Jalapeno

Intel SGX Powered Cryptographic Library

Jeremy Erickson (jericks@umich.edu) Timothy Trippel (trippel@umich.edu)
EECS 582, Fall 2016
University of Michigan, Ann Arbor

Abstract

The wide prevalence of cloud computing has recently made a category of attacks based on subverting the underlying systems of cloud tenants very attractive. Previous work has explored hypervisor-based attacks that stealthily modify the Random Number Generators of victim virtual machines (VMs) to compromise the secret keys used to initiate secure web connections with clients. In this paper, we explore using Intel's Software Guard Extensions (SGX) to provide a single root-of-trust for security-critical operations and extend the end-to-end security model commonly deployed applications in untrusted virtual cloud environments. We find that SGX provides sufficient security guarantees to do this, and present Jalapeno, an SGX-enabled cryptographic library for enabling secure TLS sessions on untrusted platforms. Jalapeno has a simple API, supports two of Mozilla's ten recommended modern cipher suites, and is designed to be dynamically linked into larger projects. Lastly, we evaluate the performance of the core cryptographic operations implemented in Jalapeno and compare it to the performance of the same operations implemented in a widely used cryptographic library, OpenSSL.

1 Introduction

In the computer and network security domain, the Nation State Actor (NSA), is typically characterized as the strongest adversary imaginable. With nearly an unlimited amount of computing resources, funding, intelligent cryptographers and engineers, and the legal authority to get access to almost any information it needs, this adversary has been able to use its strategic advantages to create sophisticated attacks which consistently subvert modern day computing and communication systems [5, 6, 7, 8].

In previous work [5], we investigated how an NSA could leverage its resources and abilities to gain widespread access to modern communications and web services while avoiding public criticism through secrecy and stealth. Specifically, we focused on a hypothetical scenario in which an NSA has, through exploitation, coercion, or another method, acquired superuser privileges on some fraction of the host machines of a cloud service provider (i.e. has access to the hypervisor), such as those operated by Amazon (AWS) [12], Microsoft (Azure) [16], and Google (Compute Engine) [14].

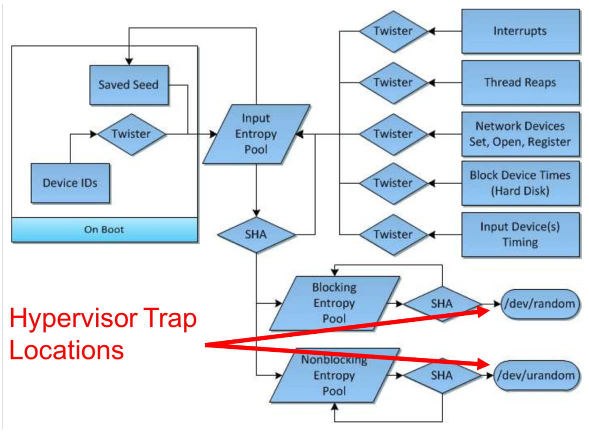


Figure 1: Hypervisor attack on the Linux Random Number Generator (RNG) as previously explored by [5]

We developed an attack against the Random Number Generator (RNG) of virtual machines through control of the hypervisor [5]. Through the hypervisor, the adversary can leverage full control over the output of /dev/random and /dev/urandom by trapping calls to read these file objects, as shown in Figure 1. This attack has several attractive qualities. First, it is possible to achieve control of the RNG without having to modify the virtual machine (VM) itself, meaning a cloud tenant, even one that inspects checksums of critical system components such as the kernel, is unlikely to detect any subversion. Second, with control over the RNG, cryptographic keys become predictable, and so there is no need to exfiltrate encryption keys out of the cloud infrastructure. Encrypted communications can be monitored, dragnet-style, from outside the cloud infrastructure with no suspicious key leakage shadowing encrypted messages. Third, since we can control the RNG with absolute precision, it is possible to cause the output random numbers to still appear truly random to any fourth party, thus facilitating the concept of a ”secure backdoor” that is very attractive to governmental organizations. This attack is weaponizable, stealthy, and has the potential to subvert the confidentiality of large swaths of the public Internet.

We believe that the community needs a defense against such a powerful attack, and propose to use recently-released hardware root-of-trust capabilities to provide secure cryptographic primitives, even on software running on an untrusted third-party's hardware. In August of 2015, Intel included Software Guard Extensions (SGX) [9] in its Skylake line of processors. SGX provides a hardware mechanism for developers to cryptographically verify the software that is being run on a given machine. Intel provides a service that can attest to the hardware integrity of the server in question, and the hardware itself can attest to the integrity of a given application running in a protected ”Enclave” mode. Enclave memory is inaccessible to software running outside the enclave, even to the host Operating System (OS) or hypervisor running in privilege ring 0, and is encrypted to prevent even physical cold-boot attacks. Thus, by leveraging SGX, we are able to perform critical RNG and cryptographic functions in a manner that precludes interference by a malicious third party.

