected a cipher that offers forward secrecy. These ciphers function by first exchanging an ephemeral asymmetric secret. This secret is then used in negotiating the symmetric secret.

   In all other cases, the server's asymmetric keys are used to negotiate the ephemeral symmetric secret.

3. **Ephemeral symmetric key negotiation**: The client and server establish the symmetric secret that is used for the current session. The symmetric ephemeral keys are calculated as outlined below:

   (a) The `ClientRandom`, `ServerRandom`, and a value denoted `PremasterSecret` are combined together, through use of a pseudo-random function (PRF), to generate a value called the `MasterSecret`. The `MasterSecret` is a 48-byte number, and is computed using the method outlined here in all SSL/TLS ciphers. In contrast, the `PremasterSecret` is a random value, established as part of this step. Arriving to the value of the `PremasterSecret`, however, depends on the cipher used.

   (b) The `MasterSecret` is then used to generate a key block. A session key block consists of:

      - `server_write_key`: this key is used by the server to encrypt outgoing packets, and by the client to decrypt incoming packets.

- `client_write_key`: this key is used by the client to encrypt outgoing packets, and by the server to decrypt incoming packets.
- `server_mac_secret`: this key is used by the server to compute a MAC over outgoing packets, and by the client to verify the MAC over incoming packets.
- `client_mac_secret`: this key is used by the client to compute a MAC over outgoing packets, and by the server to verify the MAC over incoming packets.
- `client_initialisation_vector(iv)`: this is not a key, but a value used to initialise the symmetric cipher at the client, before invoking the encryption routine on outgoing packets, and at the server, before invoking the decryption routine on incoming packets.
- `server_iv`: same as above, but used by the server before invoking the encryption routine on outgoing packets, and by the client before invoking the decryption routine on incoming packets.

4. **Verifying the integrity of the just-negotiated keys, completing the session establishment**: Both the server and the client compute a MAC across the packets exchanged in establishing the session. The resultant finished message is encrypted using the just-negotiated keys. If both sides successfully verify the MAC, the handshake is complete and the session is established. If either side fails to verify the MAC, the session is terminated.

Figure 1 illustrates this generalized SSL/TLS handshake.

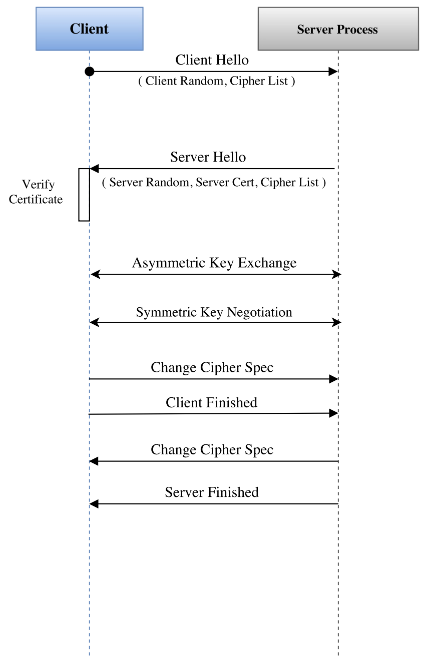

Figure 1: SSL/TLS handshake generalization

2.2 Previously Proposed Solutions

2.2.1 The Principle of Least Privilege

The first of our goals has been extensively addressed in previous works [1, 2] through use of the principle of least privilege. The principle of least privilege (PoLP) requires that components of a system operate with the minimum resources required to complete their respective tasks. PoLP is heavily utilized in systems security research to design systems that maintain their integrity in the face of an adversary capable of exploiting some of their components. In contrast, the most popular web servers today execute as monolithic applications, including Apache and NGINX, where all of their processes have the same level of privilege and have access to the sensitive key material. Exploiting any one of these processes may therefore lead to leaking the private key.

To motivate the design of our system, we begin by discussing a system called Wedge [1] that was used to refactor Apache, by enforcing PoLP, securing the private key in the face of an adversary who is capable of exploiting the web server application. First, we list the attacks possible against a monolithic web server, and then detail how the design implemented with Wedge resolved these vulnerabilities.

We only consider a threat model where the adversary is capable of passively eavesdropping on secure communication channels, and exploiting unprivileged components of the server. We did not consider the second threat model detailed in the Wedge paper, wherein an attacker is capable of actively modifying packets exchanged between the web server and client, because the solution presented in there does not hold if we remove the assumption that the OS and cloud provider are trusted. A consequence of the web server's user-level processes (dynamic content generation scripts, databases &c.) requiring data in plain-text to complete their tasks, even if we secure the encrypt/decrypt interface using the solution in the Wedge paper, is that plain-text data would be available in the non-privileged process's memory. This is further discussed in Section 3.

Possible Attacks

There are two main attacks that may be mounted by an adversary capable of exploiting *only* the network facing component of the web server application and passively eavesdropping on packets exchanged between the server and the client:

1. The adversary could leak the private key from the network facing component, which has to run as root to bind to port 80. This could be done by reading the private key from disk directly or from the process's memory space. The attack is illustrated in Figure 2.

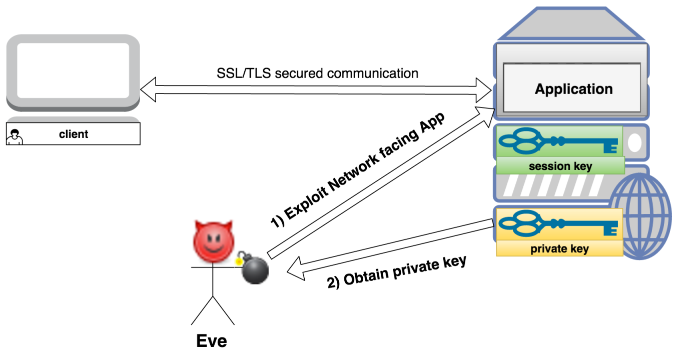

Figure 2: Exploiting the network facing component to leak the private key

2. The adversary may record traffic exchanged over the SSL/TLS channel and then exploit a naive session key generation interface to acquire the session keys used in that exchange. A naive interface is one that accepts `ClientRandom`, `ServerRandom`, and `{PremasterSecret}`$_K$ where K is the web server's public key. Such an interface allows an adversary, after exploiting the unprivileged component, to generate any previously eavesdropped session's symmetric key (assuming a cipher that does not provide the property of perfect forward secrecy) and is, therefore, no different to having read access to the private key from the adversary's perspective. This exploit only works if the cipher used is one that does not offer perfect forward secrecy (PFS), and it only affects the single session which was eavesdropped. The attack is illustrated in Figure 3.

