A Practical Intel SGX Setting for Linux Containers in the Cloud

Dave (Jing) Tian
University of Florida
daveti@ufl.edu

Joseph I. Choi
University of Florida
choijoseph007@ufl.edu

Grant Hernandez
University of Florida
grant.hernandez@ufl.edu

Patrick Traynor
University of Florida
traynor@ufl.edu

Kevin R. B. Butler
University of Florida
butler@ufl.edu

ABSTRACT

With close to native performance, Linux containers are becoming the de facto platform for cloud computing. While various solutions have been proposed to secure applications and containers in the cloud environment by leveraging Intel SGX, most cloud operators do not yet offer SGX as a service. This is likely due to a number of security, scalability, and usability concerns coming from both cloud providers and users. Cloud operators worry about the security guarantees of unofficial SDKs, limited support for remote attestation within containers, limited physical memory for the Enclave Page Cache (EPC) making it difficult to support hundreds of enclaves, and potential DoS attacks against EPC by malicious users. Meanwhile, end users need to worry about careful program partitioning to reduce the TCB and adapting legacy applications to use SGX.

We note that most of these concerns are the result of an incomplete infrastructure, from the OS to the application layer. We address these concerns with *lxcsgx*, which allows SGX applications to run inside containers while also: enabling SGX remote attestation for containerized applications, enforcing EPC memory usage control on a per-container basis, providing a general software TPM using SGX to augment legacy applications, and supporting partitioning with a GCC plugin. We then retrofit Nginx/OpenSSL and Memcached using the software TPM and SGX partitioning to defend against known and potential attacks. Thanks to the small EPC footprint of each enclave, we are able to run up to 100 containerized Memcached instances without EPC swapping. Our evaluation shows the overhead introduced by *lxcsgx* is less than 6.9% for simple SGX applications, 9.5% for Nginx/OpenSSL, and 20.9% for containerized Memcached.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Virtualization and security**; *Operating systems security*.

KEYWORDS

Cloud; Containers; Security; SGX

ACM Reference Format:
Dave (Jing) Tian, Joseph I. Choi, Grant Hernandez, Patrick Traynor, and Kevin R. B. Butler. 2019. A Practical Intel SGX Setting for Linux Containers in the Cloud. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3292006.3300030

1 INTRODUCTION

In the past few years, solutions such as Linux Containers (LXC) and Docker have provided compelling alternatives to heavyweight solutions such as virtual machine monitors running guest operating systems. Container mechanisms provide *OS-level virtualization*, where multiple isolated systems can be run under the same operating system kernel. Cloud computing providers, in particular, stand to gain from containers, as their substantially lighter use of computing resources allows far greater density of deployments per physical machine and drives down infrastructure costs. However, a significant concern with this approach is the extent to which *separation* between containers is possible. Specifically, because containers share a common OS kernel, any vulnerability that exploits the kernel would affect all other containers on the system.

Intel Software Guard Extensions (SGX) [22] provides a compelling new way to establish guarantees of trustworthy execution and platform integrity. SGX preserves the confidentiality and integrity of sensitive data in *enclaves*, secure regions of memory that are protected from unauthorized access by higher privileged processes and system software. Unfortunately, while there has been a surge of research into providing SGX-enabled security guarantees within cloud environments, including Haven [5], Graphene-SGX [59, 60], SCONE [3], and Panoply [54], these have not been adopted to-date by most cloud providers. This may be due to a number of security, scalability, and usability concerns from both cloud providers and users: cloud operators worry about the security guarantee of unofficial SDKs (current solutions that integrate SGX do not interface with the official SDK provided by Intel), limited support for remote attestation within containers, limited physical memory for Enclave Page Cache (EPC) making it difficult to support hundreds of enclaves, and potential denial-of-service attacks against EPC by malicious users; meanwhile, end users need to carefully partition SGX programs to reduce the TCB and face the challenge of rewriting legacy applications to make use of SGX. Solutions such as Haven, Graphene-SGX, and SCONE do offer convenience by removing this need to partition, but come at the cost of an increased TCB that includes all of an application's insensitive components. While Panoply places different parts of application logic into separate enclaves, the creation of and communication with multiple enclaves per application increases EPC memory consumption.

In particular, when deploying SGX in a cloud environment, we believe the following issues must be addressed:

(1) *Limited support for remote attestation:* A critically important feature of SGX is its ability to attest to the identity and integrity of SGX applications to third parties (e.g., cloud users), but neither Haven nor SCONE provides native support for CPU remote attestation.
(2) *SGX application security:* Solutions that involve placing entire applications within a secure enclave, such as Haven, Graphene-SGX, and SCONE, do not necessarily guarantee security, as they can dramatically expand the TCB and may contain vulnerabilities from within either applications or libraries, such as the Heartbleed [9] bug within OpenSSL.
(3) *Limited EPC memory:* The current maximum EPC size is 128 MB, with approximately 90 MB left for users after accounting for enclave management [28]. While EPC page swapping is supported on Linux, it leads to a considerable performance hit. A cloud operator has a vested interest in minimizing the memory footprint of enclaves, which would allow the supporting of many users and reducing of performance degradation (due to swapping) at the same time. The EPC limit also implies cloud providers need to protect the EPC from (malicious) overconsumption, a factor not considered by existing solutions. SGXv2 even allows dynamic EPC page allocation during the enclave runtime, exacerbating EPC memory consumption.[1]
(4) *Support for legacy applications:* To reduce the TCB inside enclaves, Intel [21, 23] mandates program partitioning. Unfortunately, this makes programming for SGX non-trivial.[2]

We note that most of the concerns surrounding adoption of SGX-supported containers in the cloud are the result of an incomplete infrastructure, from the OS to the application layer. We address these concerns through our development of *lxcsgx*, a platform fully enabling Intel SGX deployment for Linux Containers (LXC) in the cloud environment. Unlike past solutions, we pay particular attention to the practical deployment concerns in a cloud environment mentioned above. In so doing, our contributions include:

- Enabling SGX remote attestation for containerized applications. Compared to native host attestations, the overhead is 6.9% and 4.9% for containerized local and remote attestations.
- Enforcing EPC memory usage control per container in the Linux kernel to prevent (malicious) overuse of resources.
- Implementing a GCC plugin to assist program partitioning to reduce the TCB in the enclave and better support scalability.
- Implementing a software TPM using SGX, providing a fast hardware TPM replacement as well as socket APIs for legacy applications, which can access the TPM functionality in an attestable enclave instead of being fully refactored for SGX. The speed of TPM operations ranges from 10 to 280 μs.

[1] We provide further discussion on why SGXv2 is not a panacea in Section 6.
[2] The Intel SGX SDK Developer Manual v1.9 has 320 pages.

- Retrofitting and evaluating Nginx/OpenSSL and Memcached using SGX based on *lxcsgx*. Compared to original native applications, the overhead is less than 9.5% for Nginx/OpenSSL and 20.9% for containerized Memcached.

Outline. The remainder of this paper is structured as follows: Section 2 further motivates our work on *lxcsgx*; Section 3 describes the design and implementation of *lxcsgx*; Section 4 shows how to retrofit applications by leveraging *lxcsgx*; Section 5 evaluates the performance of *lxcsgx* and the applications built atop it. Finally we discuss takeaways from our work in Section 6, while contrasting it against the existing literature in Section 7, and conclude in Section 8.

2 MOTIVATION

Solutions allowing an entire application to run within an SGX enclave without any modifications, such as Graphene-SGX and SCONE, ease the integration of SGX with existing (legacy) applications. However, these approaches tend to bloat both TCB and enclave size, miss key features such as remote attestation, and ignore hardware constraints on EPC size (instead totally relying on EPC page swapping). We examine limitations of these solutions to further motivate our work.

2.1 Why could unofficial SDKs be problematic?

We observe that some cloud providers (e.g., IBM and Azure [48]) had SGX-capable servers available as early as 2017, but they did not officially support SGX applications until recently. We speculate that concern over unofficial SDKs was a contributing factor to the holdup.

