SCONE: Secure Linux Containers with Intel SGX

Sergei Arnautov[1], Bohdan Trach[1], Franz Gregor[1], Thomas Knauth[1], Andre Martin[1], Christian Priebe[2], Joshua Lind[2], Divya Muthukumaran[2], Dan O'Keeffe[2], Mark L Stillwell[2], David Goltzsche[3], David Eyers[4], Rüdiger Kapitza[3], Peter Pietzuch[2], and Christof Fetzer[1]

[1] *Fakultät Informatik, TU Dresden,* christof.fetzer@tu-dresden.de
[2] *Dept. of Computing, Imperial College London,* prp@imperial.ac.uk
[3] *Informatik, TU Braunschweig,* rrkapitz@ibr.cs.tu-bs.de
[4] *Dept. of Computer Science, University of Otago,* dme@cs.otago.ac.nz

Abstract

In multi-tenant environments, Linux containers managed by Docker or Kubernetes have a lower resource footprint, faster startup times, and higher I/O performance compared to virtual machines (VMs) on hypervisors. Yet their weaker isolation guarantees, enforced through software kernel mechanisms, make it easier for attackers to compromise the confidentiality and integrity of application data within containers.

We describe SCONE, a secure container mechanism for Docker that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. The design of SCONE leads to (i) a small trusted computing base (TCB) and (ii) a low performance overhead: SCONE offers a secure C standard library interface that transparently encrypts/decrypts I/O data; to reduce the performance impact of thread synchronization and system calls within SGX enclaves, SCONE supports user-level threading and asynchronous system calls. Our evaluation shows that it protects unmodified applications with SGX, achieving 0.6×–1.2× of native throughput.

1 Introduction

Container-based virtualization [53] has become popular recently. Many multi-tenant environments use Linux containers [24] for performance isolation of applications, Docker [42] for the packaging of the containers, and Docker Swarm [56] or Kubernetes [35] for their deployment. Despite improved support for hardware virtualization [21, 1, 60], containers retain a performance advantage over *virtual machines* (VMs) on hypervisors: not only are their startup times faster but also their I/O throughput and latency are superior [22]. Arguably they offer weaker security properties than VMs because the host OS kernel must protect a larger interface, and often uses only software mechanisms for isolation [8].

More fundamentally, existing container isolation mechanisms focus on protecting the environment from accesses by untrusted containers. Tenants, however, want to protect the confidentiality and integrity of their application data from accesses by unauthorized parties—not only from other containers but also from higher-privileged system software, such as the OS kernel and the hypervisor. Attackers typically target vulnerabilities in existing virtualized system software [17, 18, 19], or they compromise the credentials of privileged system administrators [65].

Until recently, there was no widely-available hardware mechanism for protecting user-level software from privileged system software. In 2015, Intel released the *Software Guard eXtensions* (SGX) [31] for their CPUs, which add support for secure *enclaves* [26]. An enclave shields application code and data from accesses by other software, including higher-privileged software. Memory pages belonging to an enclave reside in the *enclave page cache* (EPC), which cannot be accessed by code outside of the enclave. This makes SGX a promising candidate for protecting containers: the application process of a container can execute inside an enclave to ensure the confidentiality and integrity of the data.

The design of a secure container mechanism using SGX raises two challenges: (i) *minimizing* the size of the *trusted computing base* (TCB) inside an enclave while supporting existing applications in secure containers; and (ii) maintaining a *low performance overhead* for secure containers, given the restrictions of SGX.

Regarding the TCB size, prior work [6] has demonstrated that Windows applications can be executed in enclaves, but at the cost of a large TCB (millions of LOC), which includes system libraries and a library OS. Any vulnerability in the TCB may allow an attacker to access application data or compromise its integrity, which motivates us to keep a container's TCB size inside of the enclave small.

The performance overhead of enclaves comes from the fact that, since the OS kernel is untrusted, enclave code

cannot execute system calls. An enclave thread must copy memory-based arguments and leave the enclave before a system call. These thread transitions are expensive because they involve saving and restoring the enclave's execution state. In addition, enclaves have lower memory performance because, after cache misses, cache lines must be decrypted when fetched from memory. Accesses to enclave pages outside of the EPC cause expensive page faults.

To maintain a small TCB for secure containers, we observe that containers typically execute network services such as Memcached [23], Apache [44], NGINX [47] and Redis [46], which require only a limited interface for system support: they communicate with the outside via network sockets or `stdin/stdout` streams, use isolated or ephemeral file systems, and do not access other I/O devices directly. To mitigate the overhead of secure containers, we note that enclave code can access memory outside of the enclave without a performance penalty. However, for applications with a high system call frequency, the overhead of leaving and re-entering the enclave for each system call remains expensive.

We describe **SCONE**, a **Secure CONtainer Environment** for Docker that uses SGX to run Linux applications in secure containers. It has several desirable properties:

(1) Secure containers have a small TCB. SCONE exposes a *C standard library interface* to container processes, which is implemented by statically linking against a libc library [38] within the enclave. System calls are executed outside of the enclave, but they are *shielded* by transparently encrypting/decrypting application data on a per-file-descriptor basis: files stored outside of the enclave are therefore encrypted, and network communication is protected by transport layer security (TLS) [20]. SCONE also provides secure ephemeral file system semantics.

(2) Secure containers have a low overhead. To reduce costly enclave transitions of threads, SCONE provides a *user-level threading* implementation that maximizes the time that threads spend inside the enclave. SCONE maps OS threads to logical application threads in the enclave, scheduling OS threads between application threads when they are blocked due to thread synchronization.

SCONE combines this with an *asynchronous system call mechanism* in which OS threads outside the enclave execute system calls, thus avoiding the need for enclave threads to exit the enclave. In addition, SCONE reduces expensive memory accesses within the enclave by maintaining encrypted application data, such as cached files and network buffers, in non-enclave memory.

(3) Secure containers are transparent to Docker. Secure containers behave like regular containers in the Docker engine. Since container images are typically generated by experts, less experienced users can therefore benefit from SCONE, as long as they trust the creator of a secure container image. When executing secure containers, SCONE requires only an SGX-capable Intel CPU, an SGX kernel driver and an optional kernel module for asynchronous system call support.

Our experimental evaluation of SCONE on SGX hardware demonstrates that, despite the performance limitations of current SGX implementations, the throughput of popular services such as Apache, Redis, NGINX, and Memcached is 0.6×–1.2× of native execution, with a 0.6×–2× increase in code size. The performance of SCONE benefits from the asynchronous system calls and the transparent TLS encryption of client connections.

2 Secure Containers

Our goal is to create a *secure container* mechanism that protects the confidentiality and integrity of a Linux process' memory, code, and external file and network I/O from unauthorized and potentially privileged attackers.

2.1 Linux containers

Containers use OS-level virtualization [35] and have become increasingly popular for packaging, deploying and managing services such as key/value stores [46, 23] and web servers [47, 25]. Unlike VMs, they do not require hypervisors or a dedicated OS kernel. Instead, they use kernel features to isolate processes, and thus do not need to trap system calls or emulate hardware devices. This means that container processes can run as normal system processes, though features such as overlay file systems [10] can add performance overheads [22]. Another advantage of containers is that they are lightweight—they do not include the rich functionality of a standalone OS, but instead use the host OS for I/O operations, resource management, etc.

Projects such as LXC [24] and Docker [42] create containers using a number of Linux kernel features, including *namespaces* and the *cgroups* interface. By using the namespace feature, a parent process can create a child that has a restricted view of resources, including a remapped root file system and virtual network devices. The cgroups interface provides performance isolation between containers using scheduler features already present in the kernel.

For the deployment and orchestration of containers, frameworks such as Docker Swarm [56] and Kubernetes [35] instantiate and coordinate the interactions of containers across a cluster. For example, *micro-service* architectures [58] are built in this manner: a number

of lightweight containers that interact over well-defined network interfaces.