In this project, we build a secure cryptographic library with the following key properties and features:

- Sensitive key material is kept solely inside the enclave, and is incapable of being leaked or predicted.
- Key material is securely ”sealed” (encrypted) and persisted to disk to avoid loss of data in failure conditions.
- Support for Elliptic Curve Diffie-Hellman key exchange and full Transport Layer Security (TLS) session key generation.

2 Background

2.1 Software Guard Extensions

In 2015 Intel released a new feature with their Skylake line of processors known as *Software Guard Extensions* (SGX). SGX provides developers with a secure execution environment, similar to ARM's *TrustZone* technology. However, SGX has very distinct capabilities than that of TrustZone. The secure execution sandbox that SGX provides, called an *enclave*, has user-level privileges and isolates the software contained in the enclave from the outside, untrusted, environment. Even, if the operating system, hypervisor, BIOS, or other hardware on the motherboard is compromised, only a denial-of-service (DoS) attack can be mounted. SGX preserves both the confidentiality (barring side-channel attacks) and integrity of the computation performed, and data stored, inside the secure enclave. Additionally, the enclave encrypts any code and data stored in a region of memory known as the Enclave Page Cache (EPC), which is a region of memory defined by the BIOS. Anything written to, or read from, the EPC is encrypted, or decrypted, by the hardware on the processor's die [9].

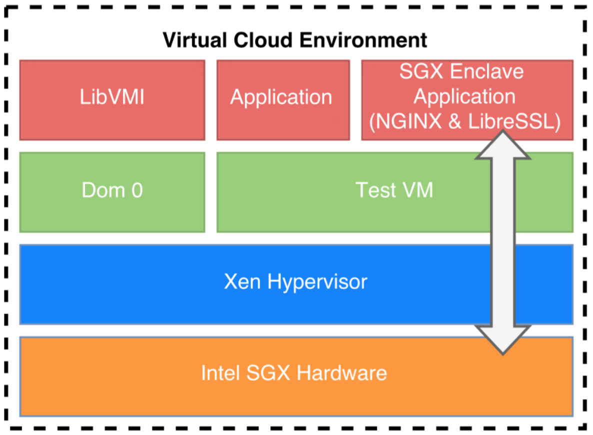

Virtual Cloud Environment
LibVMI
Application
SGX Enclave Application (NGINX & LibreSSL)
Dom 0
Test VM
Xen Hypervisor
Intel SGX Hardware

Figure 2: Diagram of a virtualized environment on single host machine using Intel SGX hardware, the `Xen` hypervisor, and `libvmi` inspection tool. Portions of security-critical applications, i.e. NGINX and LibreSSL, can execute in a secure SGX enclave to prevent an adversary with access to the hypervisor, i.e. through `libvmi`, or OS from subverting security critical tasks, i.e. random number generation. We note that Xen does not currently support SGX, but we expect it to soon.

Software applications that are built on the SGX platform are architected as shown in Figure 2. SGX applications are divided into two main components: trusted enclave code and untrusted code, as shown in Figure 4. The trusted enclave code is the portion of the application that executes in the secure enclave. Enclave code can access both encrypted enclave memory, the EPC, and unencrypted shared memory. Untrusted code can only access unencrypted shared memory. Untrusted code may initialize the execution of trusted enclave code by invoking an *ecalls*. Similarly, trusted enclave code can invoke the execution of untrusted code by an invoking an *ocalls*. The transfer of control flow between trusted code and untrusted code can only be done through the use of ecalls and ocalls. As described by Weisse et al. [3], the context switching between untrusted and trusted code invoked by ecalls and ocalls is similar to the context switching between VMs accomplished by VMENTER and VMEXIT with Intel's VT-x technology. To facilitate SGX application development, Intel provides an SGX software development kit (SDK) [13] to easily implement ecalls and ocalls by providing a tool that auto-generates "glue code" which performs state saving and restoring functions when ecalls and ocalls are made outside and within the secure enclave. The saving and restoring of execution state in and out of the secure enclave that happens during enclave context switches incurs significant overhead, as demonstrated in [3]. As a result, it is ideal for SGX on developers to minimize the use of ecalls and ocalls.

Since SGX enclaves are a sand-boxed context with user level privileges, there are specific limitations to what operation can be done inside an enclave. For instance, hardware device I/O, handling of interrupts, and paging are all operations that require assistance from the Operating System. Consequently, virtually all SGX applications must rely to some extent on untrusted software, and must remain vigilant to validate any externally-sourced data.

Essentially, Intel's SGX enables the secure execution of software in untrusted environment by allowing software developers to only trust Intel and the processor die they have manufactured. Additionally, Intel provides a *remote attestation* mechanism that allows users to verify that they are interacting with software contained in a real SGX enclave. The security guarantees Intel's SGX technology provides enable secure computation in untrusted cloud environment. Software developers that want to deploy applications and services on third party cloud computing infrastructures need only rely on a single root of trust: Intel's processor die.

