Securing Whole Applications with SGX

Tobias Konopik
Friedrich-Alexander University
Erlangen-Nuremberg, Germany

ABSTRACT

Intel Software Guards Extension (SGX) is a CPU-based mechanism for creating a Trusted Execution Environment (TEE), called an enclave, for user-level application code. The enclave is a hardware-isolated runtime environment whose memory is isolated from other hardware and software on the system, including the Operating System (OS) and hypervisors. This isolation mechanism comes at the cost of severely restricting the application's programming model, since standard OS abstractions, such as Input/Output (I/O) operations, are not safely available. This makes it necessary to adapt applications to run in a standard enclave context. Using SGX to run unmodified applications requires a secure runtime environment that provides the necessary C Standard Library (libc) interface that interacts securely with the untrusted host OS services. The most promising approaches to implementing such an environment are library OS or container-based.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Security and privacy** → **Distributed systems security**;

KEYWORDS

Cloud Computing, Trusted Computing, Intel SGX

ACM Reference Format:
Tobias Konopik. 2023. Securing Whole Applications with SGX. In *Proceedings of AKKS Conference (AKSS2023),* Arne Vogel and Rüdiger Kapitza (Eds.). ACM, New York, NY, USA, 7 pages. https://doi.org/00.048/191.058

1 INTRODUCTION

The increasing use of cloud computing to analyze confidential and sensitive information, such as medical data, requires mechanisms to ensure its security and integrity. In a traditional cloud computing environment, data security is the sole responsibility of the cloud provider. This requires unwavering trust in their delivery of secure and confidential systems. However, the cloud infrastructure provided typically only protects the provider's privileged code from malicious intrusion by application code. On the other hand, there is no protection for the application code and data from the provider or from malicious actors with super-user access to the provider's system hardware or software stack.

The goal is to provide secure and confidential data handling by establishing a TEE that is shielded from these malicious attackers. A TEE is a processing environment that is protected from all software and physical attacks, guarantees the authenticity of the executed code and data, and provides remote attestation to authenticate its trustworthiness to third parties [20].

An exemplary hardware-based mechanism for providing a TEE is the Intel SGX provided in its processors. It allows complete isolation of an application from the host system by providing an isolated virtual memory address space whose confidentiality and integrity are guaranteed. To ensure the integrity of the application, it provides an attestation mechanism at startup and restricts control flow by providing only specific entry points.

Ideally, the benefits provided by a TEE should be easily accessible to legacy applications. However, an enclave imposes several restrictions on the application code it encloses, such as not allowing the execution of system calls, so a legacy application cannot run in a standard enclave environment. To allow an unmodified application to run, a compatibility layer is required, which provides the application with a trusted libc internal interface and a shielded external interface to the untrusted host OS to use system calls.

The main goals of this compatibility layer are:

(i) Support legacy applications without modification
(ii) Minimize the Trusted Computing Base (TCB)
(iii) Minimize additional performance overhead

Currently, the most mature approaches that best fullfill the defined goals for the compatibility layer are the container-based approach used in Secure CONtainer Environment (SCONE) [3] and the library OS-based approach used in Gramine-SGX [26]. This paper first introduces both frameworks and then compares them using the aforementioned goal metrics.

2 BACKGROUND

This section summarizes the features of SGX and gives a brief introduction to the basic principles of the frameworks.

2.1 Intel SGX

Intel SGX [7] [8] is a hardware-based mechanism for creating a TEE, called an enclave, that ensures the confidentiality and integrity of the application running within it.

2.1.1 Enclave. The basic building block of an enclave is an isolated memory region in the Dynamic Random Access Memory (DRAM), called the Processor Reserved Memory (PRM), which is inaccessible to code running outside the enclave. The Enclave Page Cache (EPC) is the main subset of the PRM and contains all the pages reserved for enclaves. To ensure the freshness, confidentiality, and integrity of enclave data, these pages are encrypted in the DRAM using a dedicated hardware unit in the CPU called the Memory Encryption Engine (MEE) [11]. A processor can support multiple enclaves running in parallel, but pages of the EPC cannot be shared between enclaves and must be allocated exclusively.