2.2 Threat model

Analogous to prior work [50, 6], we assume a powerful and active adversary who has *superuser* access to the system and also access to the physical hardware. They can control the entire software stack, including privileged code, such as the container engine, the OS kernel, and other system software. This empowers the adversary to replay, record, modify, and drop any network packets or file system accesses.

We assume that container services were not designed with the above privileged attacker model in mind. They may compromise data confidentiality or integrity by trusting OS functionality. Any programming bugs or inadvertent design flaws in the application beyond trusting the OS are outside of our threat model, as mitigation would require orthogonal solutions for software reliability. In our threat model, we also do not target denial-of-service attacks, or side-channel attacks that exploit timing and page faults [63]. These are difficult to exploit in practice, and existing mitigation strategies introduce a high performance overhead [9].

2.3 Intel SGX

Intel's *Software Guard Extensions* (SGX) [29, 30, 15] allow applications to ensure confidentiality and integrity, even if the OS, hypervisor or BIOS are compromised. They also protect against attackers with physical access, assuming the CPU package is not breached.

Enclaves are *trusted execution environments* provided by SGX to applications. Enclave code and data reside in a region of protected physical memory called the *enclave page cache* (EPC). While cache-resident, enclave code and data are guarded by CPU access controls. When moved to DRAM, data in EPC pages is protected at the granularity of cache lines. An on-chip *memory encryption engine* (MEE) encrypts and decrypts cache lines in the EPC written to and fetched from DRAM. Enclave memory is also integrity protected meaning that memory modifications and rollbacks are detected.

Non-enclave code cannot access enclave memory, but enclave code can access untrusted DRAM outside the EPC directly, e.g., to pass function call parameters and results. It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data.

Enclave life-cycle. Enclaves are created by untrusted code using the `ECREATE` instruction, which initializes an *SGX enclave control structure* (SECS) in the EPC. The `EADD` instruction adds pages to the enclave. SGX records the enclave to which the page was added, its virtual address and its permissions, and it subsequently enforces security restrictions, such as ensuring the enclave maps the page at the accessed virtual address. When all enclave pages are loaded, the `EINIT` instruction creates a cryptographic measurement, which can be used by remote parties for attestation.

For Intel Skylake CPUs [31], the EPC size is between 64 MB and 128 MB. To support enclave applications with more memory, SGX provides a paging mechanism for swapping pages between the EPC and untrusted DRAM: the system software uses privileged instructions to cause the hardware to copy a page into an encrypted buffer in DRAM outside of the EPC. Before reusing the freed EPC page, the system software must follow a hardware-enforced protocol to flush TLB entries.

Threading. After enclave initialization, an unprivileged application can execute enclave code through the `EENTER` instruction, which switches the CPU to enclave mode and jumps to a predefined enclave offset. Conversely, the `EEXIT` instruction causes a thread to leave the enclave. SGX supports multi-threaded execution inside enclaves, with each thread's enclave execution state stored in a 4 KB *thread control structure* (TCS).

Performance overhead. SGX incurs a performance overhead when executing enclave code: (i) since privileged instructions cannot execute inside the enclave, threads must exit the enclave prior to system calls. Such enclave transitions come at a cost—for security reasons, a series of checks and updates must be performed, including a TLB flush. Memory-based enclave arguments must also be copied between trusted and untrusted memory; (ii) enclave code also pays a penalty for writes to memory and cache misses because the MEE must encrypt and decrypt cache lines; and (iii) applications whose memory requirements exceed the EPC size must swap pages between the EPC and unprotected DRAM. Eviction of EPC pages is costly because they must be encrypted and integrity-protected before being copied to outside DRAM. To prevent address translation attacks, the eviction protocol interrupts all enclave threads and flushes the TLB.

2.4 Design trade-offs

Designing a secure Linux container using SGX requires a fundamental decision: *what system support should be placed inside an enclave to enable the secure execution of Linux processes in a container?* As we explore in this section, this design decision affects both (i) the *security properties* of containers, in terms of the size of the TCB and the exposed interface to the outside world, and (ii) the *performance* impact due to the inherent restric-

Service	TCB size	No. host system calls	Avg. throughput	Latency	CPU utilization
Redis	6.9×	<0.1×	0.6×	2.6×	1.1×
NGINX	5.7×	0.3×	0.8×	4.5×	1.5×
SQLite	3.8×	3.1×	0.3×	4.2×	1.1×

Table 1: Relative comparison of the LKL Linux library OS (no SGX) against native processes that use glibc

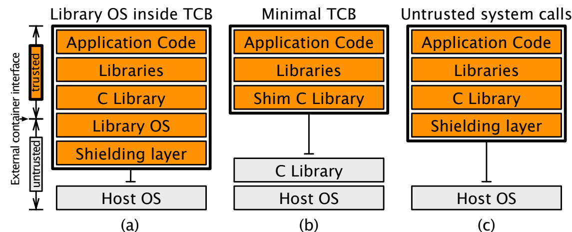


Figure 1: Alternative secure container designs

tions of SGX. To justify the design of SCONE, we first explore alternate design choices.

(1) External container interface. To execute unmodified processes inside secure containers, the container must support a C standard library (libc) interface. Since any libc implementation must use system calls, which cannot be executed inside of an enclave, a secure container must also expose an *external interface* to the host OS. As the host OS is untrusted, the external interface becomes an attack vector, and thus its design has security implications: an attacker who controls the host OS can use this interface to compromise processes running inside a secure container. A crucial decision becomes the size of (a) the external interface, and (b) the TCB required to implement the interface within the enclave.

Figure 1a shows a prior design point, as demonstrated by *Haven* [6], which minimizes the external interface by placing an entire Windows library OS inside the enclave. A benefit of this approach is that it exposes only a small external interface with 22 calls because a large portion of a process' system support can be provided by the library OS. The library OS, however, increases the TCB size inside of the enclave. In addition, it may add a performance overhead due to the extra abstractions (e.g., when performing I/O) introduced by the library OS.

We explore a similar design for Linux container processes. We deploy three typical containerized services using the *Linux Kernel Library* (LKL) [45] and the *musl* libc library [38], thus building a simple Linux library OS. The external interface of LKL has 28 calls, which is comparable to Haven.

Table 1 reports the performance and resource metrics for each service using the Linux library OS compared to a native *glibc* deployment. On average, the library OS increases the TCB size by 5×, the service latency by 4× and halves the service throughput. For Redis and NGINX, the number of system calls that propagate to the untrusted host OS are reduced as the library OS can handle many system calls directly. For SQLite, however, the number of system calls made to the host OS increases because LKL performs I/O at a finer granularity.

While our library OS lacks optimizations, e.g., minimizing the interactions between the library OS and the host OS, the results show that there is a performance degradation for both throughput and latency due to the kernel abstractions of the library OS. We conclude that the large TCB inside of the enclave and the performance overhead of this design is not a natural fit for containers.

Figure 1b shows the opposite, extreme design point: the external interface is used to perform all libc library calls made by the application. This raises the challenge of protecting the confidentiality and integrity of application data whilst exposing a wide interface. For example, I/O calls such as `read` and `write` could be used to compromise data within the enclave, and code inside the secure container cannot trust returned data. A benefit of this approach is that it leads to a minimal TCB inside the enclave—only a small shim C library needs to relay libc calls to the host libc library outside of the enclave.

Finally, Figure 1c shows a middle ground by defining the external interface at the level of system calls executed by the libc implementation. As we describe in §3, the design of SCONE explores the security and performance characteristics of this particular point in the design space. Defining the external container interface around system calls has the advantage that system calls already implement a privileged interface. While this design does not rely on a minimalist external interface to the host OS, we show that shield libraries can be used to protect a security-sensitive set of system calls: file descriptor based I/O calls, such as `read`, `write`, `send`, and `recv`, are shielded by transparently encrypting and decrypting the user data. While SCONE does not support some system calls, such as `fork`, `exec`, and `clone`, due to its user space threading model and the architectural limitations of SGX, they were not essential for the microservices that we targeted.