Solutions built on customized software stacks or unofficial/home-made SDKs, such as Haven and SCONE, cannot provide native support for remote attestation.[3] Remote attestation requires Intel's Quoting Enclave (QE), which leverages Intel Enhanced Privacy ID (EPID) [32] and is part of the Intel SGX software stack. The missing remote attestation is critical, because it provides cloud users with a guarantee that the desired enclave has the right measurement and is running on a genuine Intel CPU with SGX enabled (rather than a software emulator).[4] Without such a guarantee, it is impossible to reason about the security of the SGX-supported cloud platform.[5]

Runtime libraries within the enclave impose security concerns as well. Software Grand Exposure [7] and Cachezoom [40] have shown that traditional crypto libraries inside the enclave, such as the OpenSSL used by Graphene-SGX and Panoply, are still vulnerable to cache-based side channel attacks. Intel Integrated Performance Primitives (IPP, built into the Intel SGX SDK) appears more secure due to its usage of AES-NI [30]. Similarly, putting glibc or musl[6] into the enclave naively, as Graphene-SGX and SCONE do, might still be a vulnerable practice due to the insecure functions included (e.g., strcpy). In contrast, all "dangerous" functions are removed from Intel's trusted C library, and "sensitive" functions are implemented using hardware instructions (e.g., RDRAND for rand).

[3] Haven's attestation is emulated and requires trust in the cloud provider.
[4] SCONE provides only local attestation, which does not give the latter guarantee. Haven, while not strictly a container environment, does not support remote attestation either.
[5] With regard to the recent Foreshadow [61] attack, which leverages out-of-order execution to extract a SGX-enabled machine's private attestation key and allows an adversary to forge valid attestation responses, Intel has released microcode updates [27]. As stated by Van Bulck et al. [61], Foreshadow exploits an implementation bug and does not invalidate the architectural design of Intel SGX.
[6] https://www.musl-libc.org/

Compared to other SGX software stacks, the official Intel SGX SDK seems to be the most secure open-source solution, designed for security and with defenses against Spectre attacks [34]. Azure also exclusively supports the Intel SGX SDK in its cloud environment [48]. While other libraries may provide alternative means for attestation and hardening against attacks, reliance on them consequently alters the SGX trust model. In the remainder of the paper, we assume the official Intel SGX SDK to be deployed in the cloud environment.

2.2 Why is program partitioning preferred?

Program partitioning requires application developers to figure out the most security-sensitive parts of the code, and transform them to use SGX. Though cumbersome, this methodology may be the best security practice to reduce the attack surface via reducing the TCB in the enclave. Because syscalls are not allowed inside the enclave, any SGX solution that does not require program partitioning instead relies on an additional middle layer (e.g., LibOS) to emulate these syscalls; this practice might bloat the TCB depending on the coverage of the emulation. Furthermore, as explicitly mentioned in the SGX Developer Manual [24]), putting vulnerable code into enclaves does not suddenly make the code secure.

The other benefit of program partitioning comes from the potentially small EPC memory consumption in both loading time and runtime. The binary size of Drawbridge LibOS used by Haven is over 200 MB, which is even beyond the maximum 128 MB EPC memory limitation. While TCB size does not directly determine the memory consumption, they are related. For example, a partitioned OpenSSL library in Panoply takes around 6 MB of EPC memory, whereas an unmodified library takes 65 MB in Graphene. To assist with program partitioning, *lxcsgx* contains a GCC plugin *gccsgx*, which supports security level tagging in the source file and lightweight tainting analysis based on the tagging.

2.3 Why is EPC memory control important?

Supporting many users with only 128 MB EPC memory on a single server imposes fundamental challenges to cloud providers. A simple memory leakage bug in SGX applications can exhaust the limited EPC resource and cause the SGX kernel driver [25] to swap out enclaves of other users. Even worse, a malicious user could launch DoS attacks against the EPC memory or conceal cache attacks in the enclave [52]. The result of these attacks are performance degradations [45] and security breaches. The situation gets worse for KVM SGX [26], Intel's SGX virtualization on KVM solution. KVM SGX does not support EPC oversubscription, meaning a VM cannot be created if the virtual EPC requested is beyond the physical EPC limit. Unlike any existing SGX solutions, *lxcsgx* recognizes the importance of EPC memory protection, and enforces EPC memory usage control per container.

2.4 Why is a software TPM crucial?

Programming SGX applications using the Intel SGX SDK is not easy. It requires application developers to have a deep understanding of security concerns specific to the application, as well as of the SDK APIs. Moreover, as shown in later sections, even a simple enclave implementation may take 1 MB of EPC memory. This means a single server can support no more than 100 users at the same time.[7] As a result, EPC page swapping will eventually happen when a new user needs to create an enclave, impacting performance and security by introducing page faults. Unfortunately, unlike a typical shared library such as glibc, an enclave cannot be shared by different processes to reduce EPC memory consumption. Each process needs to allocate a new virtual address region to load the same enclave, which maps into different EPC pages. By design, EPC pages are not shared.

[7]Recall that the actual EPC memory left for users is around 90 MB.

Haven	Graphene	SCONE	Panoply	*tpmsgx*
209	64.7	>4.0	5.9	1.1

Table 1: Enclave size (MB) for Nginx/OpenSSL in different SGX solutions.

We observe that the desired SGX functionalities are usually shared among a number of applications; these include crypto operations, random number generation, and secure storage. Therefore, it is possible to have this general platform service create a single enclave that serves many different applications at the same time. This cloud service can provide user-friendly APIs, and reduce the EPC memory consumption by avoiding user enclave creation. We instantiate this service as a software TPM[8] using SGX (*tpmsgx*). As a core component of *lxcsgx*, it provides common crypto implementations based on the Intel SGX SDK, and a typical socket API for application developers. As we will later demonstrate, we transform Nginx/OpenSSL to use *tpmsgx* for crypto operations during the SSL/TLS handshake. Table 1 shows how much *tpmsgx* helps to reduce the EPC memory consumption by reducing the enclave size, compared to other SGX solutions.[9]

We summarize and compare the various features of existing SGX solutions and *lxcsgx* in Table 2. We also separately list *tpmsgx* in the table, because it can be used independently of the other components of *lxcsgx*. We believe *lxcsgx* provides an SGX solution that considers practical deployment issues for containers in a cloud environment.

3 DESIGN AND IMPLEMENTATION

Intel SGX provides a means to improve the security of applications via runtime integrity and confidentiality. We investigate how to properly intertwine Linux containers and SGX in a cloud environment through our *lxcsgx* architecture, shown in Figure 1. We choose to focus on LXC, but *lxcsgx* can be extended to support Docker as well.[10] Although the components of *lxcsgx* may appear to be loosely coupled, they share a unified goal and work together under a common platform infrastructure to facilitate SGX use within cloud environments. We fully describe the design and implementation of each component in this section; we also discuss the considerations made for balancing practical architectural limitations and the programming paradigm of SGX with respect to security, scalability, and usability.

3.1 Threat Model and Trust Model

We consider a cooperative cloud environment, where each server supports hundreds of Linux containers. This number is reasonable [33] for deployment due to the lightweight nature of containers compared to VMs, and is particularly apt for microservice-based environments.

[8] While simply plugging in a TPM does not necessarily make a legacy application secure, we hope the familiarity of a TPM, along with the provided software APIs for interfacing with it, will ease the process of supporting legacy applications.
[9] Please note that the number for SCONE is conservative, since OpenSSL is not included.
[10] Docker is descended from LXC and, while different, shares many of the same principles.

Solution	Container Support	Remote Attestation	EPC Control	TCB (LoC)	Enclave Size (MB)	Software Stack	Overhead	FOSS
Haven [5]	N/A	N	N	>1.0M	209	Simulation	<54%	N
Graphene-SGX [59, 60]	N/A	Y	N	1.3M	58.5+App	Custom	50% (avg)	Y
SCONE [3]	Docker	N	N	>187K	2.5+App	Custom	<40%	N
Panoply [54]	N/A	Y	N	140K	PartitionDep	Custom	24% (avg)	Y
lxcsgx	LXC	Y	Y	119K	PartitionDep	Intel	<20%	Y
tpmsgx	LXC	Y	N	2K	1.1	Intel	<10%	Y

Table 2: Comparison among existing SGX solutions versus *lxcsgx* and *tpmsgx*

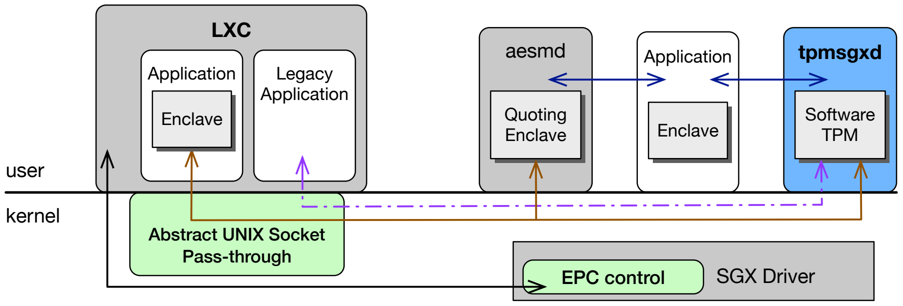


Figure 1: *lxcsgx*'s design enables containerized applications to communicate out from LXC via an abstract UNIX socket. This gives applications within a container access to Intel's *aesmd* and our software TPM (*tpmsgxd*), all while the SGX driver monitors each LXC container's EPC usage. *tpmsgxd* is also available to applications outside a container.