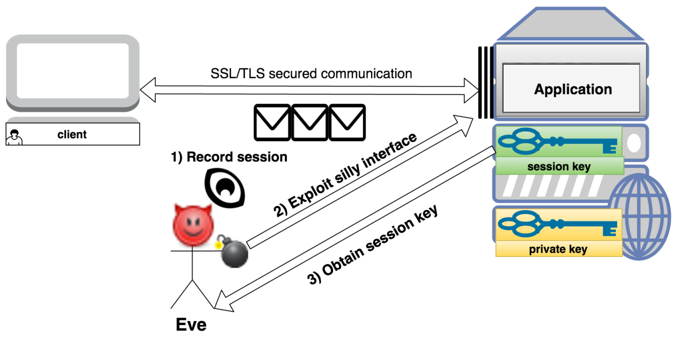

Figure 3: Exploiting the network facing component and the naive session key generation interface to generate session keys for eavesdropped session

Proposed Solution

The solution proposed in the Wedge paper is achieved through partitioning the session key generation code into its own logical compartment [1] that executes at high privilege. This partitioning can be seen in Figure 4.

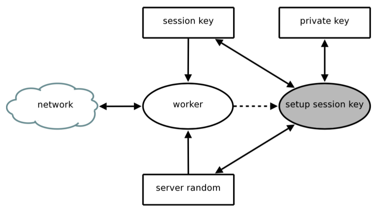

Figure 4: Partitioning Scheme to protect against leaking the private key [1]

The low privilege worker process does not possess the private key, but must rely on an interface to the session key generation process. The interface takes `ClientRandom` and $\{\texttt{PremasterSecret}\}_K$ as arguments, both of which are provided by the client, while the high privilege component generates a fresh `ServerRandom` upon invoking the interface. The freshness property ensures that, even if an adversary exploits the worker process, previous session keys cannot be generated by simply invoking the session key generation interface.

While this design secures the private key against the previously described adversary, it is not, by itself, sufficient to secure the key against an adversary capable of exploiting the OS, or from a malicious cloud-provider with full access to the physical machine. Such adversaries may leak the key from the privileged component's memory space by exploiting the OS, or, in the case of the malicious cloud provider, simply read the key from disk[2]. A hardware component, called a trusted platform module, has been utilised

[1] This compartment is called an sthread in Wedge's nomenclature

[2] Even if the key is encrypted on disk, the decryption key must be present in the plain-text binary file for the privileged component, or in the process's memory, at runtime, in plain-text

in an effort to secure the private key against the adversaries described above.

2.2.2 Trusted Platform Modules

Securing the private key against an adversary capable of exploiting the operating system is achievable through use of specialised hardware called a trusted platform module (TPM). TPMs can secure the private key through encryption via a Storage Root Key (SRK) [3]. The SRK's integrity is maintained by ensuring that its private component may never leave the TPM. As a result, the long term private key itself can never be decrypted outside the TPM. The TPM is also capable of executing cryptographic operations, including those that make use of the long term private key in SSL/TLS. Consequentially, by delegating all private key operations to the TPM, one can rest assured that their private key cannot be compromised without compromising the TPM itself.

The security benefits of a TPM, however, were outweighed by the cost of purchasing the additional piece of hardware, and TPMs did not gain any traction with cloud providers. In this project we utilized a new technology from Intel® called Software Guard Extensions (SGX). SGX is an augmentation to Intel®'s ISA which offer the ability to launch encrypted regions of memory, called enclaves, where only trusted regions of code can read/write. This allows for TPM-like functionality, but SGX has the advantage of being deployed as part of new, commodity, CPUs released by Intel®. As a result, when cloud providers upgrade their machines, they will possess the ability to support SGX programs without purchasing additional hardware.

SGX has gained momentum as a research platform for security related work such as Haven [4], which secures a legacy application from a non-trusted OS and cloud-provider *without* modifying the application's source code. Yet, there is no work, to our knowledge, that attempts to secure only the private key material through use of SGX. Narrowing the trusted region to contain only the component that handles the private key allows us to define a much smaller trusted computing base that only contains the CPU and the code that handles the long term private key.

2.3 An Overview of SGX

This section will not cover all of the details of SGX, but only those applicable to our project; for a complete treatment of SGX please refer to [5]. Intel® SGX is a set of x86 instructions that allow for a programming model wherein a program can be split into two components: an untrusted component that executes as normal and a trusted component that executes within a protected area of RAM, called an enclave, which can only be accessed when executing the trusted component.

The protection of an enclave is managed by the CPU; any data written to the enclave is encrypted first by a memory encryption engine (implemented in hardware) and is only decrypted when required by the CPU during the execution of the trusted component, which we refer to as the *enclave program* for which that enclave belongs. The key used for this encryption process is derived from a combination of a device key, unique to each SGX-enabled CPU and the "identity" of the enclave called MRENCLAVE, a cryptographic hash of the enclave's contents at the trusted component's initialization. SGX thus ensures that no process other than the one that initialized the enclave can access the protected area.

Interacting with the trusted component, as a result, may only occur through invoking a programmer defined interface, called a callgate, as depicted in Figure 5.