AKSS2023, January 2023, Erlangen, Germany

ACM ISBN 499-1-3185-2727-6/22/10…$1,000,000.00
https://doi.org/00.048/191.058

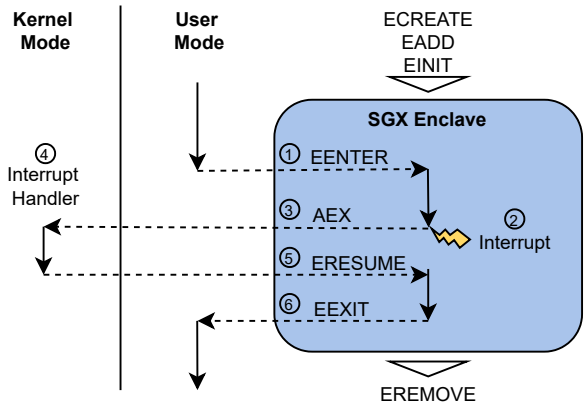

Figure 1: Enclave Thread Lifecycle

An enclave supports multiple concurrent threads and also allows dynamic creation and destruction of new threads on newer processors [28]. Code running inside an enclave is always in the lowest privilege mode, i.e. user mode. Enclave code has direct access to non-enclave memory, which must be used with care, as its security and integrity are not guaranteed.

2.1.2 Enclave Lifecycle. An enclave is created with the *ECREATE* instruction, which initializes the SGX Enclave Control Structure (SECS), the main data structure containing the enclave's metadata, from a free EPC page. The untrusted system software then loads the enclave code and data by invoking the *EADD* instruction, which cryptographically measures the memory layout of the added page in the enclave. For each thread running inside the enclave, a Thread Control Structure (TCS) must be initialized. After loading is complete, the system software must call the *EINIT* instruction to complete the initialization, for which it needs a special initialization token granted by Intel's proprietary *Launch Enclave*. On newer processors, it is also possible to dynamically add and remove pages from the PRM after the initialization [28] has been completed. The *EREMOVE* instruction tears down an enclave by freeing enclave pages as long as they are not currently being used by an enclave thread.

2.1.3 Enclave Thread Lifecycle. The thread lifecycle inside the enclave is illustrated in Figure 1. To initially enter an enclave, a process must invoke the *EENTER* instruction ①, which takes the address of an unused TCS as a parameter. The instruction puts the processor in enclave mode, allowing the thread to access the EPC pages of the specified enclave. The instruction sets the Relative Instruction-Pointer (RIP) to the specified entry point in the TCS, restricting the entry to a predefined point, similar to the invocation of system calls in the OS. When a hardware exception, fault, or interrupt is triggered for an enclave thread ②, the enclave executes a Asynchronous Enclave Exit (AEX). During the AEX ③ the current processor context is then stored in a secure memory area in the enclave, the State Save Area (SSA), to protect the secrets of the enclave from the rest of the system. After leaving the enclave, the thread enters the exception handling routine of the host OS ④. When the interrupt context terminates, the thread can re-enter the enclave by calling the *ERESUME* instruction ⑤, which restores the stored processor state from the SSA and resumes execution of the enclave code. The thread synchronously exits the enclave by calling the *EEXIT* instruction ⑥, which returns the processor to user mode and restores the context stored in *EENTER*.

2.1.4 Software Attestation. Certifying the contents of an enclave requires a cryptographic signature, which is provided by the software attestation mechanism. SGX implements two attestation schemes, one for local attestation on a machine and one for remote attestation. Local attestation is performed using the *EREPORT* command, which uses the measurements calculated during enclave initialization and a special tag to generate an attestation report. Remote attestation requires an additional proprietary *Quoting Enclave* from Intel, which verifies and signs the report before sending it to the remote party.

2.1.5 Limitations. Because the enclave code runs in lowest privilege mode, it cannot perform I/O operations without interfacing with the system software, making it impossible to make system calls from inside the enclave. To make a system call, the enclave must be exited. The parameters and result of the system call must be exchanged with the environment by accessing non-enclave memory, a so-called *shield* then checks the validity of the results. Context switches in and out of the enclave are expensive operations, nearly 60 times slower than a context switch between user and kernel mode [27], which also increases with the size of the moved memory buffer [10].