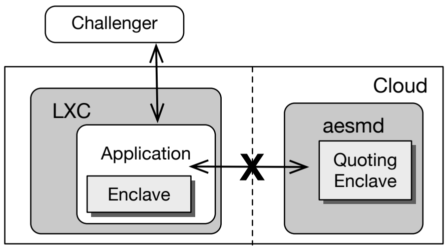


Figure 2: Where there was previously no path between an application in LXC and *aesmd*, our abstract UNIX socket pass-through enables this path and thus attestation.

We expect that the cloud service provider attempts to uphold its contract with customers (e.g., through timely system patching to fix bugs and isolating containers using kernel features). However, we do not necessarily trust the cloud provider, which may be interested in breaking the confidentiality of hosted containers unbeknownst to its customers. Malicious cloud providers may also actively try to compromise the confidentiality and integrity of hosted containers.

The TCB of *lxcsgx* comprises SGX-enabled CPUs and code/data loaded into enclaves. Neither the Linux kernel nor the LXC programs running on the cloud server are trusted, although we expect them to provide certain basic functionality (e.g., starting the system and containers). We do not consider DoS attacks launched by ring-0 attackers (e.g., to prevent users from using Intel SGX). Additionally, we do not consider controlled-channel attacks from ring-0 attackers or side-channel attacks from ring-3 attackers. These attacks [7, 52, 62, 65] are orthogonal to the problem *lxcsgx* is trying to solve and have been well considered in the literature [6, 14, 31, 38, 53, 63].

3.2 Remote Attestation for LXC Applications

When challenger and attester applications are both running within the same container, local (intra-platform) attestation may be achieved by the two applications communicating with each other and exchanging the enclave measurement [2]. However, when the challenger is running in a different container or on a different physical machine, remote (inter-platform) attestation is needed, where the remote challenger is provided a proof, or *quote*, of the desired enclave. Getting a quote requires the attester to communicate with the Quoting Enclave (QE) [2, 32], provided by Intel SGX SDK as daemon process *aesmd* running on the native machine. Due to the use of an abstract UNIX socket by *aesmd*, and the inability of LXC/Docker to mount a non file-backend socket, remote attestation for containerized applications does not simply work out-of-the-box, as shown in Figure 2.

The simplest solution would be to make the network namespace of the container the same as that of the native host (e.g., by bridging the container's network interface card (NIC) to one in the host machine). However, this configuration breaks network isolation between containers and the host, meaning it cannot be used in a cloud environment. Another potential solution would be to run *aesmd* inside containers. Unfortunately, this does not work either because the SGX kernel driver cannot be installed inside containers. Furthermore, the driver only supports one *aesmd*/QE given a platform. Even in the absence of these limitations, cloud providers might not wish to duplicate all the SGX platform services per container, which both runs up against disk quotas and wastes EPC memory.

Linux kernel implementation. While it is easier to modify the Intel SGX SDK directly,[11] we add a new feature to the Linux kernel to support abstract UNIX socket pass-through for Linux containers.[12] We believe this to be a missing feature for both the Linux kernel and LXC. We modify the *connect()* syscall for UNIX sockets. We add a new directory under */proc* called *lxcsgx*, and add an entry *sgx_sock*, which accepts inputs from user space (e.g., the container process) specifying the abstract UNIX socket to be passed through. When applications write into */proc/lxcsgx/sgx_sock*, the kernel retrieves the PID and network namespace of the application, along with the abstract socket name, storing these together in the kernel space as

[11] Intel actually did this in the recent versions, downgrading the abstract UNIX socket to a traditional one, whose pass-through is supported both by the Linux kernel and LXC.
[12] Open-source kernel changes can be verified by the community and Linux maintainers.

a record indexed by network namespace. For a connection request using an abstract UNIX socket, the original network namespace checks will reject the request unless both source and destination sockets share a namespace. We extend these checks for abstract socket pass-through by looking to see if a pass-through record exists with the source and destination abstract sockets.

LXC implementation. To enable LXC to use the abstract UNIX socket pass-through feature provided by the kernel, we add a new configuration to LXC named *lxcsgx.sgx.sock*. Requests are passed to the kernel using */proc/lxcsgx/sgx_sock*. For example, to support remote attestation, setting "*lxcsgx.sgx.sock = sgx_aesm_socket_base*" is sufficient to let the kernel pass the connection request from the container to *aesmd* outside the container. After mounting the SGX driver using *lxc.mount.entry*, we are able to support remote attestation of SGX applications running inside containers.

3.3 Controlling EPC Memory Usage

To prevent EPC memory from being (maliciously) exhausted by certain users or containers, we count the number of EPC pages allocated per container, rejecting further allocation requests if the EPC quota of the requesting container is exceeded. However, finding the corresponding container that is responsible for each EPC allocation request is non-trivial, because containers are transparent to the underlying Linux kernel, which only sees processes. One possible solution is to trace the Parent PID (PPID) all the way back to the container process (i.e., *lxc-start*) if the given process belongs to a container. Unfortunately, this method has $O(n)$ complexity, and does not work if the process is not directly forked by the container process (e.g., if running applications using *lxc-attach*). Instead, we use the network namespace as a unique identifier for containers, since it is shared by all applications running inside the container. In most cases, containers are configured with different virtual NICs (`veth` in LXC), and thus have different network namespaces. When one network namespace is shared by multiple containers, the EPC control will be applied to every container within the namespace.

SGX kernel driver implementation. The SGX kernel driver is responsible for EPC memory management, including allocation, swapping, and reclaiming. To access EPC control from user space, we add another two entries under the */proc/lxcsgx* directory, named *epc_control* and *epc_limit*. The former is used to enable/disable EPC control globally on the machine, while the latter is used by containers to pass the EPC control information to the kernel. Each EPC control record saved in the kernel contains network namespace, PID of the record creator, EPC usage limit (number of 4K pages), a flag to activate/deactivate this record, and the current usage of EPC memory of the container. Upon each attempted EPC page allocation (EADD), we find the PID of the requesting process using the enclave owner information maintained by the SGX driver. Given the PID, we find the corresponding network namespace and retrieve the EPC control record. If the record is activated and the requested EPC usage is within the limit, allocation is permitted and usage count increased. Similarly, for EPC page deallocation, we reduce the current usage count of the corresponding EPC control record.

LXC implementation. To leverage the EPC control mechanism, another two new configurations (*lxc.sgx.epc.limit* and *lxc.sgx.epc.control*) are added into LXC. For example, "*lxc.sgx.epc.limit = 1000*" is used to set the maximum EPC memory usage to be 1000 pages for the container, while "*lxc.sgx.epc.control = 1*" is used to activate the EPC control for this container. The container writes into */proc/lxcsgx/epc_limit* to add the EPC record into the kernel during startup. System administrators may also apply commands to the */proc* entries to modify EPC control records in the kernel as needed.

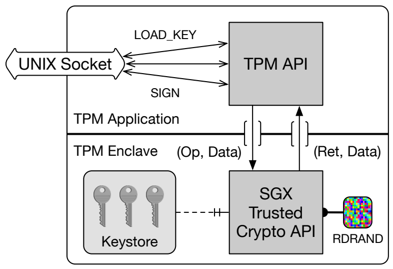


Figure 3: The architecture of *tpmsgxd*. The TPM APIs are exposed via a UNIX socket. The whole software TPM implementation is self-contained, running inside the enclave.