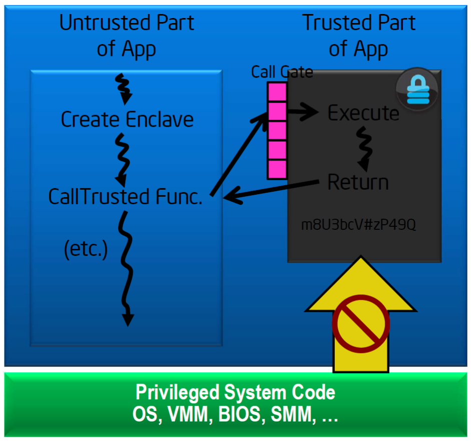

Figure 5: Interaction of the untrusted part of the application with the trusted part can only occur through a callgate

2.3.1 Initialising an SGX Enclave

Launching an SGX *enclave program* requires the execution of the following steps:

1. The *enclave program* is first compiled and then signed with an Intel® verified certificate[3]. The signing process generates a data structure called `SIGSTRUCT` that contains the aforementioned MRENCLAVE, the identity of the signing entity, MRSIGNER, and the signature across this structure.

2. When attempting to launch the *enclave program*, the CPU calculates MRENCLAVE using priveleged SGX instructions, calculates MRSIGNER by taking the hash of the public key contained in the signing entity's certificate, and then attempts to verify that MRSIGNER_{calc} is equal to $\text{MRSIGNER}_{SIGSTRUCT}$, MRENCLAVE_{calc} is equal to $\text{MRENCLAVE}_{SIGSTRUCT}$, and that the signature across `SIGSTRUCT` was generated by the signing entity indicated by MRSIGNER.

3. If all previous checks complete successfully, then the CPU proceeds to execute the *enclave program* otherwise, the CPU aborts execution.

At the time we begun working on this project we did not posses access to SGX hardware, forcing us to use a simulator. There were two choices, at the time: OpenSGX and the Intel® Windows SDK's simulation mode. We selected the former to implement our prototype as it was compatible with Linux, a platform we were more familiar with,

[3] While the requirements imposed on the certificate for successful verification are well documented, the process is not automated, and involves an application to Intel®'s Infrastruction Attestation Service (IAS)

and had several examples we could refer to for guidance in our implementation (the Intel SDK, at the time, was under-documented). The following section provides a quick examination of OpenSGX's salient features.

2.3.2 Brief Notes on OpenSGX

OpenSGX [6] is a platform that provides a software emulation layer enabling the execution of SGX instructions *without* the possession of SGX hardware. Specifically, OpenSGX implements a hardware emulation layer, implementing SGX's instructions and data structure as per Intel®'s specification [4], and an OS emulation layer, providing access to wrapper functions for privileged SGX instructions (instructions, such as EADD and EINIT, used to bootstrap an enclave fall under this category)[5].

Programs that run on top of OpenSGX consist of the trusted code and a wrapper program which initializes the enclave and provides the trusted code with an interface to `sgx-lib`, a library that contains wrapper- functions for user-level SGX instructions.

Intel®'s SGX specification does not detail how an enclave program may invoke system calls[6]; OpenSGX specifies an interface which allows an enclave program to invoke `libc` functions. The interface works as follows:

1. The enclave program writes the arguments for the system call into a shared area of memory called a *stub*
2. The enclave program invokes EEXIT, exiting enclave mode, and executes a predefined handler called a *trampoline* which invokes the system call
3. After the system call is complete, results are written to into the *stub* and ERESUME is called, resuming the execution of the enclave program

By creating this interface, writing code for OpenSGX is akin to writing C code, making the development process familiar. Figure 6 illustrates the high level design of OpenSGX.

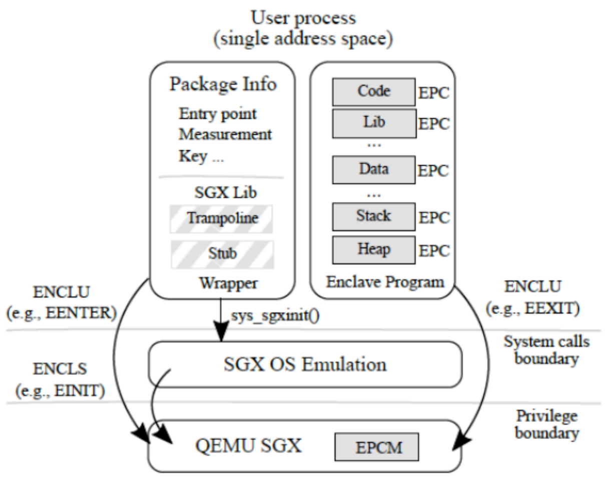

Figure 6: High level design OpenSGX. Grey boxes indicate protected areas of memory. Striped boxes indicate shared areas.

OpenSGX also provides a tool-chain for automating tasks such as compiling and signing binaries, generating signing keys, and verifying and executing correctly signed binaries. However, it is imperative to note that OpenSGX does not provide the same security guarantees as SGX; it does not encrypt the enclave memory region, for example. Yet, it has proven to be useful as a platform for testing the design of our system, and measuring its performance.

2.4 Provisioning The Enclave With The Long Term Private Key

SGX by itself, however, does not answer an imperative question: How do you get the private key into the enclave in the first place? While the key would be encrypted as it is used by an *enclave program* in RAM, it cannot be simply shipped as part of the binary. This is because the binary is not encrypted. Yet, provisioning a web server with a long-term private key for the purposes of SSL/TLS is currently done by storing the private-key along with the executable on the remote machine. This scheme, however, assumes a trusted cloud-provider/OS. Therefore, to meet our above-stated goals, this scheme is not viable.

Alternatively, we require a method by which we can verify the identity and integrity of the server application, and then, upon successful verification, we can send the long term private key to the server in a secure fashion. Roughly, the requirements for this process are as follows:

- The mechanism allows the verification of the identity and integrity of the server application and the underlying TCB. This is so that we can be sure the private-key is being sent to the same server we placed on the remote machine, and the software is being executed by trustworthy hardware.

- The mechanism allows us to setup a secure channel, ensuring that the only entities privy to the private key are the server application and the server administrator.