(2) System call overhead. All designs explored above pay the cost of executing system calls outside of the enclave (see §2.3). For container services with a high system call frequency, e.g., network-heavy services, this may result in a substantial performance impact. To quantify this issue, we conduct a micro-benchmark on an Intel Xeon CPU E3-1230 v5 at 3.4 GHz measuring the maximum rate at which `pwrite` system calls can be executed with and without an enclave. The benchmark is implemented using the Intel SGX SDK for Linux [32], which

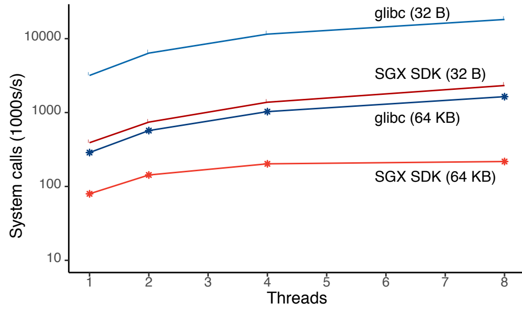


Figure 2: Number of executed `pwrite` system calls with an increasing number of threads

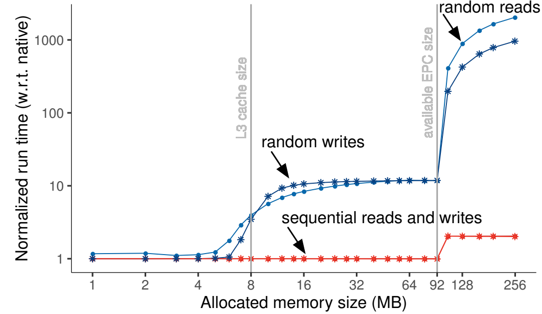


Figure 3: Normalized overhead of memory accesses with SGX enclaves

performs synchronous system calls with threads leaving and re-entering the enclave. We vary the number of threads and the `pwrite` buffer size.

Figure 2 shows that the enclave adds an overhead of an order of magnitude. The performance with large buffer sizes is limited by the copy overhead of the memory-based arguments; with small buffers, the main cost comes from the threads performing enclave transitions. We conclude that efficient system call support is a crucial requirement for secure containers. A secure container design must therefore go beyond simple synchronous support for system calls implemented using thread transitions.

(3) Memory access overhead. The memory accesses of a secure container process are affected by the higher overhead of accessing enclave pages (see §2.3). We explore this overhead using a micro-benchmark built with the Linux SGX SDK on the same hardware. The benchmark measures the time for both sequential and random read/write operations, normalized against a deployment without an enclave. All operations process a total of 256 MB, but access differently-sized memory regions.

Figure 3 shows that, as long as the accessed memory fits into the 8 MB L3 cache, the overheads are negligible. With L3 cache misses, there is a performance overhead of up to 12× for the random memory accesses. When the accessed memory is beyond the available EPC size, the triggered page faults lead to an overhead of three orders of magnitude. Sequential operations achieve better performance due to CPU prefetching, which hides some of the decryption overheads: they experience no overhead for memory ranges within the EPC size and a 2× overhead for sizes beyond that.

These results show that, for performance reasons, a secure container design should reduce access to enclave memory. Ideally, it should use untrusted non-enclave memory as much as possible, without compromising the offered security guarantees.

3 SCONE Design

Our objective is to offer *secure containers* on top of an untrusted OS: a secure container must protect containerized services from the threats defined in §2.2. We also want secure containers to fit transparently into existing Docker container environments: system administrators should be able to build secure container images with the help of Docker in a trusted environment and run secure containers in an untrusted environment.

3.1 Architecture

Figure 4 gives an overview of the SCONE architecture:

(1) SCONE exposes an *external interface* based on system calls to the host OS, which is shielded from attacks. Similar to what is done by the OS kernel to protect itself from user space attacks, SCONE performs sanity checks and copies all memory-based return values to the inside of the enclave before passing the arguments to the application (see §3.4). To protect the integrity and confidentiality of data processed via file descriptors, SCONE supports transparent encryption and authentication of data through *shields* (see §3.2).

(2) SCONE implements *M:N threading* to avoid the cost of unnecessary enclave transitions: M enclave-bound application threads are multiplexed across N OS threads. When an application thread issues a system call, SCONE checks if there is another application thread that it can wake and execute until the result of the system call is available (see §3.3).

(3) SCONE offers container processes an *asynchronous system call interface* to the host OS. Its implementation uses shared memory to pass the system call arguments and return values, and to signal that a system call should be executed. System calls are executed by separate threads running in a SCONE kernel module. Hence, the threads inside the enclave do not have to exit when performing system calls (see §3.4).

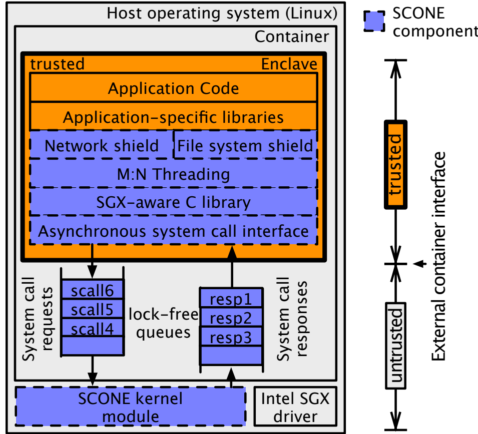


Figure 4: SCONE architecture

(4) SCONE integrates with existing Docker container environments, and ensures that secure containers are compatible with standard Linux containers (see §3.5). The host OS, however, must include a Linux SGX driver and, to boost performance, a SCONE kernel module. Note that SCONE does not use any functionality from the Intel Linux SDK [32] apart from the Linux SGX driver.

3.2 External interface shielding

So far, many popular services, such as Redis and Memcached, have been created under the assumption that the underlying OS is trusted. Such services therefore store files in the clear, communicate with other processes via unencrypted TCP channels (i.e., without TLS), and output to `stdout` and `stderr` directly.

To protect such services in secure containers, SCONE supports a set of *shields*. Shields focus on (1) preventing low-level attacks, such as the OS kernel controlling pointers and buffer sizes passed to the service (see §3.4); and (2) ensuring the confidentiality and integrity of the application data passed through the OS. A shield is enabled by statically linking the service with a given *shield library*. SCONE supports shields for (1) the transparent encryption of files, (2) the transparent encryption of communication channels via TLS, and (3) the transparent encryption of console streams.

When a file descriptor is opened, SCONE can associate the descriptor with a shield. A shield also has configuration parameters, which are encrypted and can be accessed only after the enclave has been initialized.

Note that the shields described below focus only on application data, and do not verify data maintained by the OS, such as file system metadata. If the integrity of such data is important, further shields can be added.

File system shield. The file system shield protects the confidentiality and integrity of files: files are authenticated and encrypted, transparently to the service. For the file system shield, a container image creator must define three disjoint sets of file path prefixes: prefixes of (1) *unprotected* files, (2) *encrypted and authenticated* files, and (3) *authenticated* files. When a file is opened, the shield determines the longest matching prefix for the file name. Depending on the match, the file is authenticated, encrypted, or just passed through to the host OS.

The file system shield splits files into blocks of fixed sizes. For each block, the shield keeps an authentication tag and a nonce in a metadata file. The metadata file is also authenticated to detect modifications. The keys used to encrypt and authenticate files as well as the three prefix sets are part of the configuration parameters passed to the file system shield during startup. For immutable file systems, the authentication tag of the metadata file is part of the configuration parameters for the file system shield. At runtime the metadata is maintained inside the enclave.