3.4 *tpmsgx*: A Software TPM Using SGX

To reduce the learning curve of SGX programming and free the users from creating their own enclaves, we design a software TPM using SGX, *tpmsgx*, as a general platform service providing a socket API for applications not written with SGX in mind. The whole design of *tpmsgx* is grounded in the functionality and security features provided by TPM; we focus on application-facing functionality and not on additional features (such as measured boot and system attestation) that are built upon TPM. We summarize the differences among (hardware) TPM, fTPM [47],[13] and *tpmsgx* in Table 3. Unlike low-speed TPM chips with fixed firmware installed, *tpmsgx* enjoys both CPU speed and flexible implementations, with the SGX-enabled CPU becoming the hardware root of trust. This also means the security of *tpmsgx* is heavily dependent on the code in the enclave.[14] We build upon Intel IPP within the SDK to provide common TPM functionality in *tpmsgx*, including random number generation, hashing, symmetric/asymmetric crypto primitives, and secure storage.

An issue with using SGX as a TPM is the lack of persistent storage. The data (keys) saved in the enclave will be lost after a reboot. This can be solved with CPU sealing, which encrypts data to the disk using a sealing key generated by the EGETKEY instruction based on different sealing policies [2]. For instance, the sealing can bind to the measurement of the enclave, so only the enclave with the same measurement can unseal the data (similarly to TPM sealing using PCRs). Compared to TPM-based attestation, *tpmsgx* also supports SGX attestation, which not only provides a trusted measurement of the implementation to the challenger, but also establishes a secure channel with the remote party thanks to the key exchange which occurs during the remote attestation procedure. Since the cloud provider is not trusted in our threat model, we expect *tpmsgx* to be auditable (e.g., open source), and customers can use SGX attestation to retrieve the measurement of the *tpmsgx* running within the cloud environment and verify it is as expected, thus establishing trust.

[13] An alternative TPM implementation built atop ARM TrustZone.

[14] Again, putting the entire OpenSSL/GnuTLS inside an enclave to use its functionality for *tpmsgx* might be a bad idea, due to their large attacking surfaces [15, 16].

Solution	Trust Anchor	Implementation	CPU Mode	Software Stack	Secure Storage	RT Integrity	Secure Channel	Speed*
TPM [58]	TPM	Hardware	Ring-0	TSS	NVM	N/A	N/A	10-400ms [57]
fTPM [47]	TrustZone	Firmware	Secure World	OP-TEE	eMMC RPMB	N/A	N/A	10-200ms
tpmsgx	SGX	Software	Ring-3	Intel SGX SDK	Sealing	Y	Y	15-280μs

Table 3: Comparison between *tpmsgx* and other TPM technologies. Note that the speed column provides a timing range of different TPM commands, such as RNG, key generation, quote, etc.

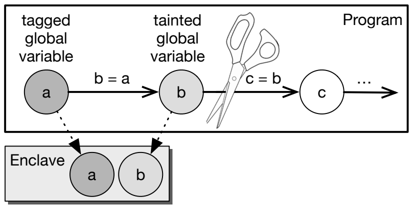


Figure 4: Given a set of tagged global variables (e.g., a), *gccsgx* identifies intermediate tainted variables (e.g., b). These global variables (e.g., a and b) constitute a minimal TCB for the enclave. The same process applies to tagged functions.

Software TPM implementation. *tpmsgx* is implemented as a daemon process *tpmsgxd* running on the native host (similarly to *aesmd*). It uses an abstract UNIX socket to receive requests from either native applications or containerized ones. Figure 3 summarizes its interface and components. Applications use the socket API to access TPM functionalities inside *tpmsgxd*. When remote attestation is enabled, an AES128 shared secret is established between the *tpmsgx* enclave and the requesting application. All responses from *tpmsgxd* are encrypted using AES128-GCM before leaving the enclave. *tpmsgx* provides the operations listed below:

- Random number generation using RDRAND.
- Loading of AES128 symmetric or ECC256 private keys.
- ECC256 key generation.
- AES128-GCM encryption/decryption.
- DH key exchange.
- SHA256 hashing.[15]
- ECDSA signing and verification.

LXC implementation. For applications running inside containers to use *tpmsgx*, we add another new configuration into LXC, named *lxc.sgx.tpm.sock*. Similarly to *lxc.sgx.sock*, "*lxc.sgx. tpm.sock = tpmsgxd_sock_base*" enables communication between the containerized applications and the *tpmsgxd*, with the help of the abstract UNIX socket pass-through feature in the kernel.

3.5 *gccsgx*: A SGX Program Partitioning Helper

Retrofitting legacy applications using SGX is challenging. Developers need to find and place security-sensitive parts of their program into an enclave while minimizing TCB, making it as small as possible to reduce the attack surface. We design *gccsgx* based on GCC to help facilitate SGX program partitioning by finding a minimal TCB for the enclave using static tainting analysis. We first add a new attribute in GCC – *SGX_ENCLAVE()*, allowing developers to tag global variables or functions considered security-sensitive within the application. We design the attribute to model multilevel security (MLS) labels supporting different security levels, such as top-secret or confidential, for two purposes. Given the EPC quota in a cloud environment, developers can decide whether or not to include code or data with lower security levels based on the resulting enclave binary size and runtime memory consumption. The tainting analysis may also taint the code or data with a different security level from what it was tagged as. By keeping the tagged level unchanged and providing a tainting level at the same time, we are able to reveal intrinsic connections within the program and catch potential mistakes made in security analysis for program partitioning.

To find a minimal TCB for the enclave, we consider global variables and functions which are directly tagged and intermediately tainted, as shown in Figure 4. This is a tradeoff between an actual minimal TCB, which only includes tagged components, and a complete minimal TCB, which incorporates everything along the tainting paths.[16] We do not consider implicit flows in *gccsgx*.

Glamdring [35] may be used instead if a complete minimal TCB is desired. To use Glamdring, developers first annotate input and output variables in the source code that contain sensitive data. Starting with the annotated inputs, Glamdring performs static dataflow analysis to track the propagation of sensitive data through the application. Using these annotations, Glamdring performs static backward slicing to identify functions that annotated outputs depend on. All security-sensitive functions identified by the above two processes are placed inside the enclave. Glamdring is compatible with our *lxcsgx* architecture, but we choose *gccsgx*; by not fully propagating taint, *gccsgx* produces a smaller TCB than approaches targeting completeness.

We assume developers would find most obvious places to tag after a security analysis, and *gccsgx* extends the minimal TCB with minimal taint propagation. Note that once this security-sensitive data/code is inside the enclave, the original tainting path will be broken. For instance, variable *c* in Figure 4 cannot get the value of *b* until an ECall is explicitly defined to grant access permission. We leave the choice to developers to determine if an ECall should be made or that part of code/data should be inside the enclave as well.[17]

Implementation. We implement *gccsgx* as a GCC 4.8.4 plugin using the GCC Python Plugin interface [37]. We implement three security levels for the *SGX_ENCLAVE()* attribute: "top-secret", "secret", and "confidential."[18] To handle memory aliasing, we require the "-O" optimization during compilation, as *gccsgx* hooks right after the "alias" GCC pass, which performs structural alias analysis, points-to and escape analysis, and flow-sensitive/insensitive analysis [18]. Example output from *gccsgx* is provided below. For example, tainting analysis may raise the tagged security level, such as *var1*, or find missing components during initial tagging, such as *var3* and *fun3*.

[15]Even though SHA256 is not a keyed operation, SHA256 is often used in security-sensitive contexts (e.g., hashing of raw data by the SGX SDK to produce signatures).

[16] It is possible that a large part of an application needs to be put inside the enclave as a result of high taint propagation.

[17] *gccsgx* only identifies the minimal TCB and does not perform enclave code generation, so we must ultimately rely on the developer to make decisions like these.

[18]Note that these labels are only used by tainting, rather than for policy control or privilege separation.

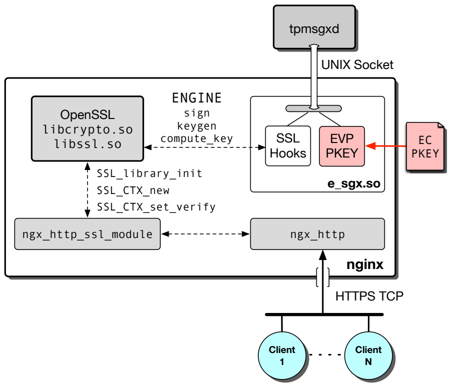


Figure 5: Relationship of *tpmsgxd*, *sgxengine* (`e_sgx.so`), and Nginx. HTTPS connections are routed through Nginx which communicates with OpenSSL to establish a TLS session.

```
********************
Global vars:
var1[test.c:14]: tagged["confidential"], tainted["secret"]
var2[test.c:25]: tagged["secret"], tainted["N/A"]
var3[test.c:17]: tagged["N/A"], tainted["confidential"]
--------------------
Functions:
fun1[test1.c:9]: tagged["secret"], tainted["N/A"]
fun2[test1.c:10]: tagged["confidential"], tainted["N/A"]
fun3[test1.c:33]: tagged["N/A"], tainted["secret"]
********************
```

4 APPLICATION CASE STUDY

We now consider how *lxcsgx* can support containerized applications. We examine Nginx/OpenSSL (a web server providing HTTPS services) and Memcached (a key-value in-memory database). Each application leverages different components of *lxcsgx*, based on different threat models and tradeoffs. Both run inside LXC containers.