- The mechanism allows for the verification of the entity providing the private key. If this requirement were not in place, the mechanism could allow any arbitrary entity to authenticate with the server and provide their own private key. Observe that such an attack does not compromise the security of the long-term secret, but makes it possible to render the server useless (if the matching public certificate is not placed onto the server, verifying the server's hostname, then clients will reject connections to the server under the SSL/TLS protocol).

[4] A few SGX instructions are not implemented by OpenSGX including: Debugger Read(EDBGRD) and Debugger Write (EDBGWR) which are used to debug an SGX program. However, OpenSGX includes a GDB plug-in that performs the same task. Furthermore, some instructions are not implemented as they were not deemed necessary such as EREMOVE. EREMOVE is called to reclaim memory after an SGX program terminates, but it is not necessary as only one enclave can run at a time on top of a single OpenSGX instance.

[5] The OS emulation layer's implementation was not guided by Intel®'s SGX specification, a consequence of Intel®not specifying the interface for invoking privileged SGX instructions

[6] Simply put, the enclave program cannot invoke I/O system calls directly as any code executed by the enclave program must be part of the signed binary. System calls for I/O operations, however, delegate work to driver software which changes from machine to machine based on the hardware provider. Consequentially, allowing arbitrary system calls from within the enclave creates a large attack surface.

The first and second requirement are met by a process called inter-platform attestation, outlined by Intel® in [5] and summarized in the following section[7].

2.4.1 Inter-platform attestation and secret provisioning

Inter-platform attestation is a mechanism that can be invoked by an entity, referred to as the challenger, running on one platform to verify an enclave running on another, remote, platform. This process enables the challenger to verify the following about the remote enclave:

1. The contents of the enclave's pages (code, data, stack and heap) upon creation (after the ECREATE instruction completes)
2. The identity of the entity that signed the enclave
3. The trustworthiness of the underlying hardware
4. Authenticity and integrity of any data generated by the enclave and sent as part of the attestation process. This allows us to satisfy the second requirement by generating an ephemeral key pair and binding it to the remote attestation process. This, therefore, allows the challenger to verify the integrity of the ephemeral public key and verify that it was generated by the server application.

The steps involved in the attestation process are as follows (illustrated in Figure 7):

1. The challenger invokes the remote attestation mechanism to verify the identity and integrity of the remote enclave
2. The non-trusted part of the web server receives the challenge, passes it along to the trusted portion of the web server along with the identity of the quoting enclave. The quoting enclave is a special enclave provided by Intel as part of the SGX platform to enable remote attestation by verifying the integrity of the underlying hardware.
3. The enclave invokes EREPORT which is an SGX instruction that generates a REPORT structure to be provided to a *local* enclave, the quoting enclave in this case. This structure contains a hash of the contents of the enclave's pages upon ECREATE's termination (MRENCLAVE), a hash of the identity of the enclave's signer, a hash of any user-data, the ephemeral key in our case, generated by the enclave. The REPORT is signed by a MAC-key that can only be accessed by the CPU and the quoting enclave. The REPORT along with the ephemeral key is then sent to the non-trusted part of the application.
4. The REPORT is sent to the quoting enclave where its integrity is verified by calculating the MAC across its contents.

[7] Intel® did not devise this mechanism, and it has been used before in other Trusted Execution Environments (TEE) to provision the TEE with highly sensitive secrets

5. Assuming the REPORT is verified successfully, the quoting enclave generates a QUOTE structure that includes the REPORT structure and a signature across the quote generated using a key known as the EPID key. to the CPU that is part of the platform, and verifies the firmware of the processor and its SGX capabilities.

6. The QUOTE is sent along with the ephemeral key to the challenger

7. The challenger verifies the QUOTE structure by using an EPID public certificate. If this is successful then the challenger is sure that this QUOTE came from a valid SGX CPU and can trust its authenticity. The challenger can then check the contents of the REPORT contained within the QUOTE to verify the identity of the remote enclave, and the integrity of the ephemeral key received along with the QUOTE. The ephemeral key, if proven to be valid, can now be used to communicate with the remote enclave in a secure manner.[8]

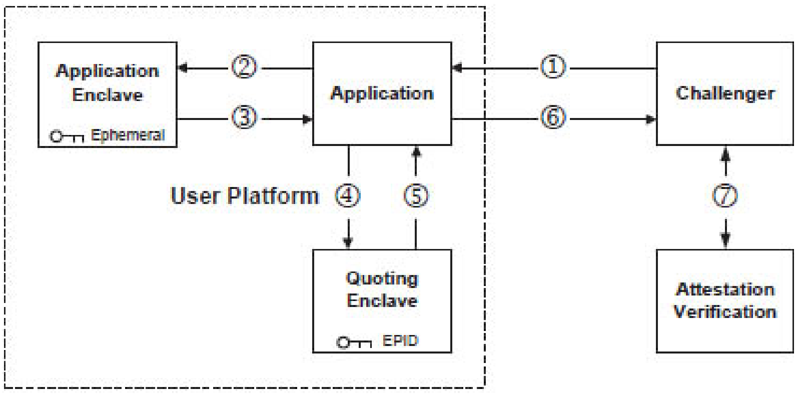

Figure 7: Remote Attestation and Secret Provisioning

Intel®has also added a construct that allows an *enclave program* to generate keys that can be used to encrypt sensitive data and persisit it on disk. The only entities privy to the key are the *enclave program* and the CPU, therefore, only these two entities can access the sealed data[9]. This process, called sealing, ensures that we execute the attestation process once, the first time the *enclave program* is launched, alleviating its cost.