2.2 Transport Layer Security

TLS underpins the confidentiality and integrity of the modern web. However, understanding how it works is integral to building a cryptographic library that supports the necessary required operations. TLS works by performing a session key negotiation between client and server, then symmetrically-encrypting all communications after the initial negotiation. [11, 10]

To start, the client will send the server a `ClientHello` message. The `ClientHello` will provide 28 random bytes, and a list of cipher suites that the client supports. The server will receive the `ClientHello`, will pick a cipher suite to use for the remainder of the connection, and send the client a `ServerHello` containing the picked cipher suite and its own set of 28 random bytes.

Jalapeno currently supports the `TLS_ECDHE_XXX_WITH_AES_128_GCM_SHA256` cipher suites, where "XXX" can be replaced with the Certificate Authority's signing mechanism (ECDSA or RSA). These are two of Mozilla's ten recommended cipher suites for modern browsers [2]. The `ECDHE` at the front refers to the Pre-Master Secret key generation algorithm, in this case Elliptic Curve Diffie Hellman Ephemeral; the `AES_128_GCM` refers to the symmetric cipher used to encrypt and decrypt messages; and `SHA256` refers to the hash algorithm used in the Pseudo-Random Function (PRF).[1]

Next, the server will send the client its Certificate, allowing the client to authenticate the server. This certificate will contain the server's public key, and will be signed by a valid Certificate Authority that the client trusts.

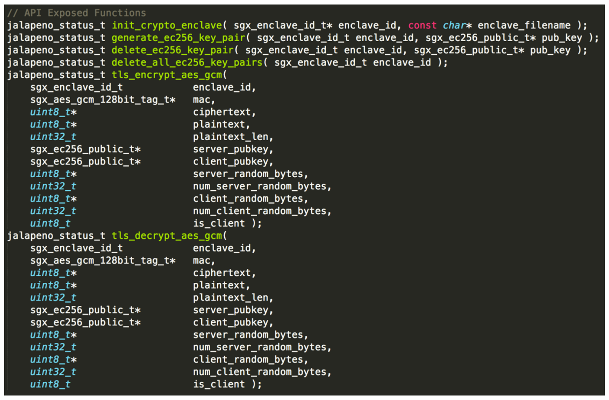
The server and client will send each other `ServerKeyExchange` and `ClientKeyExchange` messages, which contain the necessary information for generating the Pre-Master Secret. The mechanism for doing so differs between different cipher suites,[2] but in the case of `ECDHE`, the client and server simply send each other their public keys[3]. Using their own private keys and each others' public keys, as well as the 56 random bytes exchanged earlier in the connection, both the client and server compute the shared Pre-Master Secret. This Pre-Master Secret is used, with the 56 random bytes and static string "master secret", by the PRF to generate the Master Secret, and the Master Secret is used, again with the 56 random bytes and a new static string "key expansion", by the PRF to generate MAC keys, write keys, and Initialization Vectors (IVs) for the client and server. In `AES_128_GCM`, the client and server write keys and IVs are used to encrypt and decrypt session messages.

To conclude the handshake, the client and server will send each other `Finished` messages with a verification blob comprised of pseudo-random bytes generated from the Master Secret, a hash of the handshake up to this point, and the string literals "client finished" and "server finished", respectively. Once both the client and server verify these messages, the handshake is complete and subsequent data is encrypted with appropriate session keys.

[1]The PRF is a function for deterministically generating arbitrary numbers of pseudo-random bytes given a secret and seed value. This is useful for generating additional secrets using secrets that are already available.

[2]For instance, in some suites the client simply picks a pre-master secret, encrypts it with the the server's public key, and sends it to the server.

[3]The client may need to generate a public/private key pair for the connection, whereas the server can reuse its primary public key.



```
// API Exposed Functions
jalapeno_status_t init_crypto_enclave( sgx_enclave_id_t* enclave_id, const char* enclave_filename );
jalapeno_status_t generate_ec256_key_pair( sgx_enclave_id_t enclave_id, sgx_ec256_public_t* pub_key );
jalapeno_status_t delete_ec256_key_pair( sgx_enclave_id_t enclave_id, sgx_ec256_public_t* pub_key );
jalapeno_status_t delete_all_ec256_key_pairs( sgx_enclave_id_t enclave_id );
jalapeno_status_t tls_encrypt_aes_gcm(
    sgx_enclave_id_t            enclave_id,
    sgx_aes_gcm_128bit_tag_t*   mac,
    uint8_t*                    ciphertext,
    uint8_t*                    plaintext,
    uint32_t                    plaintext_len,
    sgx_ec256_public_t*         server_pubkey,
    sgx_ec256_public_t*         client_pubkey,
    uint8_t*                    server_random_bytes,
    uint32_t                    num_server_random_bytes,
    uint8_t*                    client_random_bytes,
    uint32_t                    num_client_random_bytes,
    uint8_t                     is_client );
jalapeno_status_t tls_decrypt_aes_gcm(
    sgx_enclave_id_t            enclave_id,
    sgx_aes_gcm_128bit_tag_t*   mac,
    uint8_t*                    ciphertext,
    uint8_t*                    plaintext,
    uint32_t                    plaintext_len,
    sgx_ec256_public_t*         server_pubkey,
    sgx_ec256_public_t*         client_pubkey,
    uint8_t*                    server_random_bytes,
    uint32_t                    num_server_random_bytes,
    uint8_t*                    client_random_bytes,
    uint32_t                    num_client_random_bytes,
    uint8_t                     is_client );
```