2.2 Container

Containers are a mechanism for using OS-level virtualization to isolate processes [23]. In the Linux kernel, this is primarily accomplished by using the kernel features of *namespaces* and *cgroups*. The Linux *namespaces* feature provides processes with their view of the system by wrapping the global system resource in an abstraction. A process *namespaces* thus defines its usable resources. Linux currently has eight different namespaces: *cgroups*, IPC, Network, Mount, PID, TIME, Users, and UTS [13]. The *cgroups* feature allows the partitioning of tasks into hierarchical groups. These groups can then be used by the kernel's resource controllers to monitor and limit resource allocation [18][14].

2.3 Library OS

Historical library OS or *program libraries* provide general OS abstractions as so-called *supervisor routines* that run in the same context as the application code [5]. In contrast, the modern library OS is characterized by running the OS interface on which an application depends in the application address as a library [19]. The implemented library OS interfaces with the supervisor host kernel with a small fixed set of abstractions. Thus, the libOS can be seen as a lightweight virtual machine implemented as a *picoprocess*. A *picoprocess* is a stripped down virtual machine running in an isolated memory region, but without direct device access privileges. Instead, it is provided with a very narrow system call interface to interact with its environment [9]. This approach has the advantages of providing security through application isolation and faster

independent evolution of OS components, and can provide better performance for applications by using a customized libOS [25].

2.4 Threat Model

The threat model assumes a malicious adversary with privileged access to the entire hardware and software stack of the system. This renders all other system software, including the OS and the hypervisor, untrustworthy and allows the modification of any file in unprotected memory areas as well as the I/O traffic of the system. The only trusted resources are the TCB, which is is generally defined as the "software and hardware on which security depends" [15]. It includes software and hardware components that are responsible for enforcing the security policy of the system; in this paper, the TCB includes the CPU and the code running inside the enclave. In general, the TCB should be as simple as possible in terms of the function it must perform in order to reduce its attack surface [1].

3 FRAMEWORK DESIGNS

This section provides an overview of the SCONE and Gramine-SGX framework designs for securing applications.

3.1 SCONE

To protect the integrity and confidentiality of the application and its data, SCONE [3] uses a so-called *secured container* to take advantage of the features of SGX. It uses a shielded external enclave interface based on the system calls Application Programming Interface (API) to interface with the OS host. The application inside the enclave is provided with a libc interface by including a custom musl libc library in the container. To use the SCONE framework, the application must be statically linked against the SCONE libraries. In addition, the *secured container* integrates with the Docker container environment for easy deployment and use. The design of the framework is shown in Figure 2 and will be discussed in the following sections. The main features of SCONE are the interface shielding, the M:N threading model, the asynchronous system call interface, and the Docker integration of the *secured container*.

3.1.1 Interface Shielding. The shielding library serves two purposes. The first is to protect the application from low-level attacks via the container's external interface, provided by the system call shield. The second is to ensure the confidentiality and integrity of the data passing through the OS, which is provided by the file, network and console shield.

1) System call shield: One of the main threats that must be dealt with when shielding a very large interface such as the system call API is the so-called *Iago attacks* [6]. In this attack, the kernel can manipulate the application by providing malicious system call return values. The system call shield checks the validity of system call results Section 3.1.3, this includes checking the given buffer sizes as well as the destination of given pointers to protect against these attacks.

2) File system shield: The file system shield protects the confidentiality and integrity of the application's files. The shield encrypts files before saving them to disk by splitting a given file into blocks of fixed size to allow detection of file modifications by the environment. It also assigns a unique authentication tag to each block to ensure the freshness of the data and to protect against so-called