However, the process outlined above takes no steps to verify the challenging entity to the trusted component. In an effort to authenticate the challenger, we utilize a combination of asymmetric cryptography and SGX's guarantees. First, the server-administrator, before shipping the server program to the cloud-provider, stores the public key for the challenger within the text-area of the trusted component. The reason for this comes

[8] Verifying the certificate itself is conducted through contacting Intel®'s attestation verification service

[9] This is not entirely true, Intel®defined the EGETKEY instruction, the one that generates the sealing keys such that it takes an argument of either MRENCLAVE or MRSIGNER. If MRENCLAVE is used, then the sealing key generated is unique to the invoking *enclave program*. If MRSIGNER is used, however, the sealing key generated is unique to all *enclave programs* signed by that entity

from realizing that doing so binds the public key to the identity of the trusted component, and as a result, the SGX-enabled CPU would not start the trusted-component if the public key has been tampered with. All that remains now is for challenger to sign the long term private key with their own key, enabling the trusted-component to verify it using the challenger's public-key that was shipped along with the server-program.

3 Design

3.1 The Threat Model

Before presenting our design, we introduce the threat model we utilised to scope the discussion. We consider a threat model similar to the one presented in Haven [4]. Our TCB includes a correctly implemented and SGX enabled CPU and all instructions executing and data resident within an enclave. Therefore, we assume an attacker cannot access the SGX processor key provisioned by Intel itself, which is used to generate subsequent cryptographic keys that preserve the confidentiality, integrity and authenticity of an enclave.

An adversary can take full control of everything beyond the TCB. That is, we assume all software executing on the platform (outside of the enclave), the underlying operating system, the hypervisor, all firmware and the BIOS are potentially compromised. Side-channel attacks that originate from other sources such as CPU cache timing information (L1, L2, L3), power consumption or other entropy source are considered as out of scope of this work. Finally, we assume an attacker may act as man-in-the-middle to eavesdrop active sessions and, as further analysed in the Limitations section, launch a denial-of-service attack, though without compromising any secret isolation guarantees of our design.

3.2 System Design

SSL/TLS supports a wide array of ciphers, but they can be roughly split into two categories: ciphers that support forward secrecy, and ciphers that do not. Ciphers that offer forward secrecy are ones wherein a compromised long-term private key does not allow an adversary to compromise previously eavesdropped, stored sessions. SSL/TLS's session establishment mechanism is different based on the category to which the cipher belongs. Consequentially, depending on the type of cipher used for a given session, our system must adapt to two, different,control flows. In this section we present the following: a high level design that is cipher agnostic, a design that supports ciphers that **do not** support forward secrecy, and a design specific to ciphers that **do** support forward secrecy.

High level design

By utilizing the aforementioned remote-attestation process, we can provision the trusted component of the remote server application with an SSL private key while making it extremely difficult for even *privileged* processes running on the cloud provider to access the key. This is a stark contrast to current schemes wherein the private key is merely shipped as part of the server application to the cloud provider.

As previously highlighted in Section 2.1, the private key is required by a subset of the operations executed during the handshake step. These operations have to be executed within the trusted component, and are invoked by the non-trusted component via an interface that we define in this section. Note that the interface has to be carefully designed, allowing the handshake to complete correctly while not exposing the long term private key through an oracle.

Take for example an interface where the non-trusted component supplies `(ServerRandom, ClientRandom, {PremasterSecret}`$_K$`)`. Such an interface, while maintaining the secrecy of the private key's bits, would allow an adversary capable of exploiting the non-trusted component to generate the symmetric keys for previously eavesdropped sessions. Therefore, it is no better than leaving the private key in the non-trusted component. However, observe that it is not necessary for `ServerRandom` to be provided by the non-trusted component, it need only be provided by the *server*.

We can adjust the interface so that the non-trusted component supplies `ClientRandom` and `{PremasterSecret}`$_k$, both of which are generated by the client, and the trusted component generates a new `ServerRandom` every time the interface is invoked. The resulting interface ensures that, even if a previously eavesdropped `{PremasterSecret}`$_K$ is provided, a fresh session-key is computed on every invocation.

Forward Secrecy and ECDHE Handshake In RSA key exchange process, all the messages exchanged between server and client can be compromised if the server's private key is ever disclosed. In specific, an active attacker who has recorded all the data communication between the server and client for a certain period of time and stores it until such a time he has access to the private key. The adversary can now decrypt all the historic data using the compromised private key. This is possible because the attacker will have access to all the seeds (Client Random, Server random and Pre-Master secret) required to generate the symmetric session keys This capability to decode historic data at any point of time in the future pose a serious threat to the secure communication over the cyberspace.

Forward secrecy is a feature in the key arrangement protocol, which guarantees that the symmetric encryption keys can not be compromised even if the long term key (server private key) is compromised. To achieve forward secrecy we employed elliptic curve diffie-hellman(EC-DHE) key exchange mechanism. Most of the web servers prefer elliptic curves variant over standard DHE as it yields better performance with lesser number of key bits[1]. DHE handshake uses private key only to authenticate the server and generates fresh set of symmetric keys for every new session. The keys are computed from random EC parameters, which is never reused and also deleted once the session has ended. Hence in ECDHE Private keys will not affect the confidentiality of the past conversations, albeit part of the initial handshake. The private key will never aid the attacker as it was only used to authenticate the server identity If in case the attacker managed to attain the shared secret, he can only decrypt messages from that particular session. No previous or future sessionswould be compromised.

We have partitioned the standard EC-DHE handshake implementation in Libressl and separated it into two components. One network facing component is deemed untrusted and is running as a part of nginx web server process. The other part resides inside the trusted SGX enclave and runs as an independent process. Both the components of the server communicate via a list of fine grained API's wrapped over a named pipe IPC interface.

The handshake process is carried out in two different phases, one to authenticate the server identity and other to establish the pre master secret. Figure 8) illustrates the messages exchanges between client and server's trusted and untrusted compartments.

A Https client can initiate the handshake by sending a "client hello" enclosed with a random number and a list of Elliptic curves it supports. Upon receiving this request, the nginx server initialises a new session in the enclave (TODO: Add info related to session cache/resumption.?) and configures the random number received from the client. The server random seed is generated inside the enclave. This will accord to additional security to the system by preventing an adversary from exploiting the untrusted component of the server and by influencing a session key generation based on random seeds of his choice. Enclave code maintains separate state for each session and will not accept server random as an input argument in any API interfaces. After obtaining the server random, it is sent to the client as part of the "Server Hello" message together with Server Public key certificate and a list of server preferred ciphers.