Figure 3: Jalapeno Cryptographic Software Library API. The function definitions shown above, describe the operations currently supported by the Jalapeno cryptographic library. Each function either instantiates an SGX enclave or immediately invoke ecalls into an SGX enclave where a specific sensitive operation is performed. In the future we plan to expand this interface to support more cipher suites and other security centric protocols (other than TLS).

3 Methodology

Our goal is to design a cryptographic library that can support TLS sessions for modern web browsing. Existing libraries, such as OpenSSL [18] and LibreSSL [15], are designed around easy access to important key material for authentication and encryption. However, Jalapeno aims to restrict access to secret key material to code running in the enclave. Therefore, all authentication and encryption functions that require access to secret key material must be implemented within the enclave. All actions that occur outside the enclave are untrusted.

In evaluating the fundamental actions that must be supported for a TLS session to occur, we have developed a public API that supports the following:

- Creation of public/private key pairs.
- Encryption of plaintext into ciphertext.
- Decryption of ciphertext into plaintext.

We have also implemented less-interesting, but necessary, functions to initialize the enclave, delete key pairs, and print out debug information.[4] Jalapeno's public API is presented in Figure 3.

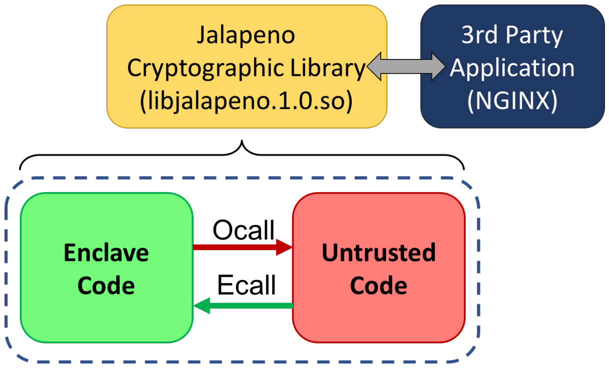

Jalapeno Cryptographic Library (libjalapeno.1.0.so)
3rd Party Application (NGINX)
Enclave Code
Ocall
Ecall
Untrusted Code

Figure 4: Jalapeno SGX Shared Library Software Architecture. The Jalapeno library is architected according to the SGX application model, with both a secure enclave code component and traditional untrusted code component. Jalapeno is compiled as a Linux shared object. Jalapeno is designed to be incorporated into existing security critical applications that are typically deployed in untrusted cloud environments, like the NGINX [17] or Apache2 [20] web servers.

3.1 Asymmetric Key Creation

Actually generating public/private key pairs is straightforward from within the Enclave. The SGX included libraries provide functionality to generate ECDSA 256-bit key pairs within the context of a particular enclave's execution. As we do not currently attempt to support alternative key pair types, this suffices.

What is more interesting is the policy we have chosen to manage multiple key pairs in a given enclave. While a typical use case may assume that a given web server will host only a single website, and therefore only need a single key pair, in reality many production servers host multiple TLS-enabled websites. Jalapeno supports the creation and storage of multiple key pairs simultaneously and allows untrusted (non-enclave) code to identify them by the corresponding public key. Key pairs are stored in a fixed size array, currently set to a size of 64. Keys may be invalidated (lazy deletion) at will, and new keys will be stored in the lowest available index. Key pair access incurs a linear search across the entire key pair array. We intentionally do not short-circuit this search or use a faster data structure to avoid a timing side channel.

To protect against failure conditions, we seal this key pair array to disk after any modification to the data structure. This requires using SGX's sealing functionality to encrypt the key pair array and an OCALL to our untrusted code to write this encrypted data structure to disk. If on any access to the in-memory key pair array, we de-

[4]Debug functions will necessarily be removed before the enclave is ready for real-world use.

tect that it is not initialized (which may happen after the application is restarted, the server is power cycled, etc.) we will attempt to load it from disk. This also requires an OCALL to read the encrypted data structure from disk, but SGX's unsealing procedure performs a validity check on the data structure before reconstructing it into our key pair array. A failure condition during an update will cause the update to fail, but should not negatively impact our stored key pair array. We don't anticipate this will cause any problems, as asymmetric keys are not typically automatically provisioned, but provisioned as part of a new manual site installation. We consider failure conditions for the persistent storage medium to be addressed through automated backup systems, and thus we consider it a solved problem outside the scope of this paper.