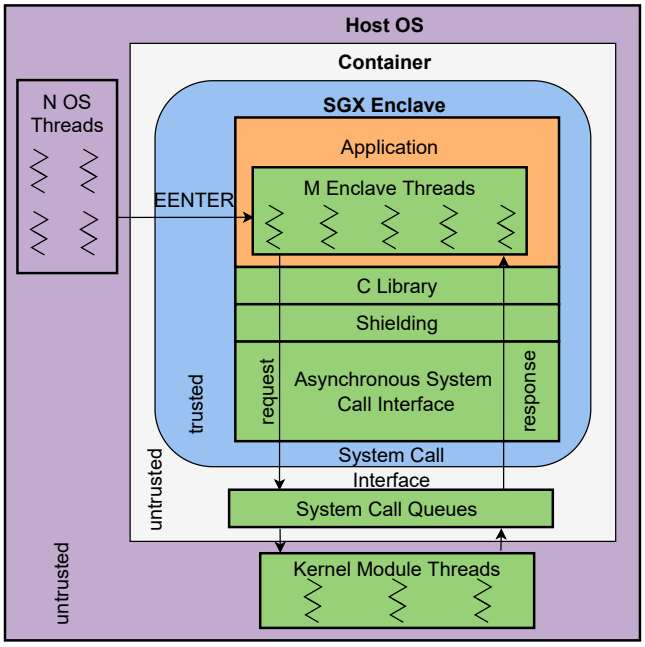

Figure 2: SCONE Design

rollback attacks, where the system state is rolled back to a previous state.

3) Network shield: The shield encrypts all network traffic to ensure its security, including inter-container communication. The network shield wraps all sockets created inside the container in a Transport Layer Security (TLS) layer, which can be achieved without any application level changes.

4) Console shield: To allow the attachment of the container environment an I/O stream of the containerized application must be provided that protects the confidentiality of the data being sent. The console shield uses symmetric encryption to send the contents of the stream with block-sized granularity, and adds protection against message reordering or replay by assigning a unique identifier to each block.

3.1.2 Threading Model. The SCONE framework uses an M:N threading model, where M application threads running in the enclave are mapped to N OS threads. The number of application or OS threads can be dynamically changed, but the allowed number of OS threads within the enclave is typically limited by the number of available CPU cores to allow the enclave to utilize all cores. The framework implements a user-level scheduler that controls the mapping of OS to enclave threads, allowing it to save enclave transitions for operations that require waiting. The user-level scheduler does not support preemption, which is not a problem in the examples considered, since all threads execute system calls or synchronization routines. In addition, the SCONE kernel module spawns multiple OS threads to execute the system call requests of the asynchronous system call mechanism described in Section 3.1.3.

3.1.3 Asynchronous System Calls Interface. As explained in Section 2.1, it is not possible to make system calls from the enclave

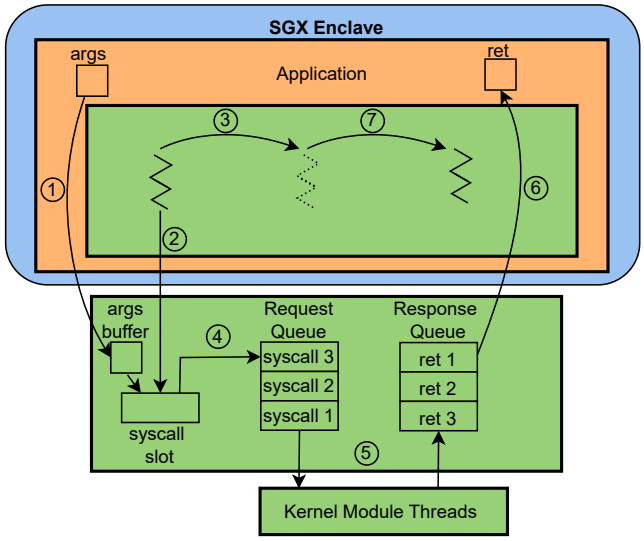

Figure 3: Asynchronous System Call Interface

context, so an enclave-external mechanism is required to execute them. To execute the system call, the arguments and their referenced memory areas must be copied out and the results must be copied back into the enclave. Since enclave transitions are slow, the SCONE provides a *asynchronous system call interface* that uses two lock-free, multi-producer, multi-consumer queues [2], the request queue and the response queue, to handle the application's system calls. The calls are then executed using the OS threads of the SCONE kernel module, as explained in Section 3.1.2.

The procedure for executing a system call is shown in Figure 3. The application's enclave thread copies the arguments and their referenced memory ①, if any, to a buffer outside the enclave. The thread then adds the description of the system call to a *syscall_slot* data structure ② using the thread's local storage. The application thread then yields to the user-level enclave scheduler ③, which runs another application thread until the system call is complete. The system call is issued by placing the reference to its *syscall_slot* in the request queue ④. One of the kernel module threads dequeues the request ⑤, executes the system call, and then places the result in the response queue. The result is then copied from the result memory buffer into the enclave ⑥ and validated by the shield. The application thread is rescheduled ⑦.