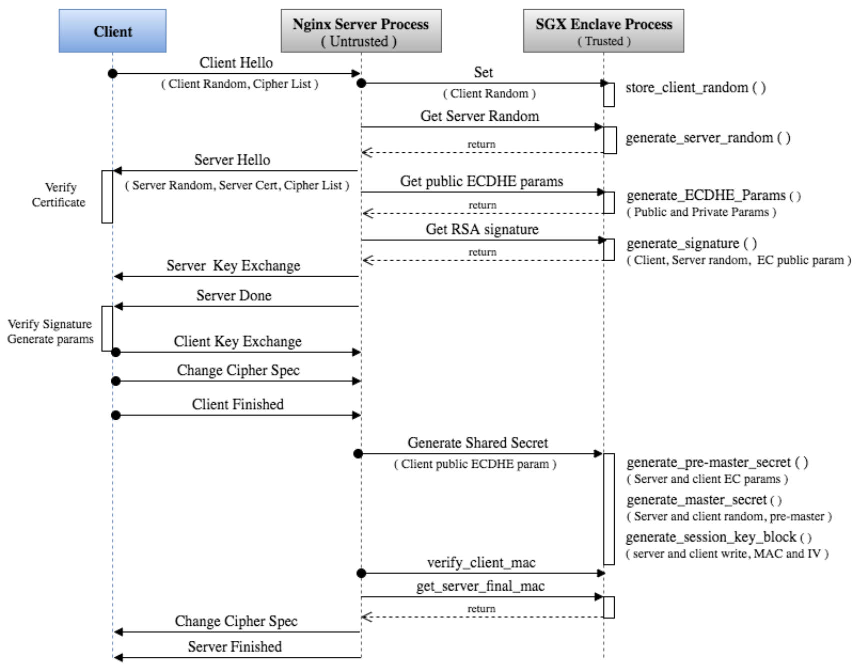
Client
Nginx Server Process
(Untrusted)
SGX Enclave Process
(Trusted)
Client Hello
(Client Random, Cipher List)
Set
(Client Random)
store_client_random ( )
Get Server Random
generate_server_random ( )
return
Server Hello
(Server Random, Server Cert, Cipher List)
Verify Certificate
Get public ECDHE params
generate_ECDHE_Params ( )
(Public and Private Params)
return
Get RSA signature
generate_signature ( )
(Client, Server random, EC public param)
return
Server Key Exchange
Server Done
Verify Signature Generate params
Client Key Exchange
Change Cipher Spec
Client Finished
Generate Shared Secret
(Client public ECDHE param)
generate_pre-master_secret ( )
(Server and client EC params)
generate_master_secret ( )
(Server and client random, pre-master)
generate_session_key_block ( )
(server and client write, MAC and IV)
verify_client_mac
get_server_final_mac
return
Change Cipher Spec
Server Finished

Figure 8: EC-DHE Handshake

In the second phase of DH key exchange, the server will pick one of the client supported elliptic curves and send the curve id to the enclave. In the enclave we will generate both the public and private part of the EC-DH parameters based on the chosen curve id. Only the public part of the DH param is exposed to the nginx server process and the private part is kept secret within the enclave. This ensures additional security to the system by making the enclave completely abstain from accepting any server side seeds pertaining the shared secret generation.

In order to prove, that the server has control of the private key and DH params, the Server has to produce a digital signature. In our design this is computed within the enclave. This will include Client Random, Server Random and Public DH Params signed using server private key. Server sends the ECDHE parameter (in clear) together with Signature to the client as part of the "Server Key Exchange" message. Client verifies this message using the server public key. If the verification is successful , the client generates its part of EC params and sends it to the server as part of "Client Key Exchange" message.

After receiving the "Client Key Exchange" message, the server will attain all the required seeds to launch the shared key generation process. The keys are calculated in a three level process. First a Pre-Master key is computed from Client Public ECDHE param and Server ECDHE params(both Private and Public). The pre-master is then used as one of the seed along with Client Random and Server Random for the shared Master secret generation. Server will finally generate the symmetric session keys with the associated MACs and Initialisation vectors from the master secret. All the three steps of key generation processes is securely performed within the enclave.

The client will then send a "Change Cipher Spec" message to indicate it will use the newly generated keys to hash and encrypt further messages. The handshake is complete from the client end after sending a "Finished" message containing a hash and MAC over all the previous handshake messages. Upon receiving client finished, the server will decrypt it using the session keys and validate the integrity of the handshake process. If the decryption or verification fails, the handshake is deemed to be failed and all the allocated session resources are freed. If successful, the server sends a "Change Cipher Spec" message indicating all messages send from now on will be encrypted using the shared session secret keys. Finally server computes its authenticated and encrypted "Finished" Message and sends it to the Client. This completes the full Handshake process.

We consider two scenarios that slightly change the exchange of messages with the enclave during the handshake:

- Session keys available to the OS
- Session keys hidden inside the enclave and accessible through encrypt/decrypt oracle