4.1 Hardening Nginx/OpenSSL

Nginx [55] is a popular open-source web server that touts high performance and scalability. It embeds the OpenSSL [44] library to handle all low-level details for the web server, such as crypto operations. In order to provide authenticity to remote users via HTTPS, the web server requires a server certificate and private key to be generated. This certificate is signed by a Certificate Authority (CA) in order to create a chain of trust from a well-known issuer to the website. The server's private key is an important component in running HTTPS. This key is loaded upon Nginx boot from the file system and passed into the OpenSSL library for signing. During an SSL/TLS handshake, the private key is used to sign data, and the client verifies the signature using the server's public key (available in the server certificate). The security of the private key is essential; if compromised, an attacker can impersonate the web server.

A major bug found in OpenSSL in 2014 called Heartbleed [67] allowed remote attackers to leak random ranges of server memory, some of which were discovered to contain cookies, HTTP request bodies, passwords, and server private keys. Unless a web server was using a physical TPM for all TLS signing operations, the private key would have to be in memory at some point during Nginx's lifetime. For most cloud users running on remote hosting, placing private keys in a physical TPM may be difficult or impossible to coordinate with their cloud providers. The low speed of TPM also interferes with the scalability of Nginx [41]. In response, software-only solutions have attempted to partition server private keys away from the Nginx process and OpenSSL library [56]. Unfortunately, these solutions merely place the private key in a different process/service on the same machine, while requiring changes to Nginx and OpenSSL.

This is a typical use case of *tpmsgx*, where legacy applications need a trusted execution environment for crypto operations, and yet developers do not need to accommodate a new programming paradigm. Leveraging *tpmsgx*, we develop an OpenSSL `ENGINE` called *sgxengine*, a pluggable module that allows hooking of key cryptographic operations. This `ENGINE` talks to *tpmsgxd*, transferring and translating function calls dealing with the private key into the TPM enclave, as shown in Figure 5. With all functionality pushed into the SGX-based TPM, the ***private key never has to be loaded into Nginx/OpenSSL memory***. As we show in later sections, our solution achieves security and scalability at the same time.

Implementation. OpenSSL started to support the `ENGINE` API from version 0.9.6 onwards. Thus we choose OpenSSL 1.1.0c as the testing base. For Nginx, we choose 1.11.9, and slightly patch it to force the usage of our OpenSSL `ENGINE`. Note that we do not modify OpenSSL itself. In the *sgxengine*, we set the default private key loading function of OpenSSL to the one provided by *tpmsgx*. We then do a one-time load of the private key into the TPM and receive a key-id back for future reference. To guarantee that this key only stays inside the enclave memory, we fake the `PKEY` structure that is returned to OpenSSL so that we don't break the `ENGINE` API.

We configure Nginx to use the strongest cipher suite, ECDHE-ECDSA-AES128-GCM-SHA256, also known as NSA Suite B [49] combination 1, supporting TLSv1.2 connections. The *sgxengine* is essentially a hooked implementation of `EC_KEY_METHOD` in OpenSSL. This covers key generation, shared key computation, and signing, which are mapped into EC key generation, ECDH shared secret computation, and ECDSA signing in *tpmsgx*. During TLS session key generation, the real public key and a fake private key are returned to OpenSSL, while the real private key stays only inside the enclave memory. All future TLS operations that would normally use the private key in OpenSSL are now routed to and handled by *tpmsgx*.

One `ENGINE` API limitation we discovered was how the design of signing in OpenSSL differs from that in Intel SGX SDK. The signing function in OpenSSL expects the message digest, while the SGX SDK receives the raw data and performs hashing internally. We opt to not change any source code within OpenSSL or Intel SGX SDK. Therefore, we hook the SHA256 digest functions in the *sgxengine* to record the raw data of digest for future transfer to *tpmsgx*.

4.2 Securing Memcached Credentials

Memcached [17] is a key-value store used to alleviate the load from web databases by first attempting to handle queries from RAM before forwarding them to the database. While Graphene-SGX [60] and Glamdring [35] have tried to secure the database by putting the whole application or a portion of it into the enclave, we consider user credential protection as an alternative. Starting from 2014 [19], Memcached supports the Simple Authentication and Security Layer (SASL) [42] which requires clients to supply a username and password before a connection. During authentication, the supplied credentials are checked against an internal SASL password database containing plaintext entries, which are loaded from disk into memory.

Unfortunately, due to remote code execution vulnerabilities [10–12], attackers could not only change the data saved in the memory, but also steal user credentials. Ironically, one of the vulnerabilities lies in the SASL authentication [12]. Ideally, SASL should be used only as a way to provide an authentication protocol for receiving the input from the client and sending back the authentication result. Even if the SASL layer is compromised, the sensitive information saved in Memcached should not be leaked to its third-party libraries. We achieve this security goal by retrofitting the authentication part of Memcached using SGX. We find the security-sensitive code and data with the help of *gccsgx*, and put them into the enclave, preventing credential leakage to the untrusted code.

Implementation. As the first step in security analysis, We first pinpoint the global variable *memcached_sasl_pwdb*, which refers to the database holding credentials. We then tag this variable as an enclave candidate with security level "top-secret", and enable *gccsgx* for GCC in the Makefile. The output of *gccsgx* is shown below:

```
********************
Global vars:
memcached_sasl_pwdb[sasl_defs.c:37]: tagged["top-secret"], tainted
      ["N/A"]
--------------------
Functions:
init_sasl[sasl_defs.c:170]: tagged["N/A"], tainted["top-secret"]
sasl_log[sasl_defs.c:122]: tagged["N/A"], tainted["N/A"]
sasl_server_userdb_checkpass[sasl_defs.c:40]: tagged["N/A"],
      tainted["top-secret"]
********************
```

Even though we have not tagged any functions, there are two functions tainted by the tagged global variable, these being *init_sasl* and *sasl_server_userdb_checkpass*. We manually verified that this result of *gccsgx* is correct (e.g., no memory aliasing for the tagged global variable). The tainted functions, together with the global variable, need to be placed in the enclave. We also provide an interface for *sasl_server_userdb_checkpass* function in EDL, making this function accessible to the untrusted part of the program as an ECall. We will later demonstrate the running of 100 containers with SASL-enabled Memcached at the same time without introducing EPC page swapping, thanks to the limited TCB and EPC memory consumption as the result of program partitioning.

5 EVALUATION

All evaluation is done on a machine with a 4-core SGX-enabled CPU and 8 GB memory. The machine is running Ubuntu 14.04 LTS, with Linux kernel 4.2.8, LXC 2.0.0, GCC 4.8.4, Intel SGX SDK 1.7, and Intel SGX driver 1.0.[19] In evaluating our case study applications, we use Nginx 1.11.9, OpenSSL 1.1.0c, and Memcached 1.4.33. Our Linux container instances also use the template for Ubuntu 14.04 LTS on amd64. All containers are created with abstract UNIX socket pass-through and EPC control enabled. The socket pass-through is configured with *aesmd* and *tpmsgxd*; the EPC limit is set to 1000K pages (4MB) for each container, making sure all tests run to completion without being throttled. Both *aesmd* and *tpmsgxd* run on the native host, as shown in Figure 1. We consider three running environments:

(1) *Stock*: stock kernel + original SGX driver.
(2) *lxcsgx*: *lxcsgx* kernel + SGX driver with EPC control.
(3) *LXC*: container running atop the *lxcsgx* environment.

Unless stated otherwise, assume all cases are repeated 100 times.