Containerized services often exclusively use a read-only file system and consider writes to be ephemeral. While processes in a secure container have access to the standard Docker `tmpfs`, it requires costly interaction with the kernel and its file system implementation. As a lightweight alternative, SCONE also supports a dedicated secure *ephemeral file system* through its file system shield. The shield ensures the integrity and confidentiality of ephemeral files: the ephemeral file system maintains the state of modified files in non-enclave memory. Our evaluation results show that the performance of ephemeral files is better than those of `tmpfs` (see §4.3).

The ephemeral file system implementation is resilient against *rollback attack*: after restarting the container process, the file system returns to a preconfigured startup state that is validated by the file system shield, and therefore it is not possible for an attacker to rollback the file system to an intermediate state. This is also true during runtime, since the metadata for files' blocks resides within the enclave.

Network shield. Some container services, such as Apache [44] and NGINX [47], always encrypt network traffic; others, such as Redis [46] and Memcached [23], assume that the traffic is protected by orthogonal means, such as TLS proxies, which terminate the encrypted connection and forward the traffic to the service in plaintext. Such a setup is appropriate only for data centers in which the communication between the proxy and the service is assumed to be trusted, which is incompatible with our threat model: an attacker could control the unprotected channel between the proxy and the service and modify the data. Therefore, for secure containers, a TLS network connection must be terminated inside the enclave.

SCONE permits clients to establish secure tunnels to

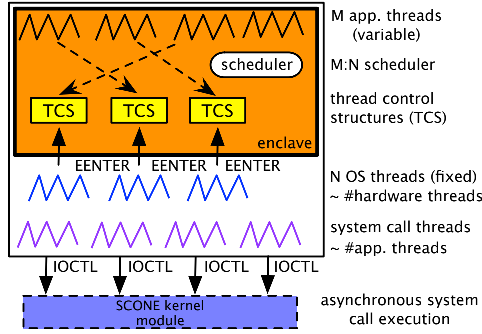


Figure 5: M:N threading model

container services using TLS. It wraps all socket operations and redirects them to a *network shield*. The network shield, upon establishing a new connection, performs a TLS handshake and encrypts/decrypts any data transmitted through the socket. This approach does not require client- or service-side changes. The private key and certificate are read from the container's file system. Thus, they are protected by the file system shield.

Console shield. Container environments permit authorized processes to attach to the `stdin`, `stdout`, and `stderr` console streams. To ensure the confidentiality of application data sent to these streams, SCONE supports transparent encryption for them. The symmetric encryption key is exchanged between the secure container and the SCONE client during the startup procedure (see §3.5).

Console streams are unidirectional, which means that they cannot be protected by the network shield whose underlying TLS implementation requires bidirectional streams. A *console shield* encrypts a stream by splitting it into variable-sized blocks based on flushing patterns. A stream is protected against replay and reordering attacks by assigning each block a unique identifier, which is checked by the authorized SCONE client.

3.3 Threading model

SCONE supports an M:N threading model in which M application threads inside the enclave are mapped to N OS threads. SCONE thus has fewer enclave transitions, and, even though the maximum thread count must be specified at enclave creation time in SGX version 1 [29], SCONE supports a variable number of application threads.

As shown in Figure 5, multiple OS threads in SCONE can enter an enclave. Each thread executes the *scheduler*, which checks if: (i) an application thread needs to be woken due to an expired timeout or the arrival of a system call response; or (ii) an application thread is waiting to be scheduled. In both cases, the scheduler executes the associated thread. If no threads can be executed, the scheduler backs off: an OS thread may choose to sleep outside of the enclave when the back-off time is longer than the time that it takes to leave and reenter the enclave.

The number of OS threads inside the enclave is typically bound by the number of CPU cores. In this way, SCONE utilizes all cores without the need for a large number of OS threads inside the enclave. The scheduler does not support preemption. This is not a limitation in practice because almost all application threads perform either system calls or synchronization primitives at which point the scheduler can reschedule threads.

In addition to spawning N OS threads inside the enclave, SCONE also "captures" several OS threads inside the SCONE kernel module. The threads dequeue requests from the system call request queue, perform system calls, and enqueue results into the response queue (see Figure 4). The system call threads reside in the kernel indefinitely to eliminate the overhead of kernel mode switches. The number of system call threads must be at least the number of application threads to avoid stalling when system call threads block. Periodically, the system call threads leave the kernel module to trigger Linux housekeeping tasks, such as the cleanup of TCP state. When there are no pending system calls, the threads back-off exponentially to reduce CPU load.

SCONE does not support the `fork` system call. Enclave memory is tied to a specific process, and therefore the execution of `fork` would require the allocation, initialization, and attestation of an independent copy of an enclave. In current SGX implementations, the OS kernel cannot copy enclave memory to achieve this.

3.4 Asynchronous system calls

Since SGX does not allow system calls to be issued from within an enclave, they must be implemented with the help of calls to functions outside of the enclave. This means that the executing thread must copy memory-based arguments to non-enclave memory, exit the enclave and execute the outside function to issue the system call. When the system call returns, the thread must re-enter the enclave, and copy memory-based results back to the enclave. As we showed in §2.4, such *synchronous system calls* have acceptable performance only for applications with a low system call rate.

To address this problem, SCONE also provides an *asynchronous system call interface* [52] (see Figure 6). This interface consists of two lock-free, multi-producer, multi-consumer queues: a *request queue* and a *response queue*. System calls are issued by placing a request into the request queue. An OS thread inside the SCONE kernel module receives and processes these requests. When the system call returns, the OS thread places the result into the response queue.

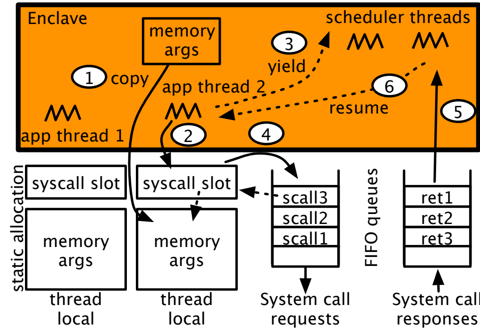


Figure 6: Asynchronous system calls

As shown in Figure 6, an application thread first copies memory-based arguments outside of the enclave ① and adds a description of the system call to a `syscall_slot` data structure ②, containing the system call number and arguments. The `syscall_slot` and the arguments use thread-local storage, which is reused by subsequent system calls. Next the application thread yields to the scheduler ③, which will execute other application threads until the reply to the system call is received in the response queue. The system call is issued by placing a reference to the `syscall slot` into the request queue ④. When the result is available in the response queue ⑤, buffers are copied to the inside of the enclave, and all pointers are updated to point to enclave memory buffers. As part of the copy operation, there are checks of the buffer sizes, ensuring that no *malicious* pointers referring to the outside of an enclave can reach the application. Finally, the associated application thread is scheduled again ⑥.

The enclave code handling system calls also ensures that pointers passed by the OS to the enclave do not point to enclave memory. This check protects the enclave from memory-based Iago attacks [12] and is performed for all shield libraries.

3.5 Docker integration

We chose to integrate SCONE with Docker because it is the most popular and widely used container platform. A future version of SCONE may use the open container platform [28], which would make it compatible with both Docker and rkt (CoreOS) [48]. With SCONE, a secure container consists of a single Linux process that is protected by an enclave, but otherwise it is indistinguishable from a regular Docker container, e.g., relying on the shared host OS kernel for the execution of system calls.

The integration of secure containers with Docker requires changes to the build process of secure images, and client-side extensions for spawning secure containers and for secure communication with these containers.