3.2 Encryption and Decryption

The SGX included libraries conveniently also include functionality to generate 256-bit shared Elliptic Curve Diffie Hellman (ECDH)[5] secrets, perform AES 128-bit GCM encryption and decryption on a buffer, and generate SHA256 hashes. Notably, it does not provide TLS's PRF or an HMAC function. Therefore, following RFCs 2246 and 5246 [11, 10], we implemented our own compliant PRF and HMAC functions.

In a TLS handshake, the client and server will exchange public keys and 28-byte random nonces, then begin sending encrypted payload messages. The server will need to perform *encrypt* and *decrypt* functions, and Jalapeno provides these directly in its API. The untrusted code may pass the enclave the plaintext (or ciphertext) and these four public parameters, and receive the corresponding ciphertext (or plaintext). Using the provided ECDH function, provided client public key, and retrieved server private key (using the server's public key), Jalapeno will compute the shared TLS Pre-Master Secret. The Pre-Master Secret and two random nonces yield the Master Secret using the PRF function, and the Master Secret with the PRF function yield the various keys used by the SGX-provided AES encryption and decryption functions to manipulate the payload.

Currently, Jalapeno recomputes these session keys for every encrypt or decrypt operation. This is inefficient, but as shown in Figure 5, does not appear to incur a significant fixed cost when compared against the cost of encrypting buffers of 10KB or larger. We can improve this by caching these keys for the duration of the session and we intend to in future work. However, we have avoided implementing this logic so far, as we anticipate the precise requirements to become more apparent as we integrate with a production web server.

[5]The difference between ECDH and ECDHE algorithms is simply that in the latter, instead of using the keypair associated with the site's certificate, the server generates a new keypair for each connection, signs it with the certificate's keypair, and deletes it after the session closes.

4 Evaluation

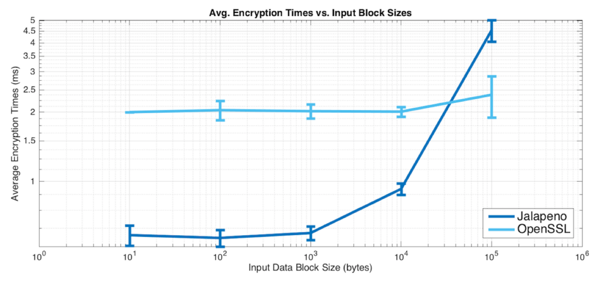
We evaluated the performance of the core cryptographic operations incorporated into the Jalapeno library with respect to the performance of the same cryptographic operations incorporated into the widely used OpenSSL [18] cryptographic library. We also provide an analysis of the overhead incurred by ecalls and ocalls that Jalapeno operations invoke when jumping in and out of the SGX enclave. The main goal of our evaluation was to provide insights into the performance loss suffered when performing sensitive cryptographic computations in an SGX enclave. Our evaluation indicates that performance loss is manageable and most likely worth the added security benefits of utilizing Intel's SGX technology to perform sensitive computations in untrustworthy cloud environments.

4.1 Encryption and Decryption

We evaluate how Jalapeno performs compared with OpenSSL when encrypting and decrypting various packet sizes. We utilize the *clock_gettime* function with the *CLOCK_PROCESS_CPUTIME_ID* parameter in the Linux C Standard Library to measure the execution time of the AES128-GCM encryption and decryption functions in Jalapeno. To measure the execution times of the AES128-GCM encryption and decryption functions in OpenSSL we used the UNIX *time* command and acquire the *real* time.

Figure 5 shows the execution times of the encryption and decryption functions in Jalapeno vs. OpenSSL. Data packet sizes of 10, 100, 1,000, 10,000, and 100,000 bytes were encrypted and decrypted using both libraries. For each packet size 100 trials were run and the means were plotted. The error bars indicate the standard deviation of execution times for the 100 trials for each packet size.

At first glance, it might seem that Jalapeno drastically out-performs OpenSSL, even with performance overhead incurred from SGX ecalls and ocalls. However, when deciphering the plots in Figure 5, it is vital to consider how the execution times of each library's encryption and decryption functions were measured. OpenSSL functions were measured with the UNIX time command, and take into consideration the time it takes to load OpenSSL into memory and begin program execution. Jalapeno functions were measured using calls to time functions in the C Standard Library, a time measurement which does not take into consideration the time it