3.1.4 Docker Integration. Integration of the *secured container* with Docker is accomplished by wrapping the standard Docker client with a SCONE client. Creating a *secured container* image requires a trusted environment where the SCONE client initializes the necessary metadata for the file system shield to ensure the confidentiality and integrity of the files contained in the image. This metadata is stored in a *filesystem protection file*, which is then signed by the author of the image to protect its integrity. To start the container, each *secure container* needs a Startup Configuration File (SCF) containing the encryption keys of the I/O streams and the *file system protection file* A hash of the *file system protection file* is included to verify its integrity. Since SGX does not protect the confidentiality of the enclave code, the SCF is passed over a secure TLS connection at startup, after validating the correct setup of the container using the attestation mechanism provided by SGX.

3.1.5 Limitations. The SCONE framework has some usability and performance limitations. First, it does not support all possible system calls; for example, the *fork* system call [21], although support for creating new threads is possible with newer versions of SGX that allow dynamic thread creation [28]. It only supports creating application threads with *pthread_create* [22]. Full cloning of a process and its enclave is also not possible. An unmodified application binary will not work with the SCONE framework, as it must first be statically linked against the SCONE library. It also does not support shared libraries to ensure that all code is verified by the enclave during its initialization. In addition, the user-level scheduler requires constant execution of system calls, which may exclude certain applications from use in the enclave context.

3.2 Gramine-SGX

Gramine-SGX [26] runs unmodified applications on SGX by including the Gramine library OS. [25] in the enclave. Figure 4 shows the design of the framework. The libOS provides the application with its expected runtime environment by including the glibc C library and supports dynamic loading and linking, allowing it to run unmodified binaries in the enclave. The C library then interfaces with the Gramine library OS, which implements the standard system call API. The libOS implements features of OS, such as memory segmentation, to reduce the number of enclave transitions. To interface with the outside of the enclave, it uses a custom Gramine Application Binary Interface (ABI) [26]. The ABI calls are received by a Platform Adaption Layer (PAL) which translates the enclave's Gramine ABI calls into a more restricted set of system calls on the host OS. This section first summarizes the features of the OS gramine library, then describes the customizations needed to implement it inside the enclave, and finally explains the shielding of the external interface.

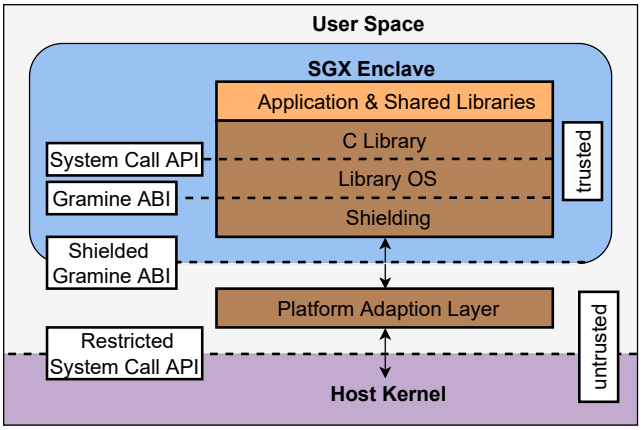

Figure 4: Gramine SGX Design

3.2.1 Gramine Library OS. The main idea behind libOS is to provide OS virtualization to the application by implementing the OS

interface as an application library. This can improve the efficiency of virtualization by extracting the functional layer below the system call interface into the memory mapping of the application. The Gramine library OS supports multi-process applications, where multiple instances of the OS jointly implement abstractions of the Portable Operating System Interface (POSIX), but appear as a single OS. To do this, the instances use message passing over byte streams that connect the picoprocesses, on which it implements a Remote Procedure Call (RPC) interface for coordination.

The application and library OS are isolated from the rest of the system by the *Trusted Reference Monitor* and run in a *picoprocess*, explained in Section 2.3, The main resource required by the monitor is the application specific *manifest* file, which contains the allowed file system view and network restrictions, allowing it to prevent access to unauthorized resources. It also installs a system call filter and intercepts critical system calls to ensure isolation of the application, thereby also protecting the host OS from the library OS instances.