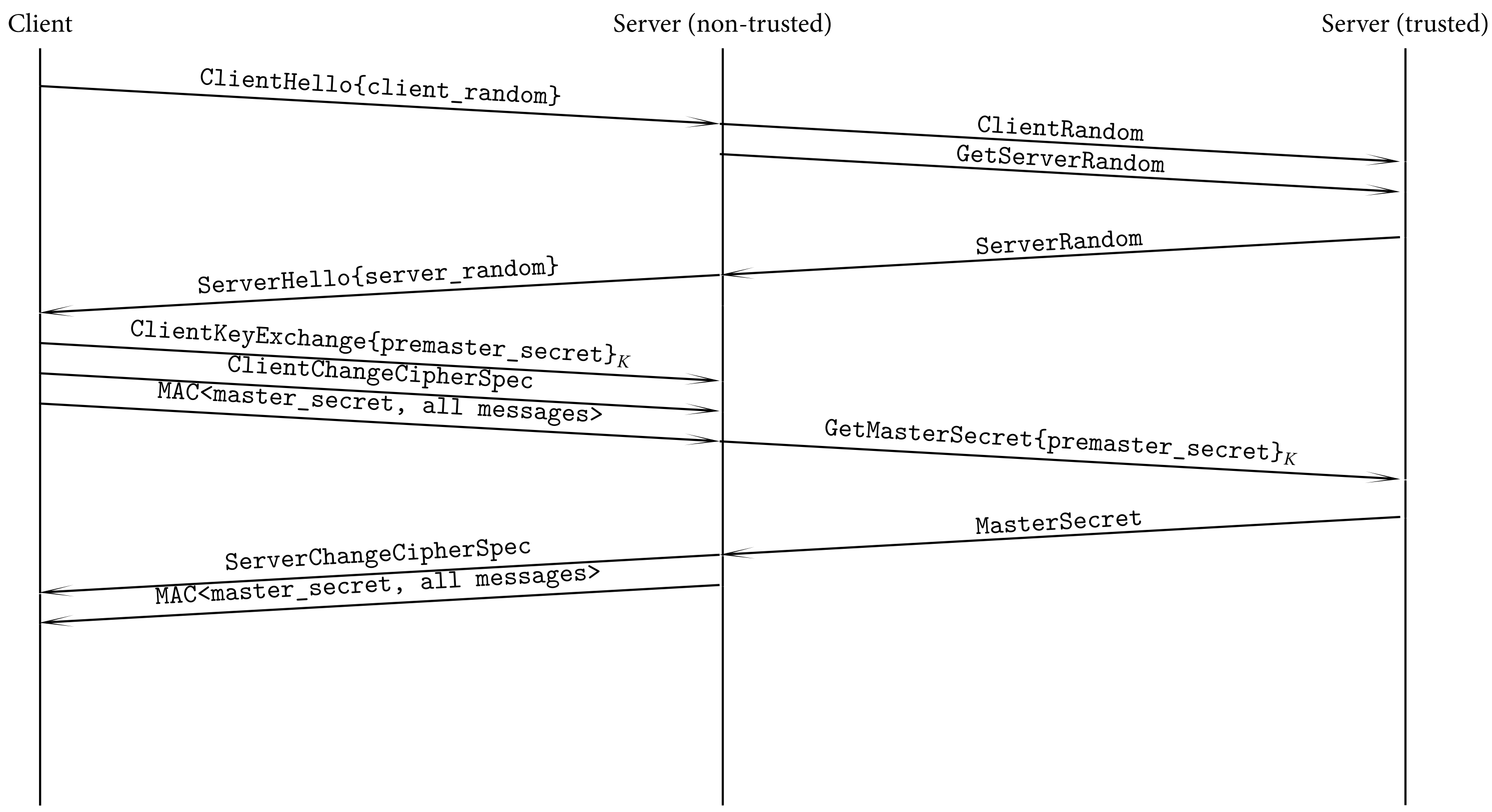
Client
Server (non-trusted)
Server (trusted)
ClientHello{client_random}
ClientRandom
GetServerRandom
ServerRandom
ServerHello{server_random}
ClientKeyExchange{premaster_secret}$_K$
ClientChangeCipherSpec
MAC<master_secret, all messages>
GetMasterSecret{premaster_secret}$_K$
MasterSecret
ServerChangeCipherSpec
MAC<master_secret, all messages>

Figure 9: SSL handshake

Session keys available to the OS If our sole goal is to protect the long term, private key then we may decide it is safe to keep the active sessions' keys outside of the TCB. This allows the compromised OS to use the keys as it pleases. It can just read them out from memory and send to a man in the middle (on a different to compromised machine). MITM could then decrypt previously captured traffic. This method it is possible to leak current (and cached) sessions keys. Old, captured sessions for which session keys have been deleted can no longer be recovered. However, if the OS persists to be malicious it can always monitor future (current) connections.

One could prefer this option for performance reasons - such design puts less burden on the enclave program. In particular the role of the enclave can finish after the master secret is computed from server and client randoms and pre-master secret. This is because a pseudo-random function (PRF) lies between master secret and the input values so no information can be learned about the private key. The derivation of session keys and encryption / decryption of messages can be left unaltered in the untrusted program and therefore less ćontextświtches to the enclave would be required.

Session keys hidden within the enclave One may want to further protect the key material by requiring that the session keys themselves do not leave the enclave. Instead an encrypt / decrypt oracle interface is presented to the untrusted component. The enclave performs the cryptographic operations using stored state accessed by the session_id. This prevents the compromised OS from leaking the session keys to a different machine, but requires a more complicated and expensive design. One can imagine that if the attacker is leaking ton of data out of the server, it may be more easily noticeable than if the session keys are exported outside of the server once.

If we decide to keep our session keys in the enclave apart from providing a mechanism to transfer the cipher context we have to additionally present an interface that would allow to sign the handshake finish messages.

To allow the trusted component to encrypt / decrypt messages we also need to transfer the symmetric cipher context which can be done during computation of master secret. Additionally, every time we need to pass the initialization values such as initialization vector or nonce need to be transferred to the enclave.

The performance cost is increased since now at least 4 context switches between untrusted and trusted processes are required for each packet processing, i.e. 2 switches to decrypt an incoming packet and pass plain-text back to the application and 2 to encrypt the outgoing result after application processing (plain-text in, cipher-text out of an enclave).

Also every handshake now needs 6 (TODO: ???) more context switches during digest update.