Simple SGX Applications. To measure the general overhead of enclave creation, local attestation, and remote attestation in each of our three environments, we use the SampleEnclave, LocalAttestation, and RemoteAttestation applications contained in the Intel SGX SDK. Our results are displayed in Figure 6(a). In all testing cases, the stock kernel has the best performance, followed first by our *lxcsgx* kernel, with added overhead from EPC control, and then LXC, which is slowed down by both the socket pass-through and EPC control in the container environment. The overhead introduced by *lxcsgx* and LXC is fairly small compared to the stock setting, ranging from 0.6% to 4.6% and 4.9% to 6.9%, respectively. In fact, since EPC control is enforced only during the initialization phase,[20] the overhead of *lxcsgx* and LXC will be amortized once the application reaches a stable state.

tpmsgx. To measure the performance of our software TPM, we run *tpmsgxd* on the host machine and our benchmark tool in all environments to time eight TPM commands, including: generating a 16-byte random number, loading an AES 128-bit key, encrypting/decrypting 32-byte data using AES128-GCM, generating an ECC256 key pair, computing DH key exchange based on ECC256, and signing/verifying 32-byte data or a 64-byte signature using ECDSA. We repeat all test cases with remote attestation enabled to establish a secure channel between the benchmarking tool and *tpmsgxd*, measuring the overhead due to the extra encryption/decryption. The results are shown in Figure 6(b).

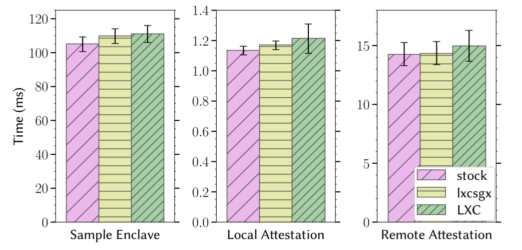
Random number generation, key loading, and encryption/decryption commands return a response to the client in less than 15 μs. When remote attestation is enabled, the response is slower due to the internal encryption before the result leaves the enclave. However, the response is still within 23 μs. Key loading has the lowest overhead with remote attestation enabled. Whereas other TPM commands return at least 16 bytes of data to be encrypted, key loading just returns a 4-byte key handle. Crypto operations involving ECC256 are slow compared to the first group of TPM commands, ranging from 200 μs to 280 μs. The overhead of remote attestation is not obvious anymore, as the operations themselves take most of the computation cycles. Since we run the same command 100 times, the overhead introduced by the SGX enclave creation, EPC control, and abstract socket pass-through in both *lxcsgx* and LXC environments are amortized, achieving close performance to the stock environment.

Nginx/OpenSSL. To measure the overhead introduced by *sgxengine* communicating with *tpmsgx* in Nginx, we configure Nginx to use NSA Suite B [49] combination 1[21] as the only cipher supported in the SSL/TLS handshake, and we compare against the original Nginx running on the native host. We use the OpenSSL client tool (*s_client*) to establish a HTTPS connection, and measure the connection setup

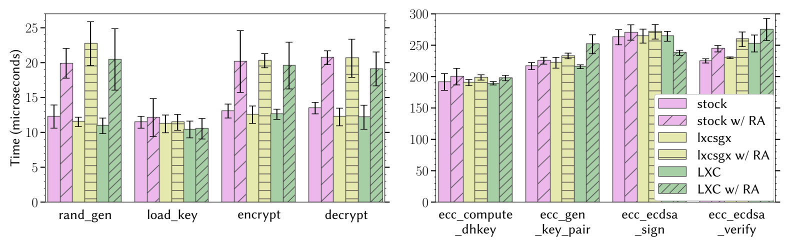
[19] No fundamental changes were made to the driver since the tested version (e.g., still no EPC control), but incremental changes to SGX added support for SGXv2 instructions.

[20] SGXv1 does not support dynamic allocation of more EPC pages to applications after enclave build time [64]. Our EPC control is also applicable to SGXv2; the only difference is that EPC control will be additionally enforced whenever more pages are allocated.

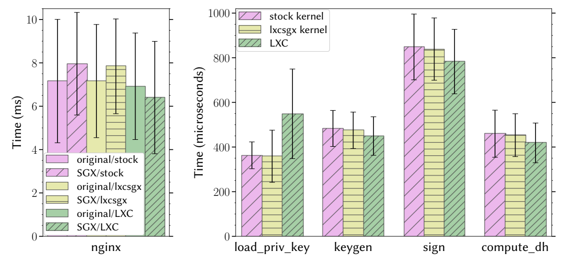
[21] AES with 128-bit key in GCM mode, ECDH using the 256-bit prime, modulus curve P-256 [DSS], TLS PRF with SHA-256 [SHS].



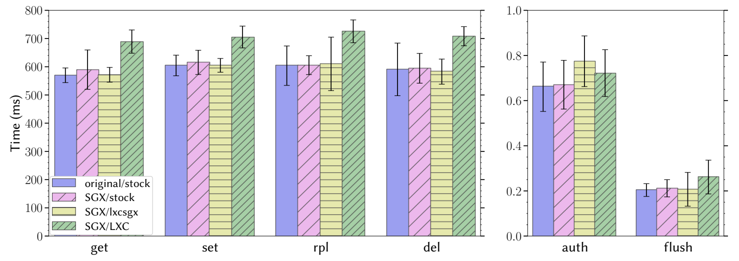
(a) Timing in ms of three simple SGX applications provided in the Intel SGX SDK in three settings (1) stock kernel, (2) *lxcsgx* kernel, and (3) LXC.



(b) Timing in μs of 8 TPM commands provided by *tpmsgxd*. Commands were run, both with and without remote attestation (RA) enabled, in three settings: (1) stock kernel, (2) *lxcsgx* kernel, and (3) LXC.



(c) Timing in ms of original Nginx and SGX version with *sgxengine* communicating with *tpmsgxd* (left); timing in μs for 4 operations in Nginx, each run in three settings (right).



(d) Timing in ms for 6 operations in Memcached. Each was run in four settings, with the stock kernel + original Memcached setting providing a baseline. Get, set, replace, and delete each reflect 2.5k requests, a moderate workload.

Figure 6: Our evaluation results for (a) simple SGX applications, (b) *tpmsgx*, and (c) Nginx, and (d) Memcached.

time. As shown in Figure 6(c)-L, the introduced overhead is less than 1 *ms* for all three SGX environments. Comparing to the raw Nginx performance, **the overhead introduced by *lxcsgx*, *tpmsgx*, and *sgxengine* together is less than 9.5%, which is much lower than Graphene-SGX (50%), SCONE (20%), and Panoply (24%).**[22]

To understand the overhead caused by *tpmsgx* in the *sgxengine*, we measure the time taken for key loading, ECDSA signing, key generation, and DH key computation, hooked by *sgxengine*, as shown in Figure 6(c)-R. The slowest operation is ECDSA signing, which still takes less than 0.9 *ms*. Besides key loading, all operations exhibit similar performance regardless of environment. If we compare this benchmark with the *tpmsgx* benchmark, we see a big portion of the overhead resulting from socket communication and OpenSSL rather than the software TPM itself (e.g., around 0.5 *ms* for signing). Interestingly, key loading is particularly slow in the LXC environment. We suspect this is related with the container rootfs being mounted under the native rootfs, complicating file accesses in the kernel.

Memcached. To measure the overhead introduced by performing the SASL authentication of Memcached in SGX, we use python-binary-memcached [50] as a client to benchmark the Memcached server with SASL authentication support. In each run, we set/get/replace/delete 2.5K entries and perform 1 authentication and 1 flush operation. We use stock kernel + unmodified Memcached environment as a baseline, comparing it against our SGX-version of Memcached under the three environments.

As shown in Figure 6(d), all SGX-enabled environments except LXC show comparative performance with the baseline, with overhead ranging from 0.3% to 3.5%. This is reasonable, since most of the

[22]Panoply did not evaluate Nginx. The number we reference is an average overhead for applications they tested.

Compilation	Min	Avg	Max	Stdev
Memcached	0.59	0.67	0.78	0.06
GCC w/ *gccsgx*	36.49	36.55	36.66	0.05
OpenSSL	32.61	33.08	33.48	0.27
GCC w/ *gccsgx*	693.22	695.73	707.86	4.11

Table 4: Memcached and OpenSSL compilation time in seconds using GCC both w/o and w/ *gccsgx*.

operations are pure database manipulations, which do not involve SGX. However, the LXC environment does show some overhead compared to the other configurations, ranging from 16.7% to 20.9%. We suspect this may be due to its intrinsic resource constraints, such as default quotas on CPU/memory, and networking overhead introduced by containers. The authentication operation captures the overhead of using SGX enclave to hold the SASL credentials and check against the password. With *lxcsgx* enabled, native host and LXC environments show an overhead of 9.1% to 17.1% relative to the stock kernel. Nevertheless, **the maximum overhead is less than 0.2 ms per authentication**.