To interface with the outside world, Gramine uses a simple generic ABI adapted from Drawbridge [19]. Drawbridge is a library OS similar to Gramine that provides application-compatible OS virtualization and isolation on Windows. In this context, Drawbridge provides a small set of ABI calls sufficient to interface with the host kernel. Gramine requires a lightweight intermediary layer, the PAL, outside the *picoprocess* to translate the Gramine ABI calls into a restricted set of system calls to the host kernel.

3.2.2 Gramine Adaption for SGX. To ensure the integrity of the files used and to protect the application from the host file system, the application's *manifest* is extended to include hashes of the trusted files for verification. The manifest is used to initialize the enclave and thus also contains initialization parameters of the SGX enclave. The *manifest* file itself is also signed to protect its integrity.

The untrusted PAL uses the *manifest* file to initialize the enclave by invoking the SGX driver, and ensures correctness by using local attestation. The untrustworthiness of the PAL requires the addition of a shield to protect the application and libOS from potentially malicious input from the PAL.

The framework supports multi-process applications by combining multiple enclaves into an enclave group. To allow coordination within the group, separate instances of libOS communicate via byte streams secured with TLS and authenticate themselves using the local attestation mechanism provided by SGX.

3.2.3 Interface Shielding. To shield the application from the untrusted PAL and the host OS, the framework implements three types of shields.

1) Loader shield: The framework allows dynamic loading and linking, so it must include a trusted dynamic loader in the enclave. After the enclave has been set up correctly by the PAL, the trusted bootloader loads the libraries and files specified in the manifest and ensures their correctness by performing a cryptographic measurement and verifying it against the provided hash of the *manifest*. To verify the correctness of a library, the framework must generate a unique signature for each combination of supported library versions with the application. Therefore, it only loads libraries whose hash it recognizes, which contains the problem of loading corrupt libraries with known Common Vulnerabilities and Exposures (CVE) that appear after the *manifest* is created.

2) Interface shield: Gramine-SGX defines a custom ABI to interface with the untrusted environment, since the system call interface was not designed with a model of mutual distrust between the OS and the application, as evidenced by its vulnerability to *Iago attacks*. [6]. The Gramine ABI has a more limited interface, which eases the challenge of distinguishing expected from malicious behavior when validating the response from the untrusted environment. The ABI interface is divided into three categories based on the ease of validation into *safe*, *benign* or *unsafe* ABI calls. Most ABI calls are characterized as *safe* and are easy to validate. For *benign* calls, the shield can check if the response violates the specification and either reject it or let the libOS handle the violation. Only two calls are categorized as *unsafe*, which means that the validity of their responses cannot be checked. The framework maps the null page into the enclave to protect the application from errors caused by null pointer references, which could otherwise result in a page fault that would allow the OS to map it and add malicious data to the enclave.

3) Communication shield: To implement multi-process abstractions within an enclave group, the framework requires secure message passing, since enclaves are not allowed to share protected memory. The framework therefore wraps the byte stream used for the RPC interface in a TLS connection that authenticates the other enclaves using the attestation mechanism. This allows it to implement process creation via the *fork* system call, where an enclave initializes a new child enclave and then sends its state over the secured stream.

3.2.4 Limitations. The main limitations are that only a certain set of operations are supported in the enclave, since the framework does not yet cover the full functionality provided by the system call interface with its ABI and libOS. Gramine-SGX requires a trusted dynamic loader that requires all loaded libraries, which can quickly lead to an explosion of combinations and includes the aforementioned vulnerability to compromised library versions Section 3.2.3.

4 EVALUATION OF THE FRAMEWORKS

This section compares the frameworks in terms of TCB size and performance.

4.1 General Comparison

Both frameworks allow unmodified execution of the application code, but Gramine-SGX also allows execution of the unmodified binary through its dynamic linking and loading, while SCONE requires static linking and thus recompilation of the source code. Gramine-SGX only allows a 1:1 threading model within the enclave and has no asynchronous system call interface. SCONE also implements a file system shield to allow persistent storage of enclave data. Both frameworks don't implement the full system call interface of the host OS, most notably the lack of support for the common *fork* system call in SCONE.