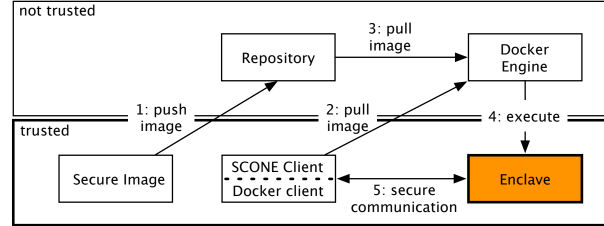


Figure 7: Using secure containers with Docker

SCONE does not require modifications to the Docker Engine or its API, but it relies on a wrapper around the original Docker client. A *secure SCONE client* is used to create configuration files and launch containers in an untrusted environment. SCONE supports a typical Docker workflow: a developer publishes an image with their application, and a user can customize the image by adding extra layers.

Image creation. Images are created in a trusted environment (see Figure 7). The image creator must be familiar with the security-relevant aspects of the service, e.g., which files to protect and which shields to activate.

To create a secure container image, the image creator first builds a SCONE executable of the application. They statically compile the application with its library dependencies and the SCONE library. SCONE does not support shared libraries by design to ensure that all enclave code is verified by SGX when an enclave is created.

Next, the image creator uses the SCONE client to create the metadata necessary to protect the file system. The client encrypts specified files and creates a *file system (FS) protection* file, which contains the message authentication codes (MACs) for file chunks and the keys used for encryption. The FS protection file itself is encrypted and added to the image. After that, the secure image is published using standard Docker mechanisms. SCONE does not need to trust the Docker registry, because the security-relevant parts are protected by the FS protection file.

If the image creator wants to support the composition of a secure Docker image [42], they only sign the FS protection file with their public key, but do not encrypt it. In this way, only its integrity is ensured, permitting additional customization. The confidentiality of the files is assured only after finishing the customization process.

Container startup. Each secure container requires a *startup configuration file* (SCF). The SCF contains keys to encrypt standard I/O streams, a hash of the FS protection file and its encryption key, application arguments and environment variables. Only an enclave whose identity has been verified can access the SCF. Since SGX does not protect the confidentiality of enclave code, em-

bedding the startup configuration in the enclave itself is not an option. Instead, after the executable has initialized the enclave, the SCF is received through a TLS-protected network connection, established during enclave startup [2]. In production use, the container owner would validate that the container is configured securely before sending it the SCF. The SGX remote attestation mechanism [29] can attest to the enclave to enable this validation, but our current SCONE prototype does not support remote attestation.

4 Evaluation

Our evaluation of SCONE on SGX hardware is split into three parts: (i) we present application benchmarks for Apache [44], NGINX [47], Redis [46] and Memcached [23]. We compare the performance of these applications with SCONE against native variants (§4.2); (ii) we evaluate the performance impact of SCONE's file system shield with a set of micro-benchmarks (§4.3); and (iii) we discuss results from a micro-benchmark regarding the system call overhead (§4.4).

4.1 Methodology

All experiments use an Intel Xeon E3-1270 v5 CPU with 4 cores at 3.6 GHz and 8 hyper-threads (2 per core) and 8 MB cache. The server has 64 GB of memory and runs Ubuntu 14.04.4 LTS with Linux kernel version 4.2. We disable dynamic frequency scaling to reduce interference. The workload generators run on a machine with two 14-core Intel Xeon E5-2683 v3 CPUs at 2 GHz with 112 GB of RAM and Ubuntu 15.10. Each machine has a 10 Gb Ethernet NIC connected to a dedicated switch. The disk configuration is irrelevant as the workloads fit entirely into memory.

We evaluate two web servers, Apache [44], and NGINX [47]; Memcached [23]; Redis [46]; and SQLite [55]. The applications include a mix of compute (e.g., SQLite) and I/O intensive (e.g., Apache and Memcached) workloads. We compare the performance of three variants for each application: (i) one built with the GNU C library (glibc); (ii) one built with the musl [38] C library adapted to run inside SGX enclaves with synchronous system calls (SCONE-sync); and (iii) one built with the same musl C library but with asynchronous system calls (SCONE-async). We compare with glibc because it is the standard C library for most Linux distributions, and constitutes a more conservative baseline than musl. In our experiments, applications compiled against glibc perform the same or better than the musl-based variants. The application process (and Stunnel) execute inside a Docker container.

Application	Worker threads		Enclave threads		Syscall threads	
	async	sync	async	sync	async	sync
Apache	25	25	4	8	32	-
NGINX	1	1	1	1	16	-
Redis	1	1	1	1	16	-
Memcached	4	8	4	8	32	-

Table 2: Thread configuration used for applications

SCONE-async uses the SCONE kernel module to capture system call threads in the kernel. For each application and variant, we configure the number of threads (see §3.3) to give the best results, as determined experimentally. We summarize the thread configuration in Table 2. Worker threads are threads created by the application, e.g., using `pthread_create()`. In the glibc variant, worker threads are real OS threads, while in SCONE they represent user space threads. Enclave threads are OS threads that run permanently inside the enclave, while system call threads are OS threads that run permanently outside. With SCONE-sync, there are no dedicated system call threads because the enclave threads synchronously exit the enclave to perform system calls.

For applications that do not support encryption (e.g., Memcached and Redis), we use Stunnel [61] to encrypt their communication in the glibc variant. When reporting CPU utilization, the application's glibc variant includes the utilization due to Stunnel processes. In SCONE, the network shield subsumes the functionality of Stunnel.

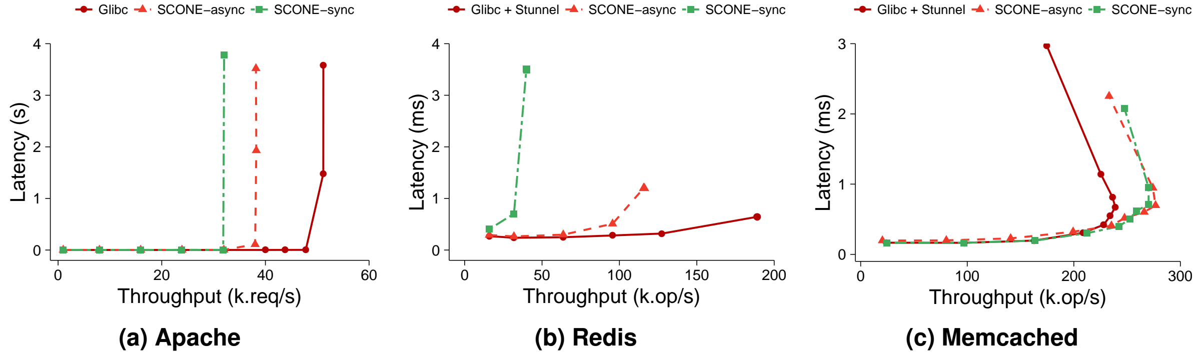
Reported data points are based on ten runs, and we compute the 30% trimmed mean (i.e., without the top and bottom 30% outliers) and its variance. The trimmed mean is a robust estimator insensitive to outliers: it measures the central tendency even with jitter. Unless stated otherwise, the variance is small, and we omit error bars.

4.2 Application benchmarks

Apache is a highly configurable and mature web server, originally designed to spawn a process for each connection. This differs from the architecture of the other benchmarked web server—NGINX employs an event-driven design. By default, it uses a single thread but current versions can be configured to use multiple threads.