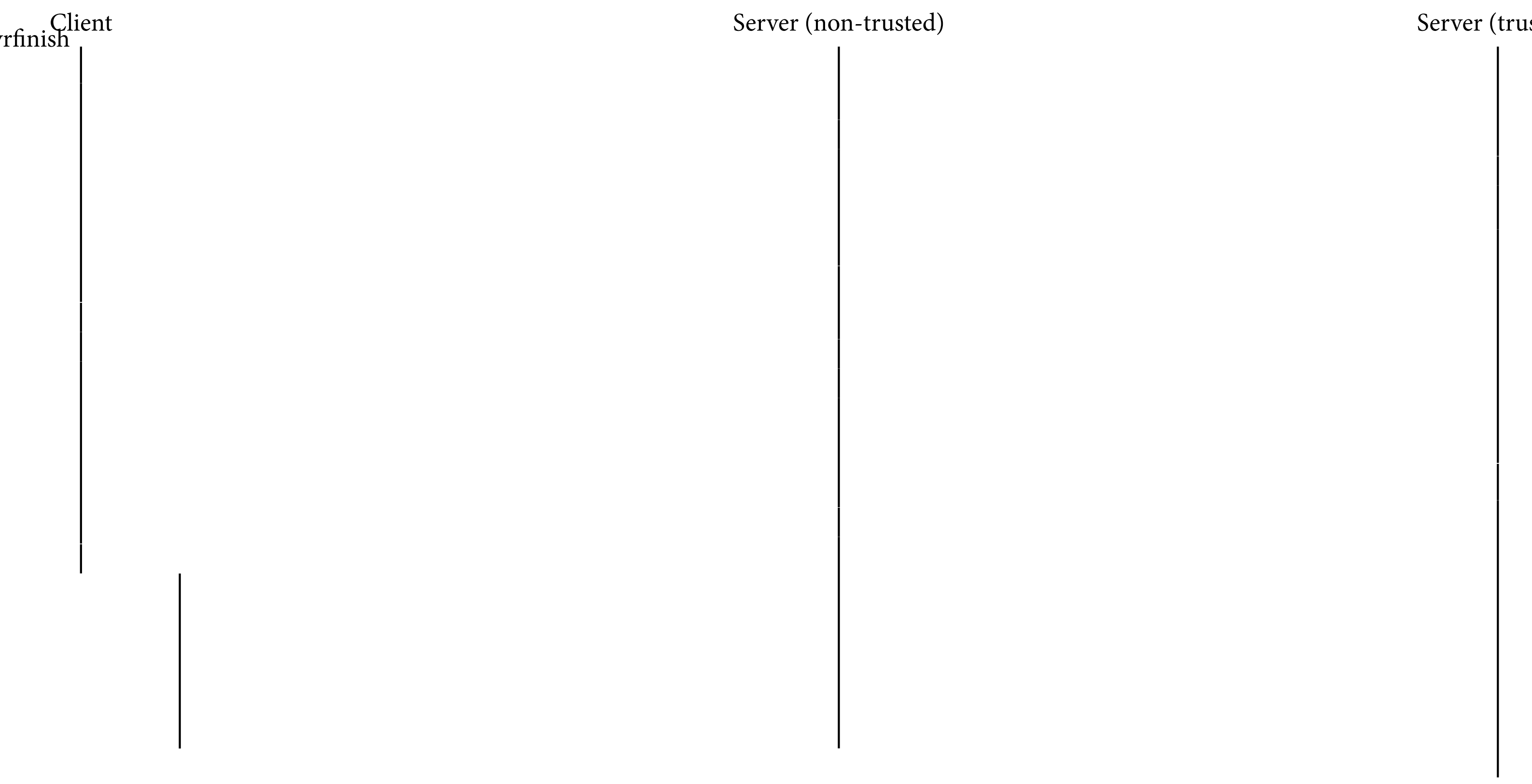

Figure 10: SSL Handshake with session keys hidden in the enclave

4 Implementation

5 Performance Evaluation

We developed our prototype using OpenSGX which, as detailed in Section 2, is an emulator for the SGX instruction set and relies on QEMU to execute the trusted code. Consequentially, we cannot measure end to end performance by simply executing our prototype and tabulating how many requests per second are completed, due to the overhead of emulation. Instead, we resorted to modeling the performance of the prototype in a manner similar to [4].

The remainder of this section describes the model we implemented to carry out our performance measurements and details the results we acquired from running our tests.

5.1 Performance Model

To model the performance of our prototype we implemented a second version that has no dependencies on OpenSGX and replaces certain SGX instructions with busy waits, simulating the overhead in executing them. In dong so, we make the same assumptions as [4] namely:

1. We assume that the CPU used in testing performs the same as an SGX-enabled CPU for all non-SGX instructions
2. We assume that the EPC is large enough to accommodate the entirety of the trusted component and any data structures created after program start-up

Leaving the overhead of SGX instructions, memory encryption, and asynchronous exits unaccounted for.

SGX instructions that bootstrap the enclave and verify its integrity are only executed once at startup, and therefore have no effect on performance. As a result, we only simulate the overhead of EENTER, ERESUME and EEXIT. Simulating the slow down in memory accesses, resultant from the memory encryption engine, was carried out by reducing the memory's clock in [4]. Such a proxy works for their system because its entirety executes on top of SGX. Clearly, such an approach would underestimate the performance of our system because only the trusted component incurs the overhead of memory encryption. In solving this we....

5.2 Results

5.3 Discussion

6 Project Management

Hello world!

Hello, here is some text without a meaning. This...

7 Conclusion

Hello world!

Hello, here is some text without a meaning. This...

References

[1] Andrea Bittau et al. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *NSDI*. 2008.

[2] Maxwell Krohn. "Building secure high-performance web services with OKWS". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. 2004. URL: http://portal.acm.org/citation.cfm?id=1247415.1247430.

[3] C. Latze and U. Ultes-Nitsche. *Transport Layer Security (TLS) Extensions for the Trusted Platform Module (TPM)*. RFC Draft. University of Fribourg, 2010. URL: https://tools.ietf.org/html/draft-latze-tls-tpm-extns-02.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[5] Intel Corporation. *Intel® Software Guard Extensions Evaluation SDK for Windows* OS*. 2010. URL: https://software.intel.com/sites/default/files/managed/d5/e7/Intel-SGX-SDK-Users-Guide-for-Windows-OS.pdf.

[6] Prerit Jain et al. "OpenSGX: An Open Platform for SGX Research". In: *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, Feb. 2016.