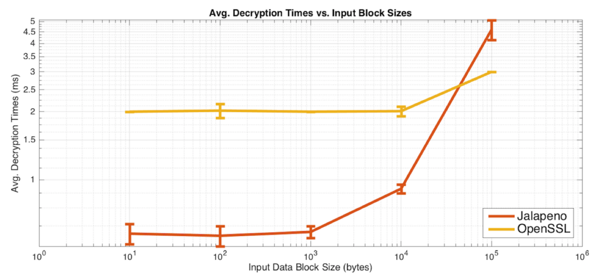


Figure 5: Execution times of AES128-GCM encryption and decryption functions in Jalapeno vs. OpenSSL. The execution time of AES encryption/decryption with Jalapeno was measured using the time functions in the Linux C Standard Library, which did not include program loading time in the measurement. The execution time of AES encryption/decryption with OpenSSL was measured using the UNIX *time* command, which included program loading time in the measurement. It is important to note the general trend indicated by each line plot, not the difference between the two lines, as different measurement mechanisms had to be used. Jalapeno performs worse than OpenSSL: a greater exponential-like increase in execution time when encrypting/decrypting increasing data packet sizes. This is expected as SGX requires copying the memory containing the ciphertext/plaintext to and from the secure enclave's encrypted memory (EPC) for encryption/decryption operations.

takes to load Jalapeno into memory and begin its execution. Because of this difference, it is wise to analyze the trend of each OpenSSL and Jalapeno line, in both encryption and decryption plots, rather than the difference between them. With this in mind, it is clear that the execution times of Jalapeno's encryption and decryption functions increase rapidly as larger the packet sizes are input. This is mostly due to the fact that the Jalapeno encryption/decryption functions invoke an immediate ecall which must copy the plain-text/cipher-text data packet, byte-by-byte, into enclave memory before the encryption/decryption algorithm can begin. In OpenSSL, there is no ecall overhead, and its encryption/decryption execution times scale better with packet size.

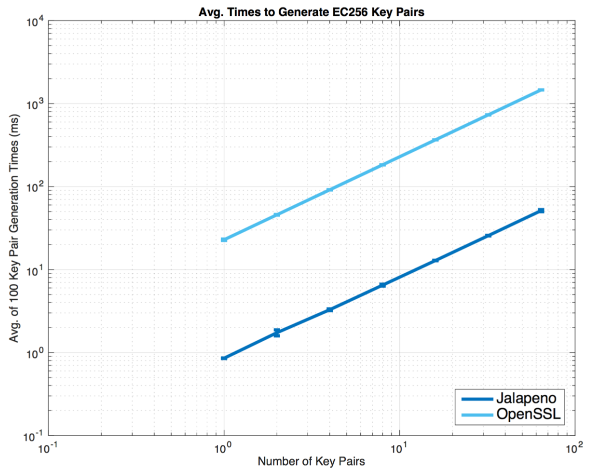


Figure 6: Execution times of functions in Jalapeno vs. OpenSSL that generate 256-bit elliptic curve (EC256) key pairs. The execution time of generating EC256 key pairs with Jalapeno was measured using the time functions in the Linux C Standard Library, which did not include program loading time in the measurement. The execution time of generating EC256 key pairs with OpenSSL was measured using the UNIX *time* command, which included program loading time in the measurement. It is important to note the general trend indicated by each line plot, not the difference between the two lines, as different measurement mechanisms had to be used. Jalapeno performs similar to OpenSSL: a linear increase in execution time when generating increasing numbers of key pairs.

4.2 Elliptic Curve Key Pair Generation

We evaluate how Jalapeno performs compared with OpenSSL when generating 256-bit elliptic curve (EC256) key pairs. We utilize the *clock_gettime* function with the *CLOCK_PROCESS_CPUTIME_ID* parameter in the Linux C Standard Library to measure the execution time of the EC256 key pair generator function in Jalapeno. To measure the execution times of the EC256 key pair generation function in OpenSSL we used the UNIX *time* command and acquire the *real* time.

Figure 6 shows the execution times of the 256-bit elliptic curve key pair generation functions in Jalapeno vs. OpenSSL. Varying number of key pairs were sequentially generated, 1, 2, 4, 8, 16, 32, and 64 key pairs, using both libraries. For each number of sequentially generated EC256 key pairs, 100 trials were run and the means were plotted. The error bars indicate the standard deviation of execution times for the 100 trials for each set of key pairs generated.

Similar to the encryption and decryption times shown in Figure 5, at first glance, it may seem that Jalapeno out-performs OpenSSL when generating EC256 key pairs. Again, when deciphering the plots in Figure 6, it is vital to consider how the execution times of each library's EC256 key pair generation functions were measured. Consequently, it is again wise to analyze the trend of each line as a whole in Figure 5, rather than the difference between them. With this in mind, it seem Jalapeno's EC256 key pair generation function performs quite similarly to the EC256 key pair generation function in OpenSSL. This seems logical as the ecall that is immediately invoked by calling the EC256 key pair generation function in Jalapeno does not requiring copying large buffers of data into the SGX enclave. Rather, the ecall simply transfers the execution control flow to the enclave where the key generation computation is launched. The private key is stored within the enclave (which seals and stores a back up on disk), and only the public key is transferred out of the enclave when the ecall returns. Since the keys have a fixed size, the number of public key bytes transferred out of the enclave remains the same upon every invocation of the EC256 key generation ecall. Therefore, the performance of Jalapeno is similar to the performance of OpenSSL: the time to generate multiple key pairs increases linearly with the number key pairs requested.