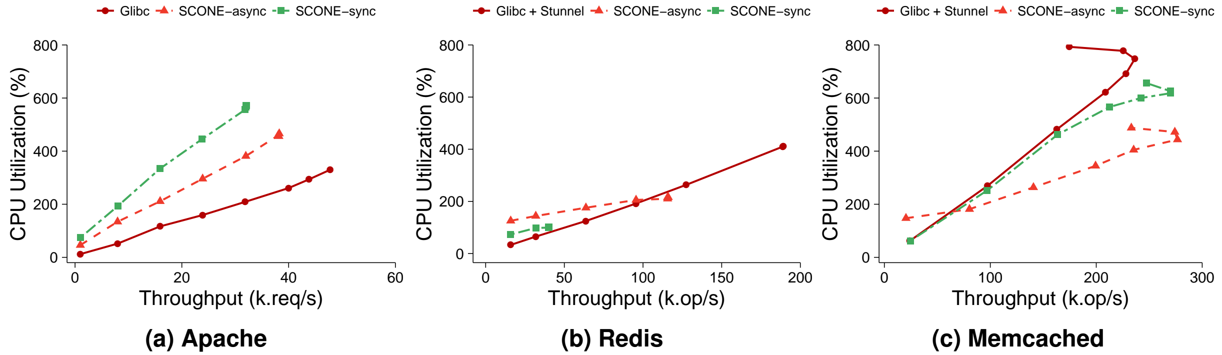
We use wrk2 [62] to fetch a web page. We increase the number of concurrent clients and the frequency at which they retrieve the page until the response times start to degrade. Since Apache supports application-level encryption in the form of HTTPS, we do not use Stunnel or SCONE's network shield.

Figure 8a shows that all three variants exhibit comparable performance until about 32,000 requests per sec-



(a) Apache (b) Redis (c) Memcached

Figure 8: Throughput versus latency for Apache, Redis, and Memcached



(a) Apache (b) Redis (c) Memcached

Figure 9: CPU utilization for Apache, Redis, and Memcached

ond, at which point the latency of the SCONE-sync increases dramatically. SCONE-async performs slightly better, reaching 38,000 requests per second. The glibc variant achieves 48,000 requests per second.

As shown in Figure 9a, SCONE-sync utilizes the CPU more despite the fact that SCONE-async uses extra threads to execute system calls. As we show below, the synchronous system call interface is not as performant as the asynchronous interface, resulting in a higher CPU utilization. However, SCONE-async has a higher CPU utilization than glibc. This is caused by the slower execution time of Apache running inside the enclave as well as the extra threads used in the SCONE kernel module to execute the system calls.

Redis is a distributed in-memory key/value store and represents an I/O-intensive network service. Typical workloads with many concurrent operations exhibit a high system call frequency. Persistence in Redis is achieved by forking and writing the state to stable storage in the background. Fundamentally, forking for enclave applications is difficult to implement and not supported by SCONE. Hence, we deploy Redis solely as an in-memory store.

We use workloads A to D from the YCSB benchmark suite [14]. In these workloads, both the application code and data fit into the EPC, so the SGX driver does not need to page-in EPC pages. We present results only for workload A (50% reads and 50% updates); the other workloads exhibit similar behaviour. We deploy 100 clients and increase their request frequency until reaching maximum throughput.

Figure 8b shows that Redis with glibc achieves a throughput of 189,000 operations per second. At this point, as shown in Figure 9b, Redis, which is single-threaded, becomes CPU-bound with an overall CPU utilization of 400% (4 hyper-threads): 1 hyper-thread is used by Redis, and 3 hyper-threads are used by Stunnel.

SCONE-sync cannot scale beyond 40,000 operations per second (21% of glibc), also due to Redis' single application thread. By design, SCONE-sync performs encryption as part of the network shield within the application thread. Hence, it cannot balance the encryption overhead across multiple hyper-threads, as Stunnel does, and its utilization peaks at 100%.

SCONE-async reaches a maximum throughput of 116,000 operations per second (61% of glibc). In addition to the single application thread, multiple OS threads execute system calls inside the SCONE kernel module. Ultimately, SCONE-async is also limited by the single Redis application thread, which is why CPU utilization peaks at 200% under maximum throughput. The per-

Variant	LOC (1000s)	libc.a Size (KB)	Apache Size (KB)	NGINX Size (KB)	Redis Size (KB)	Memcached Size (KB)	SQLite Size (KB)
glibc	1195	4710	5437	3975	2445	1143	801
musl	88	1498	4546	3088	682	289	832
SCONE libc	97	1572	4829	3286	853	357	880
Shielded SCONE libc	187	2491	5496	4082	1665	1208	1724

Table 3: Comparison of the binary sizes of statically linked applications using SCONE with native libc variants

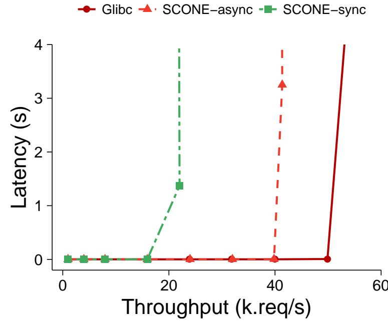


Figure 10: Throughput versus latency for NGINX

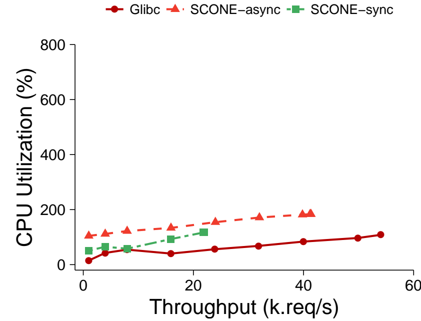


Figure 11: CPU utilization for NGINX

formance of SCONE-async is better than SCONE-sync because SCONE-async has a higher single thread system call throughput. However, as SCONE-async does not assign TLS termination to a separate thread either, it cannot reach the throughput of glibc.

Memcached, a popular key/value cache, is evaluated with the same YCSB workloads as Redis. We increase the number of clients until each variant reaches its saturation point. Again, the application fits into the EPC.

Figure 8c shows that the client latencies of all three variants exhibited for a given throughput are similar until approximately 230,000 operations per second, at which point the latency of glibc starts to increase faster than that of SCONE-async and SCONE-sync. The maximum achieved throughput of the SCONE variants (277,000 operations per second for SCONE-async and 270,000 operations per second for SCONE-sync) is higher than that of the glibc variant (238,000 operations per second).

For all three variants, the CPU utilization in Figure 9c increases with throughput. Both SCONE variants experience a lower CPU utilization than the Memcached and Stunnel deployment. This differs from single-threaded Redis, as Memcached can utilize more CPU cores with multiple threads and starts to compete for CPU cycles with Stunnel. SCONE's network shield encryption is more efficient, allowing it to have a lower CPU load and achieve higher throughput.

NGINX is a web server with an alternative architecture to Apache—NGINX typically uses one worker process per CPU core. Each worker process executes a non-blocking, event-driven loop to handle connections, process requests and send replies. We configure NGINX to use a single worker process.

Figure 10 and Figure 11 show the throughput and CPU utilization for NGINX, respectively. The glibc variant achieves approximately 50,000 requests per second—similar to Apache, but at a much lower utilization (>300% vs. 100%). SCONE-sync shows good performance up to 18,000 requests per second. This is less than Apache, but NGINX also only utilizes a single thread. With SCONE-async, NGINX achieves 80% of the native performance again at a much lower overall CPU utilization than Apache (200% vs. 500%). This demonstrates that SCONE can achieve acceptable performance both for multi-threaded applications, such as Apache, and non-blocking, event-based servers, such as NGINX.

Code size. We compare the code sizes of our applications in Table 3. The size of applications linked against the musl C library is in most cases smaller than that of applications linked against glibc. The modifications to musl for running inside of enclaves add 11,000 LOC, primarily due to the asynchronous system call wrappers (5000 LOC of generated code). The shields increase the code size further (around 99,000 LOC), primarily due to the TLS library. Nevertheless, the binary sizes of shielded applications increase only to 0.6×–2× compared to the glibc-linked variants.