***gccsgx*.** To measure the overhead of the GCC plugin during the compilation, we compile Memcached and OpenSSL using GCC, both without and with *gccsgx* enabled for 10 times each, as shown in Table 4. For Memcached, a normal compilation takes less than 1 second to finish. With *gccsgx* enabled, compilation takes around 36 seconds even though we are performing a lightweight static tainting analysis. The overhead compared to normal compilation is around 55x. This slowdown may be the result of Python implementation of the plugin, the number of global variables contained in the program, and the fact that *gccsgx* processes each function three times across the different compilation phases to pass the tainting information. OpenSSL compilation with *gccsgx* takes around 11 min, introducing

Application	Peak Stack	Peak Heap	Enclave Size
SampleEnclave	11 KB	12 KB	439 KB
LocalAttestation	3*6 KB	3*16 KB	3*1.1 MB
RemoteAttestation	3 KB	16 KB	1.0 MB
tpmsgx	2 KB	12 KB	1.1 MB
Memcached	2 KB	4 KB	130 KB

Table 5: EPC memory consumption for all applications we tested. Note that LocalAttestation creates 3 enclaves.

Application	EADD	EREMOVE	EPC Pages Used
aesmd	4574	4123	451
LXC/Memcached	75911	52458	23453
Total	80485	56581	23904

Table 6: Number of SGX instructions and EPC pages allocated/reclaimed by the SGX kernel driver with 100 LXC/Memcached instances. Total EPC consumption is around 96 MB.

an overhead of around 21x compared to the normal build. Fortunately, this overhead is just a one-time effort during partitioning.

Through *gccsgx*, we offer a lightweight alternative for partitioning SGX programs. Unlike the Glamdring [35] approach, we further reduce the TCB via minimal taint propagation based on tagged global variables and functions. We believe the plugin would be faster than Glamdring with respect to analysis time; unfortunately, as Glamdring is not open-source, we were unable to compare *gccsgx* with it.

EPC memory consumption. The EPC memory consumption of an application includes the static enclave size and the dynamic memory allocation. We measure static enclave size by looking at the size of the enclave image generated during the SGX build. We measure dynamic memory consumption using sgx-gdb provided by Intel SGX SDK with memory measurement enabled in application enclaves. The results are shown in Table 5. Most applications use 1 MB or less of EPC memory per enclave, thanks to program partitioning. The enclave binary itself appears to be responsible for most of the EPC consumption due to static linking against extra libraries, such as trusted C/C++ runtime and crypto. To support more keys generated or loaded into *tpmsgx*, we need to increase the heap size of the enclave, which is configurable during the SGX build. If we assume AES 128-bit keys, each EPC page (4KB) can hold around 256 keys. This means 1 MB EPC memory could support 65536 TLS connections at the same time. Note that these numbers are conservative. Actual runtime consumption is higher due to extra memory needed for enclave management and buffers holding data from untrusted memory.

We create and start 100 LXC containers with SASL-enabled Memcached running inside. We record the number of SGX instructions issued by the SGX kernel driver, to inspect the actual runtime EPC page consumption from the kernel, rather than counting what is visible in the user-space. The SGXv1 instructions we focus on are EADD, which adds a page into an uninitialized enclave, and EREMOVE, which removes a page from the EPC. As shown in Table 6, there are around 24K EPC pages used in total. Most of these are requested by the 100 containers.[23] Therefore, the actual EPC memory consumption is $24K * 4KB = 96MB$ in total. Each container consumes less than 1 MB EPC at runtime. Although EPC memory consumption is application-dependent, given a reasonable security analysis and program partitioning, having 1 MB EPC memory usage might be practical for many applications.

[23]We ignore the impact of other platform software, such as *aesmd.*

6 DISCUSSION

Abstract UNIX socket pass-through in *lxcsgx* may open a new attack vector on the host. However, mounting an abstract UNIX socket into a container should be more secure than mounting a traditional UNIX socket, which has well-known exploitations (20 CVEs [13]) due to its use of the file backend. The abstract UNIX socket is designed to solve known vulnerabilities in traditional UNIX sockets. Moreover, abstract UNIX sockets are still under the control of SELinux [36], which could also limit the impact of exploiting *aesmd* or *tpmsgxd.*

The EPC memory consumption control in *lxcsgx* currently only works for containers rather than normal processes running on the native host machine, although it can easily be extended to support EPC control per process. Since SGXv1 does not support adding more EPC pages after the enclave is initialized [64], the final EPC memory consumption is constrained by the static configuration of the enclave (determined by the user), and EPC quote (determined by the cloud provider). For SGXv2, EPC control is the only defense against (malicious) EPC memory oversubscription, because enclaves can request more EPC pages during the runtime, breaking the limit in the static configuration.[24] A future work for EPC control is to merge it into the *cgroups* [39] subsystem in the Linux kernel as a new resource controller, providing a more general group-wide policy control.

While fTPM [47] makes fair points on why ARM TrustZone [1] is the superior software TPM candidate, use of SGX comes with its own advantages. Since *tpmsgx* just relies on enclave code running at ring-3, extending or bug fixing for *tpmsgx* is much like normal user-space programming. This upgrade flexibility is important, as demonstrated when RSA encryption keys were exposed by a recent firmware bug found in Infineon TPMs [43]. The fix required hardware OEMs to change their own BIOS/UEFI to update the TPM firmware. *tpmsgx* is also very fast, running at CPU speed with a small software stack.

One limitation of *tpmsgx* lies in the persistent storage (sealing), since we could not stop the cloud provider from deleting the sealed data. As we consider a cooperative cloud environment, we assume the cloud provider would not erase user data in reality because doing so is in violation of the service contract with the end users. However, rolling back the old sealed data is possible since old data can still be unsealed by *tpmsgx*. A potential solution is to use the monotonic counter and trusted time features of Intel Management Engine (ME) to defend against these rollback attacks [29]. DoS attacks against *tpmgsx* are also possible when some containers issue a large number of requests. A potential solution is to add request rate control per container within *tpmsgx*, throttling requests from certain containers if the sending rate exceeds certain limits.

Automatic program partitioning is a hard problem that has been an active research topic for years. Being a lightweight static tainting analysis plugin for GCC, *gccsgx* cannot automatically generate compiler-ready code partitions, but it gives a framework for assisting with this problem that is quite challenging to do in a completely manual fashion. Developers still need to write in EDL to define ECalls and OCalls, which bridge the normal application and the enclave implementation. Nevertheless, *gccsgx* is designed to seamlessly work with the current build system (rather than requiring another toolchain,

[24] SGXv2's max-heap and max-stack parameters [64] only reserve the address space. Developers may specify a large number (e.g., 1 GB) for these limits to make sure the address space is large enough for future EPC expansion, in which case actual EPC page allocation remains unrestricted.

such as LLVM used by Glamdring [35]), to help facilitate program partitioning and to provide a minimum TCB option following the lightweight tainting analysis. For fully automatic SGX program partitioning with sound tainting analysis, one can refer to Glamdring, which can also be integrated into the *lxcsgx* infrastructure.

7 RELATED WORK

Intel SGX. Haven [5] is the first attempt to protect user applications from an untrusted cloud environment using SGX, by putting the unmodified application and a LibOS together into the enclave. Subsequently, VC^3 [51] tries to secure MapReduce computations in an untrusted cloud environment. Both Haven and VC^3 are based on an SGX emulator rather than real SGX hardware. Graphene-SGX is an extension of Graphene LibOS [59, 60] to support SGX. Similarly to Haven, Graphene-SGX relies on LibLinux to run the application inside the enclave without changes. Note that these solutions focus on easing SGX adoption for legacy applications but, unlike *lxcsgx*, downplay or ignore the potential impact of a large TCB inside the enclave and limited EPC memory available in a system

To reduce the TCB in the enclave, SCONE [3] replaces the LibOS with a shim layer, and supports running a whole Docker container inside the enclave. Ryoan [20] alternatively uses NaCl [66] to build an application sandbox within the enclave. While these solutions have a thinner middleware layer than LibOS approaches, putting the whole application into the enclave may still bloat the TCB, introducing a large attack surface as a result. Panoply [54] further reduces the TCB by partitioning programs and putting part of the application logic (namely "micron") into the enclave. Each application may depend on interaction with many microns; the creation and communication of multiple microns per application come at the cost of EPC memory consumption and performance degradation. Our solution instead keeps with the standard partitioning practice for SGX applications.