4.2 TCB Comparison

The TCB numbers of the reviewed frameworks are shown in Figure 5. As can be seen, the total TCB in the enclave for Gramine-SGX

is about 1,348 thousand lines of code (kloc), but about 1,292 kloc are due to the included glibc. Only 34 kloc are needed for the OS library and 22 kloc for the shield. The SCONE framework, on the other hand, only has a TCB of about 187 kloc, of which 88 are the musl libc and 99 kloc for the shield. Excluding the differences due to the included C library, Gramine-SGX has a 44 % smaller TCB than SCONE. This is mainly due to the nature of the Gramine ABI as a simpler and more limited interface than the API system call, which needs to be shielded from the untrusted environment. In addition, SCONE implements a more sophisticated interface mechanism with the asynchronous system call and a user-level scheduler.

Since previous work shows [12] that code size, in general, only affects vulnerability when there is a difference in the size of the TCB. Thus, security determined by TCB size can be considered equivalent.

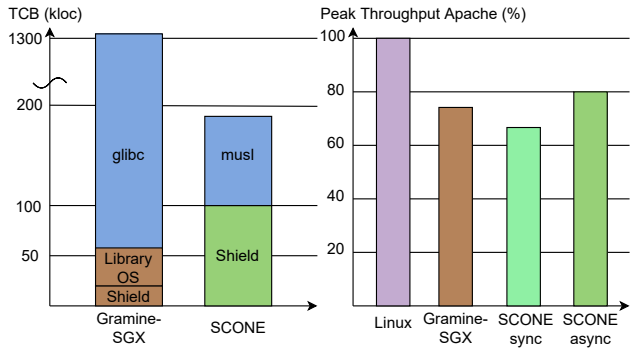

Figure 5: Framework Comparison

4.3 Performance Comparison

Using only the surveyed papers as a reference to draw a performance comparison is difficult because both papers mostly use different benchmark applications to showcase their framework. The only common measured application is the Apache web server. The multi-threaded nature of the application, the differences in multi-thread handling between the frameworks complicate the comparison. On Gramine-SGX, the Apache server runs with five threads, each occupying a separate enclave. The SCONE paper doesn't specify the number of enclave threads used in the benchmarking, it generally runs with one thread per core, the use of four threads according to the specified CPU is assumed. Both frameworks also use different hardware in their benchmarking, SCONE uses server-grade hardware such as a more powerful Intel Xeon CPU with hyper-threading and 64 GB of memory, while Gramine-SGX benchmarks run on a consumer-grade Intel Core 5 with no hyper-threading and only 8 GB of memory, Therefore, the comparison provided in Figure 5 compares the peak throughput of the server running inside the framework with the baseline of a native Linux system.

Gramine-SGX's peak throughput is about 74% of native Linux. Most of the overhead is due to the enclave transitions required by the mutual coordination of the instances. Because of this high coordination overhead, increasing the degree of parallelization by scaling the number of enclaves can only improve performance up to a limited number of enclaves, depending on the application.

Using SCONE without the asynchronous system call mechanism achieves a peak throughput of about 66% compared to the Linux baseline. The main performance penalty compared to Gramine-SGX is the avoided enclave transitions due to their higher functionality implemented in glibc and the libOS. The asynchronous system call interface improves performance by about 13% to 79% of the native Linux application, demonstrating the performance benefits of saving enclave transitions.

5 RELATED WORK

This section provides a brief overview of other approaches to running unmodified applications in an enclave and tools for automatically partitioning an application into different privilege levels that reduce, but don't eliminate, developer effort.

5.1 Running Unmodified Applications

Panoply [24] is another framework for running an unmodified application in an enclave context. Its main goal is to reduce the TCB to make it possible to formally verify the correctness of trusted code in the enclave. To do this, it provides the POSIX API as its external interface, allowing it to implement only a very thin intermediate layer in the enclave, shielding the application and eliminating the need to include a C library. Haven [4] takes a similar approach to Gramine-SGX by using the Drawbridge library OS built for Windows to allow legacy applications to run in the enclave. Compared to Gramine-SGX, the Haven library OS has a significantly higher TCB of millions of lines of code, which greatly increases the attack surface and results in higher memory requirements for the enclave.