Although the binary size of the SCONE libc variant is only 74 KB larger than musl, the binary size of some programs compiled against SCONE increase by more than this amount. This is due to how we build the SCONE

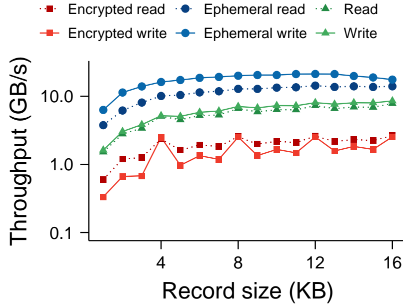


Figure 12: Throughput of random reads/writes with ephemeral file system versus `tmpfs`

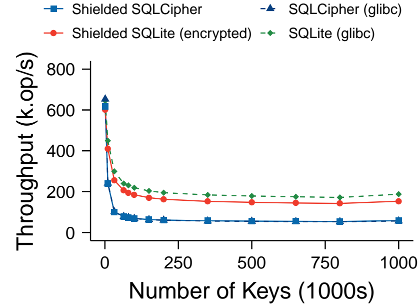


Figure 13: Throughput of SQLite and SQLCipher with file system shield

Application	SCONE-async	
	T'put	CPU util.
Apache	0.8×	1.4×
Redis	0.6×	1.0×
Memcached	1.2×	0.6×
NGINX	0.8×	1.8×

Table 4: Normalized application performance

binary: an application and all its dependent libraries are linked into a position-independent shared object file. The relocation information included in the shared object file comprises up to a few hundred kilobytes.

Discussion. Table 4 summarizes the normalized results for all the throughput-oriented applications. For Apache, SCONE-async achieves performance on par with that of the native version. The performance of SCONE-async is, as expected, faster than that of SCONE-sync—SCONE-async can switch to another Apache thread while waiting for system call results.

For single-threaded applications such as Redis, asynchronous system calls offer limited benefit, mostly due to the faster response times compared to synchronous system calls. However, SCONE-async cannot run other application threads while waiting for the result of a system call. The same is true for NGINX despite supporting multiple threads. In our experiments, NGINX did not scale as well as Apache with the same number of threads.

The results demonstrate that SCONE-async can execute scalable container services with throughput and latency that are comparable to native versions. This is in some sense surprising given the micro-benchmarks from §2.4, as they would suggest that applications inside SGX enclaves would suffer a more serious performance hit.

4.3 File system shield

We evaluate the performance of the file system shield with micro-benchmarks. We use IOZone [34] to subject the shield to random and sequential reads/writes. We compare the throughput of three different IOZone versions: (i) native glibc accessing a `tmpfs` file system; (ii) SCONE with the ephemeral file system without protection; and (iii) SCONE with an encrypted ephemeral file system.

Figure 12 shows the measured throughput as we vary the record size. We observe that IOZone on an ephemeral file system achieves a higher throughput than the native glibc IOZone on `tmpfs`. This is because the application does not issue any system calls when accessing data on the ephemeral file system—instead, it accesses the untrusted memory directly, without exiting the enclave. Enabling encryption on the ephemeral file system reduces the throughput by an order of magnitude.

In addition to the synthetic IOZone benchmark, we also measure how the shield impacts the performance of SQLite. We compare four versions: (i) SQLite with no protection; (ii) SQLCipher [54], which is SQLite with application level encryption; (iii) shielded SQLite on an encrypted ephemeral file system; and (iv) SQLCipher on an ephemeral file system (no authentication or encryption). The shielded versions use the memory-backed ephemeral file system, whereas the glibc versions of SQLite and SQLCipher use the standard Linux `tmpfs`. All protected versions use 256-bit keys.

Figure 13 shows the result of the SQLite benchmark. With small datasets (1000 keys), no cryptographic operations are necessary because the working set fits into SQLite's in-memory cache. This results in comparable performance across all versions. With bigger datasets, however, performance of the different versions diverge from the baseline, as SQLite starts to persist data on the file system resulting in an increasing number of cryptographic operations. We observe that the achieved throughput of the SQLCipher versions (approx. 60,000 operations per second) is about 35% of the baseline (approx. 170,000 operations per second), while the shielded SQLite version (approx. 140,000 operations per second) reaches about 80%. This is because the file

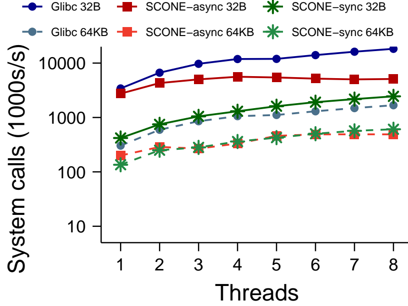


Figure 14: Frequency of system calls with asynchronous system call support

system shield of SCONE uses AES-GCM encryption, which outperforms AES in cipher block chaining (CBC) mode as used by default in SQLCipher. CBC mode restricts parallelism and requires an additional authentication mechanism.

4.4 Asynchronous system calls

Similar to Figure 2, Figure 14 shows how many `pwrite` calls can be executed by SCONE-async, SCONE-sync and natively. The x-axis refers to the number of OS-visible enclave threads for SCONE-async and SCONE-sync; for glibc, this is the number of native Linux threads. We vary the buffer size to see how the copy overhead influences the system call frequency. Larger buffers increase the overhead to move system call parameters from the enclave to the outside memory (§3.4); smaller buffers stress the shared memory queues to pass system call data.

For one OS thread, SCONE-async reaches almost the same number of system calls per second as glibc. Further scalability is likely to be possible by specialising our implementation of the lock-free FIFO queue.

5 Related Work

We discuss (i) software approaches that protect applications from privileged code, (ii) trusted hardware support and (iii) asynchronous system calls.

Software protection against privileged code. Protecting applications and their data from unauthorized access by privileged system software is a long-standing research objective. Initial work such as NGSCB [11, 43] and Proxos [57] executes untrusted and trusted OSs side-by-side using virtualization, with security-sensitive applications hosted by the trusted OS.

Subsequent work, including Overshadow [13], SP^3 [64], InkTag [27] and Virtual Ghost [16], has focused on reducing the size of the TCB by directly protecting application memory from unauthorized OS accesses. SEGO [36] extends these approaches by securing data handling inside and across devices using trusted metadata. Minibox [37] is a hypervisor-based sandbox that provides two-way protection between native applications and the guest OS. Unlike SCONE, all of these systems assume a trusted virtualization layer and struggle to protect applications from an attacker with physical access to the machine or who controls the virtualization layer.

Trusted hardware can protect security-sensitive applications, and implementations differ in their performance, commoditization, and security functionality.

Secure co-processors [39] offer tamper-proof physical isolation and can host arbitrary functionality. However, they are usually costly and limited in processing power. While in practice used to protect high-value secrets such as cryptographic keys [51], Bajaj and Sion [4, 5] demonstrate that secure co-processors can be used to split a database engine into trusted and untrusted parts. SCONE instead focuses on securing entire commodity container workloads and uses SGX to achieve better performance.

Trusted platform modules (TPM) [59] offer tailored services for securing commodity systems. They support remote attestation, size-restricted trusted storage and sealing of application data. Flicker [41] enables the multiplexing of secure modules that are integrity protected by the TPM. What limits Flicker's usability for arbitrary applications is the high cost of switching between secure and untrusted processing modes due to the performance limitations of current TPM implementations. TrustVisor [40] and CloudVisor [66] avoid the problem of frequent TPM usage by including the hypervisor within the TCB using remote attestation. This virtualization layer increases the size of the TCB, and neither solution can protect against an attacker with physical access to the machine's DRAM.

ARM TrustZone [3] has two system personalities, *secure* and *normal world*. This split meets the needs of mobile devices in which a rich OS must be separated from the system software controlling basic operations. Santos et al. [49] use TrustZone to establish trusted components for securing mobile applications. However, isolation of mutually distrustful components requires a trusted language runtime in the TCB because there is only a single secure world. TrustZone also does not protect against attackers with physical DRAM access.