Eleos [45] notices issues caused by the limited EPC memory, and proposes a user-space memory allocator for EPC to reduce EPC page faults that result from EPC page swapping. We believe Eleos can be integrated into *lxcsgx* to improve the performance of SGX applications in general by reducing their EPC memory requirements. To automate the program partitioning using Intel SGX SDK, Glamdring [35] applies heavyweight data-flow analysis and backward slicing to find security sensitive code and data, as well as source-level transformation to generate compiler-ready code. It requires the LLVM toolchain rather than GCC, and the compilation overhead is not clear. In cases where a minimal, yet complete, TCB is desired, Glamdring may be integrated into *lxcsgx* as a substitute for our partitioning using *gccsgx*.

TPM. As the cornerstone of Trusted Computing, TPM [58] provides a hardware root of trust for software stacks. TPM 2.0 [4] even includes different specifications for PC client, mobile, and automotive systems. Unfortunately, these TPM chips are well-known to be slow and hard to patch or fix. Software TPM solutions instead try to provide better performance and/or extend more functionalities. vTPM [46] virtualizes TPM in the Xen environment, by having slave vTPMs in domains and a master vTPM in domain 0, which serializes the communication between vTPMs and the hardware TPM. cTPM [8] extends the TPM commands for cross-device applications by sharing an additional root key with the cloud. fTPM [47] implements a software TPM using TrustZone [1], which is available on mobile platforms. Nevertheless, due to their dependency on a hardware TPM or on running inside TrustZone's secure world, these solutions are still slow when compared to native application speed. Patching or upgrading is still challenging since most of these implementations are in the form of firmware. To the best of our knowledge, *tpmsgx* is the first software TPM solution to use SGX, achieving CPU speed and enabling normal user-space programming.

8 CONCLUSION

We design and implement *lxcsgx*, providing a practical Intel SGX setting for Linux containers in the cloud environment. This includes an infrastructure that supports remote attestation and EPC memory control for containers, a software TPM that can be more easily allow legacy application to leverage SGX, and a GCC plugin to assist with program partitioning. We retrofit different applications using the software TPM or program partitioning to use SGX, while reducing the TCB and EPC memory consumption at the same time. The evaluation shows reasonable overhead ranging from 6.9% to 20.9%, which is the lowest when compared to that of previous SGX solutions.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation under grant numbers CNS-1642973, CNS-1540217, and CNS-1815883; and by Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Intel Corporation.

REFERENCES

[1] Tiago Alves, Don Felton, et al. 2004. TrustZone: Integrated Hardware and Software Security. *ARM White Paper* 3, 4 (2004), 18–24.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP).*

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, and Others. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[4] Will Arthur and David Challener. 2015. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security.* Apress.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[6] Jethro Beekman. 2017. On the recent side-channel attacks on Intel SGX. https://jbeekman.nl/blog/2017/03/sgx-side-channel-attacks/ Accessed.

[7] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT).*

[8] Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. cTPM: A Cloud TPM for Cross-Device Trusted Applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[9] Common Vulnerabilities and Exposures. 2016. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[10] Common Vulnerabilities and Exposures. 2016. CVE-2016-8704. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8704.

[11] Common Vulnerabilities and Exposures. 2016. CVE-2016-8705. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8705.

[12] Common Vulnerabilities and Exposures. 2016. CVE-2016-8706. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8706.

[13] Common Vulnerabilities and Exposures. 2017. UNIX Socket Vulnerabilities. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=unix+socket+vulnerability Accessed.

[14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium.*

[15] CVE Details. 2017. GNU/TLS Vulnerability Statistics. https://www.cvedetails.com/product/4433/GNU-Gnutls.html?vendor_id=72 Accessed.

[16] CVE Details. 2017. OpenSSL Vulnerability Statistics. http://www.cvedetails.com/product/383/Openssl-Openssl.html?vendor_id=217 Accessed.

[17] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004), 5.

[18] Free Software Foundation, Inc. 2017. GNU Compiler Collection (GCC) Internals. https://gcc.gnu.org/onlinedocs/gccint/ Accessed.

[19] Patrick Galbraith. 2017. SASL memcached now available! https://blog.couchbase.com/sasl-memcached-now-available/ Accessed.

[20] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[21] Intel Corporation. 2016. Intel Software Guard Extensions for Linux OS. https://01.org/intel-softwareguard-extensions.

[22] Intel Corporation. 2016. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[23] Intel Corporation. 2016. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk.

[24] Intel Corporation. 2016. Intel® Software Guard Extensions SDK for Linux* OS Developer Reference . https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.7_Open_Source.pdf.

[25] Intel Corporation. 2016. Linux SGX Driver. https://github.com/01org/linux-sgx-driver.

[26] Intel Corporation. 2017. KVM SGX. https://github.com/intel/kvm-sgx.

[27] Intel Corporation. 2018. Intel-SA-00161: Q3 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html.

[28] Intel Developer Zone. 2017. SGX - is HeapMaxSize necessary? https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/607004#comment-1857071 Accessed.

[29] Intel Developer Zone. 2017. SGX enclaves. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/670388 Accessed.

[30] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun Kanuparthi, Thomas Eisenbarth, and Berk Sunar. 2017. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *arXiv preprint arXiv:1709.01552* (2017).

[31] Seo Jaebaek, Lee Byoungyoung, Kim Sungmin, Shih Ming-Wei, Shin Insik, Han Dongsu, and Kim Taesoo. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[32] Simon Johnson, Vincent Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. *White Paper* (March 2016).

[33] Dustin Kirkland. 2015. How many containers can you run on your machine? https://blog.ubuntu.com/2015/06/11/how-many-containers-can-you-run-on-your-machine.

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203* (2018).

[35] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC).*

[36] Peter Loscocco and Stephen Smalley. 2001. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the Ottawa Linux Symposium.*

[37] David Malcolm. 2017. GCC plugin that embeds CPython inside the compiler. https://github.com/davidmalcolm/gcc-python-plugin Accessed.

[38] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Security Symposium.*

[39] Paul Menage. 2016. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[40] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES).*

[41] Thomas Moyer, Kevin Butler, Joshua Schiffman, Patrick McDaniel, and Trent Jaeger. 2009. Scalable Web Content Attestation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).*

[42] John G Myers. 1997. Simple Authentication and Security Layer (SASL). (1997).

[43] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. 2017. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[44] OpenSSL. 2017. OpenSSL- Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[45] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys).* 238–253.

[46] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium.*

[47] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *Proceedings of the 25th USENIX Security Symposium.*

[48] Mark Russinovich. 2018. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing/.

[49] Margaret Salter and Russ Housley. 2012. *Suite B Profile for Transport Layer Security (TLS).* Technical Report. RFC 6460.

[50] Jayson Santos. 2017. python-binary-memcached. https://github.com/jaysonsantos/python-binary-memcached.

[51] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[52] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).*

[53] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[54] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS).*

[55] Igor Sysoev. 2017. NGINX. https://nginx.org/en/ Accessed.

[56] The H2O project. 2017. Privilege separation engine for OpenSSL / LibreSSL. https://github.com/h2o/neverbleed Accessed.

[57] Dave (Jing) Tian, Kevin R. B. Butler, Joseph I. Choi, Patrick McDaniel, and Padma Krishnaswamy. 2017. Securing ARP/NDP From the Ground Up. *IEEE Transactions on Information Forensics and Security (TIFS)* 12, 9 (2017), 2131–2143.

[58] Trusted Computing Group. 2011. Trusted Platform Module Main Specification, version 1.2, revision 116. https://linuxcontainers.org/.

[59] 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).*

[60] 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 USENIX Annual Technical Conference (ATC).*

[61] Jo Van Bulck, Minkin Marina, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium.*

[62] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[63] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. Async-Shock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[64] 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 Conference on Hardware and Architectural Support for Security and Privacy (HASP).*

[65] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[66] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P).*

[67] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitras, Alan Mislove, Aaron Schulman, and Christo Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *Proceedings of the Internet Measurement Conference (IMC).*