5.2 Automatic Privilege Separation

The approach used in Glamdring [16] is to place only the functionality of the application that handles sensitive data in the enclave, thus reducing the TCB. The framework then performs static dataflow analysis and static backward slicing to identify all of the functions that handle the sensitive data that the developer has to annotate. It then automatically extracts the code and generates an interface at the enclave boundary that protects and verifies the sensitive data. EnclaveDom [17] is an in-enclave privilege separation system using the Gramine libOS. It divides an enclave into multiple isolated memory areas annotated by the developer to restrict direct access to sensitive data for potentially vulnerable third-party libraries.

6 CONCLUSION

Intel SGX provides security for code running inside the enclave against a malicious environment and shields the data it handles. To extend this security benefit to legacy applications, the enclave requires a runtime environment. This paper summarizes and compares two approaches to implementing such an environment: SCONE uses the container abstraction as an intermediate layer and interfaces with the environment via a shielded system call interface. Gramine-SGX runs a library OS inside the enclave and interfaces to the environment via a custom ABI with a more restricted interface.

REFERENCES

[1] DoD 5200.28-STD. 1985. *Trusted Computer System Evaluation Criteria*. DoD Computer Security Center.

[2] Sergei Arnautov, Pascal Felber, Christof Fetzer, and Bohdan Trach. 2017. FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 907–916. https://doi.org/10.1109/IPDPS.2017.41

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 689–703.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (aug 2015), 26 pages. https://doi.org/10.1145/2799647

[5] Maarten Bullynck. 2019. What is an Operating System? A historical investigation (1954–1964). In *Reflections on Programming Systems. Historical and Philosophical Aspects*. https://shs.hal.science/halshs-01541602 Draft version of a contribution, initially presented at HaPoP-3 in Paris, 2016.

[6] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2451116.2451145

[7] Inter Corporation. 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3D*.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. (2016). https://eprint.iacr.org/2016/086

[9] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. 2008. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 339–354.

[10] Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen, and Dag Johansen. 2017. Performance of Trusted Computing in Cloud Infrastructures with Intel SGX. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 696–703. https://doi.org/10.5220/0006373706960703

[11] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.* 2016 (2016), 204.

[12] Bhushan Jain, Chia-Che Tsai, and Donald E. Porter. 2017. A Clairvoyant Approach to Evaluating Software (In)Security. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 62–68. https://doi.org/10.1145/3102980.3102991

[13] Michael Kerrisk. 2013. Namespaces in operation, part 1: namespaces overview. (2013). https://lwn.net/Articles/531114/#series_index ,accessed at 04.12.2022.

[14] Michael Kerrisk. 2021. CGROUPS(7) - Linux Programmer's Manual. (2021). https://man7.org/linux/man-pages/man7/cgroups.7.html ,accessed at 04.12.2022.

[15] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. 1991. Authentication in Distributed Systems: Theory and Practice. *SIGOPS Oper. Syst. Rev.* 25, 5 (sep 1991), 165–182. https://doi.org/10.1145/121133.121160

[16] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, USA, 285–298.

[17] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. 2019. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. (2019). https://doi.org/10.48550/ARXIV.1907.13245

[18] Paul Jackson Paul Menage and Christoph Lameter. 2004. Control Groups. (2004). https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html ,accessed at 04.12.2022.

[19] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[20] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 57–64. https://doi.org/10.1109/Trustcom.2015.357

[21] Scontain. 2023. Frequently Asked Questions. (2023). https://sconedocs.github.io/faq/ ,accessed at 20.01.2023.

[22] Scontain. 2023. Glossary. (2023). https://sconedocs.github.io/glossary/ ,accessed at 20.01.2023.

[23] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/1272996.1273025

[24] Dat Le Tien, Shweta Shinde, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*. https://www.microsoft.com/en-us/research/publication/panoply-low-tcb-linux-applications-with-sgx-enclaves-2/

[25] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

[26] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, USA, 645–658.

[27] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3079856.3080208

[28] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP 2016)*. Association for Computing Machinery, New York, NY, USA, Article 11, 9 pages. https://doi.org/10.1145/2948618.2954330