As we described in §2.3, *Intel SGX* [29] offers fine-grained confidentiality and integrity at the enclave level. However, unlike TrustZone's secure world, enclaves cannot execute privileged code. Along the lines of the original SGX design goals of protecting tailored code for specific security-sensitive tasks [26], Intel provides an SDK [32, 33] to facilitate the implementation of simple

enclaves. It features an interface definition language together with a code generator and a basic enclave library. Unlike SCONE, the SDK misses support for system calls and offers only restricted functionality inside the enclave.

Haven [6] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows library OS to SGX. Relative to the limited EPC size of current SGX hardware, the memory requirements of a library OS are large. In addition, porting a complete library OS with a TCB containing millions of LOC also results in a large attack surface. By using only a modified C standard library, SCONE targets the demands of Linux containers, keeping the TCB small and addressing current SGX hardware constraints. Using asynchronous system calls, SCONE reduces enclave transition costs and puts emphasis on securing file and network communication for applications that do not use encryption.

VC3 [50] uses SGX to achieve confidentiality and integrity as part of the MapReduce programming model. VC3 jobs follow the executor interface of Hadoop but are not permitted to perform system calls. SCONE focuses on generic system support for container-based, interactive workloads but could be used as a basis for VC3 jobs that require extended system functionality.

Asynchronous system calls. FlexSC [52] batches system calls, reducing user/kernel transitions: when a batch is available, FlexSC signals the OS. In SCONE, application threads place system calls into a shared queue instead, which permits the OS threads to switch to other threads and stay inside the enclave. Moreover, SCONE uses a kernel module to execute system calls, while FlexSC requires invasive changes to the Linux kernel. Earlier work such as ULRPC [7] improves the performance of inter-process communication (IPC) using asynchronous, cross address space procedure calls via shared memory. In contrast, SCONE uses asynchronous system calls for all privileged operations, not just IPC.

6 Conclusion

SCONE increases the confidentiality and integrity of containerized services using Intel SGX. The secure containers of SCONE feature a TCB of only 0.6×–2× the application code size and are compatible with Docker. Using asynchronous system calls and a kernel module, SGX-imposed enclave transition overheads are reduced effectively. For all evaluated services, we achieve at least 60% of the native throughput; for Memcached, the throughput with SCONE is even higher than with native execution. At the same time, SCONE does not require changes to applications or the Linux kernel besides static recompilation of the application and the loading of a kernel module.

Acknowledgments. The authors thank the anonymous reviewers and our shepherd, Jeff Chase, for their valuable feedback. The work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements 645011 (SERECA) and 690111 (SecureCloud), and from the UK Engineering and Physical Sciences Research Council (EPSRC) under the CloudSafetyNet project (EP/K008129).

References

[1] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).

[2] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative Technology for CPU Based Attestation and Sealing. In *HASP* (2013).

[3] ARM LIMITED. *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

[4] BAJAJ, S., AND SION, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *SIGMOD* (2011).

[5] BAJAJ, S., AND SION, R. CorrectDB: SQL Engine with Practical Query Authentication. *VLDB* (2013).

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI* (2014).

[7] BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. User-level interprocess communication for shared memory multiprocessors. *ACM TOCS 9*, 2 (May 1991), 175–198.

[8] BREWER, E. A. Kubernetes and the Path to Cloud Native. In *SoCC* (2015).

[9] BRICKELL, E., GRAUNKE, G., NEVE, M., AND SEIFERT, J.-P. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive 2006* (2006), 52.

[10] BROWN, N. Linux Kernel Overlay Filesystem Documentation. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`, 2015.

[11] CARROLL, A., JUAREZ, M., POLK, J., AND LEININGER, T. Microsoft Palladium: A Business Overview. *Microsoft Content Security Business Unit* (2002), 1–9.

[12] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS* (2013).

[13] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS* (2008).

[14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *SoCC* (2010).

[15] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Tech. rep., Cryptology ePrint Archive, Report 2016/086, 2016.

[16] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS* (2014).

[17] CVE-ID: CVE-2014-9357. Available from MITRE at `https://cve.mitre.org`, Dec. 2014.

[18] CVE-ID: CVE-2015-3456. Available from MITRE at https://cve.mitre.org, May 2015.

[19] CVE-ID: CVE-2015-5154. Available from MITRE at https://cve.mitre.org, Aug. 2015.

[20] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[21] DONG, Y., YANG, X., LI, J., LIAO, G., TIAN, K., AND GUAN, H. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing 72*, 11 (2012), 1471–1480.

[22] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and Linux containers. In *ISPASS* (2015).

[23] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal* (Aug. 2004).

[24] GRABER, H. LXC Linux Containers, 2014.

[25] HAPROXY. http://www.haproxy.org, 2016.

[26] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using Innovative Instructions to Create Trustworthy Software Solutions. In *HASP* (2013).

[27] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS* (2013).

[28] INITIATIVE, T. O. C. https://www.opencontainers.org, 2016.

[29] INTEL CORP. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.

[30] INTEL CORP. Intel Software Guard Extensions (Intel SGX), Ref. 332680-002. https://software.intel.com/sites/default/files/332680-002.pdf, June 2015.

[31] INTEL CORP. Product Change Notification 114074-00. https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf, October 2015.

[32] INTEL CORP. Intel Software Guard Extensions for Linux OS. https://01.org/intel-softwareguard-extensions, June 2016.

[33] INTEL CORP. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk, 2016.

[34] IOZONE. http://www.iozone.org, 2016.

[35] KUBERNETES. http://kubernetes.io, 2016.

[36] KWON, Y., DUNN, A. M., LEE, M. Z., HOFMANN, O. S., XU, Y., AND WITCHEL, E. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *ASPLOS* (2016).

[37] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. MiniBox: A Two-Way Sandbox for x86 Native Code. In *ATC* (2014).

[38] LIBC, M. https://www.musl-libc.org, 2016.

[39] LINDEMANN, M., PEREZ, R., SAILER, R., VAN DOORN, L., AND SMITH, S. Building the IBM 4758 Secure Coprocessor. *Computer 34*, 10 (Oct 2001), 57–66.

[40] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *S&P* (2010).

[41] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys* (2008).

[42] MERKEL, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (Mar. 2014).

[43] PEINADO, M., CHEN, Y., ENGLAND, P., AND MANFERDELLI, J. NGSCB: A Trusted Open System. In *ACISP* (2004).

[44] PROJECT, A. H. S. https://httpd.apache.org, 2016.

[45] PURDILA, O., GRIJINCU, L. A., AND TAPUS, N. LKL: The Linux kernel library. In *RoEduNet* (2010).

[46] REDIS. http://redis.io, 2016.

[47] REESE, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* (Sept. 2008).

[48] RKT (COREOS). https://coreos.com/rkt, 2016.

[49] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *ASPLOS* (2014).

[50] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *S&P* (2015).

[51] SERVICES, A. W. AWS CloudHSM Getting Started Guide. http://aws.amazon.com/cloudhsm, 2016.

[52] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *OSDI* (2010).

[53] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS OSR* (Mar. 2007).

[54] SQLCIPHER. https://www.zetetic.net/sqlcipher, 2016.

[55] SQLite. https://www.sqlite.org, 2016.

[56] SWARM, D. https://docs.docker.com/swarm, 2016.

[57] TA-MIN, R., LITTY, L., AND LIE, D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI* (2006).

[58] THONES, J. Microservices. *IEEE Software 32*, 1 (2015), 116–116.

[59] TRUSTED COMPUTING GROUP. Trusted Platform Module Main Specification, version 1.2, revision 116, 2011.

[60] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F., ANDERSON, A. V., BENNETT, S. M., KÄGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[61] WONG, W. Stunnel: SSLing Internet Services Easily. *SANS Institute, November* (2001).

[62] A HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2, 2016.

[63] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (2015).

[64] YANG, J., AND SHIN, K. G. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *VEE* (2008).

[65] ZETTER, K. NSA Hacker Chief Explains How to Keep Him Out of Your System. *Wired* (Jan. 2016).

[66] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP* (2011).