5 Related Works

To the best of our knowledge, using the hypervisor to poison a random number generator has only previously been explored by Alt et al. [4] and Erickson et al. [5]. In the first work, the authors explored three ways to poison the output of the Linux RNG, one of which involved modifying the QEMU[19] emulator to intercept calls to specific RNG functions in a VM and alter their return values. Escalating this attack, Erickson et al. showed how using a flexible virtual machine introspection (VMI) tool, closely coupled with the Xen hypervisor, an attacker can attach to a running VM, trap function calls to the Linux RNG, and alter their return values. They demonstrate that using a VMI tool allows the attacker to dynamically adapt their attack to target cryptographic applications running within a VM instance as well. The goal of our work here is to defend against these kinds of attacks by leveraging Intel's SGX technology to enable trustworthy computing in untrusted environments.

Weisse et al. [3] have provided the first performance analysis of the Intel's SGX architecture. They found that since invoking ecalls and ocalls to jump into and out of an SGX enclave can incur cycle overheads between 8,200 and 17,000 cycles, their can be a significant performance overhead to incorporating the use of SGX technology into existing applications. Given that system calls cannot be invoked from within an enclave they require the invocation of an ocall to first exit the enclave, as previously mentioned in Section 2.1. Weisse et al. measure the performance overhead incurred by porting popular system call heavy applications with intensive network requirements, such as memcached, openVPN, and lighttpd, to the SGX application architecture. They found that performance degradation, with regards to network latency and throughput was as high as 79% in some cases. They propose a mechanism, called *HotCalls*, to replace the ecall and ocall SGX enclave interface, that trades process thread resources for SGX application performance gains. Their approach to SGX ecalls and ocalls, *HotCalls*, is able to provide a 13–27x speedup compared to the existing SGX enclave interface.

In 2014, Microsoft published *Haven*, an SGX-enabled platform for protecting unmodified application data from privileged processes on third-party hardware. Haven works similar to the container model, where each application is isolated in its own runtime environment, and splits the Operating System into two main components. The untrusted kernel component manages the hardware resources of the server, while the trusted component, replicated in each application's SGX enclave, provides a Windows-compliant ABI that manages threading, virtual memory, and file system access. Haven takes the approach of attempting to migrate entire applications inside the enclave in a general fashion. While this may make adoption of SGX possible for some applications, it has two primary downsides. First, Haven is monolithic, cluttering *each* enclave with the majority of the Windows runtime OS. This makes its memory usage inefficient, and its inability to handle basic operations such as interrupts that cannot be handled within the enclave lead to performance penalties of 13% in the best case, 31-54% in the common case, and in the wost case, only described as "relatively poor". Second, Haven's monolithic nature makes it problematic to verify for security guarantees. In comparison, Jalapeno aims to provide a verifiable code base to handle a specific task. While Jalapeno is not yet mature enough to conclude much about its eventual performance, we anticipate dramatically better performance than Haven is able to provide. Jalapeno also allows simultaneous operation of non-SGX applications on the same unmodified OS and hardware.

6 Discussion

6.1 Load Balancing

One of Jalapeno's key considerations is that the secret key material cannot be exported outside the enclave. However, real world systems, particular at scale, require distribution across multiple servers, and often across geographically diverse data centers. In this paper, we primarily focus on the use case of a single web server requiring the ability to host web content to a limited set of users, but Jalapeno is extendable to a fully distributed setup for load-balanced operation and arbitrarily extensible throughput.

Jalapeno already handles server failure conditions by sealing[1] its key material to disk in an encrypted form. Currently in our development enclave, this key material is encrypted with the *Signing Identity*, the default option. The Signing Identity refers to the public key used to sign the enclave, and in our development version, this is a debug key. However, a production enclave would have a Signing Identity that in turn is signed by Intel and would be controlled by the Enclave developer. In this form, additional enclaves, also signed with the same Signing Identity, can unseal the key material and use it. Thus, the ability for the enclave's key material to be exported outside the enclave is reduced to the security of the Signing Identity's private key.

However, another mode is possible with trivial modification. When sealing the key material, we can optionally specify that it be encrypted using the *Enclave Identity*. The Enclave Identity is an entity that is formed by hashing the enclave memory as it is initially being constructed, and is unique to the exact enclave being executed. However, this enclave can be executed on multiple machines, each constructing an identical Enclave Identity and therefore able to unseal the keys. Since a second enclave must be functionally identical, by simply ensuring that the enclave code has no functionality to export the key material in any unsealed form, we ensure that *any* enclave able to unseal the secret key material will be unable to export it. This does, however, preclude software updates to the enclave code base, as updated enclaves will be unable to access the key material of previous versions. Our development code uses the Signing Identity for convenience, but there is no technical barrier to using the Enclave Identity, and we plan to do so in later versions of Jalapeno and before making it available for use in real systems.

6.2 Integration with a Production Web Server

We have performed only limited investigation into what it will take to integrate Jalapeno with a production-grade web server. In particular, we have in prior work[5] explored the Apache2 web server and OpenSSL cryptographic library. OpenSSL is widely regarded to be poorly maintained, and a successor project, LibreSSL has sprung up to take its place with a backwards-compatible API. We found the Apache2 code base to be similarly difficult to interpret, and decided to explore another popular web server, NGINX. Our preliminary exploration of these four code bases leads us to believe that the most straightforward path for integrating Jalapeno in a production web server is to fork and modify the code bases of the NGINX and LibreSSL code bases.

One potentially-large complication for our efforts is that production web servers are written in a particularly abstract object-oriented form. Connections will have properties that refer to metadata such as the connection type, which may refer to a TLS connection or not. TLS connections will have further properties that specify things like the specific cipher suite being used. While we have not yet tracked down the precise mechanisms used for encrypting and decrypting TLS traffic in LibreSSL, we anticipate that it is sufficiently abstract that it may be more difficult than simply shimming an "encrypt" function and redirecting its execution to use Jalapeno instead. We plan to explore this further in future work, but we anticipate this will require a substantial engineering effort to accomplish.

7 Conclusion

Ultimately, Jalapeno is only one step towards updating end-to-end security for cloud computing, but it's an important step. Adversaries with control over the cloud computing base can currently subvert privileged cryptographic operations, and Jalapeno is the first cryptographic library that leverages hardware protections to guarantee that secret key material cannot be leaked or tampered with. Our performance results show that there is some penalty to moving these operations into an Enclave, but our plans to enable load-balancing will still allow Jalapeno to scale to meet a distributed workload. Going beyond the work presented in this paper, future work will entail full integration into a production web server and a full system performance analysis that can comprehensively analyze the security/performance trade-off space.

References

[1] Alexander B. *Introduction to Intel® SGX Sealing*. May 4, 2016. URL: https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

[2] Julien Vehent. *Security/Server Side TLS*. Oct. 31, 2016. URL: https://wiki.mozilla.org/Security/Server_Side_TLS#Recommended_configurations.

[3] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining Lost Cycles with HotCalls: A New Fast Interface for SGX Secure Enclaves". In: *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. Under submission. 2016.

[4] Matthew Alt et al. *Entropy Poisoning from the Hypervisor*. Unpublished class project. 2015. URL: https://courses.csail.mit.edu/6.857/2016/files/alt-barto-fasano-king.pdf.

[5] Jeremy Erickson, Timothy Trippel, and Andrew Quinn. *Cloaking Order in Chaos: Subverting the random number generator via the hypervisor*. Unpublished class project. 2015. URL: https://jeremy-erickson.com/static_docs/EECS588/paper.pdf.

[6] April Glaser. *After NSA Backdoors, Security Experts Leave RSA for a Conference They Can Trust*. Jan. 30, 2014. URL: https://www.eff.org/deeplinks/2014/01/after-nsa-backdoors-security-experts-leave-rsa-conference-they-can-trust.

[7] Olga Khazan. *The Creepy, Long-Standing Practice of Undersea Cable Tapping*. July 13, 2013. URL: http://www.theatlantic.com/international/archive/2013/07/the-creepy-long-standing-practice-of-undersea-cable-tapping/277855/.

[8] Mandient. *APT1. Exposing One of China's Cyber Espionage Units*. Feb. 19, 2013. URL: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf.

[9] Frank McKeen et al. "Innovative instructions and software model for isolated execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013. URL: http://dl.acm.org/citation.cfm?id=2488368.

[10] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol, Version 1.2*. RFC. 2008. URL: https://tools.ietf.org/html/rfc2246.

[11] T. Dierks and C. Allen. *The TLS Protocol, Version 1.0*. RFC. 1999. URL: https://tools.ietf.org/html/rfc2246.

[12] *Amazon Web Services*. URL: https://aws.amazon.com/.

[13] Intel Corporation. *Intel SGX Software Development Kit (SDK)*. URL: https://software.intel.com/en-us/sgx-sdk.

[14] *Google Cloud Compute*. URL: https://cloud.google.com/.

[15] *LibreSSL*. URL: https://www.libressl.org/.

[16] *Microsoft Azure*. URL: https://azure.microsoft.com/en-us/.

[17] *NGINX*. URL: https://www.nginx.com/.

[18] *OpenSSL Project*. URL: http://www.openssl.org/.

[19] *QEMU: Open Source Processor Emulator*. URL: http://wiki.qemu.org/Main_Page.

[20] *The Apache HTTP Server Project*. URL: https://httpd.apache.org/.