TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitectur

Systems Support for Trusted Execution Environments

Bohdan Trach

Born on: 3rd October 1991 in Ivano-Frankivsk, Ukraine

Dissertation

to achieve the academic degree

Doctor of Philosophy (Ph.D.)

First referee
Prof. Dr. (Ph.D.) Christof Fetzer

Second referee
Prof. Dr. (Dr.-Ing.) Thorsten Strufe

Submitted on: 24th April 2021
Defended on: 3rd February 2022

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitectur

Abstract

Cloud computing has become a default choice for data processing by both large corporations and individuals due to its economy of scale and ease of system management. However, the question of *trust* and *trustoworthy computing* inside the Cloud environments has been long neglected in practice and further exacerbated by the proliferation of AI and its use for processing of sensitive user data. Attempts to implement the mechanisms for trustworthy computing in the cloud have previously remained theoretical due to lack of hardware primitives in the commodity CPUs, while a combination of Secure Boot, TPMs, and virtualization has seen only limited adoption.

The situation has changed in 2016, when Intel introduced the Software Guard Extensions (SGX) and its enclaves to the x86 ISA CPUs: for the first time, it became possible to build trustworthy applications relying on a commonly available technology. However, Intel SGX posed challenges to the practitioners who discovered the limitations of this technology, from the limited support of legacy applications and integration of SGX enclaves into the existing system, to the performance bottlenecks on communication, startup, and memory utilization.

In this thesis, our goal is enable trustworthy computing in the cloud by relying on the imperfect SGX promitives. To this end, we develop and evaluate solutions to issues stemming from limited systems support of Intel SGX: we investigate the mechanisms for runtime support of POSIX applications with SCONE, an efficient SGX runtime library developed with performance limitations of SGX in mind. We further develop this topic with FFQ, which is a concurrent queue for SCONE's asynchronous system call interface. ShieldBox is our study of interplay of kernel bypass and trusted execution technologies for NFV, which also tackles the problem of low-latency clocks inside enclave. The two last systems, Clemmys and T-Lease are built on a more recent SGXv2 ISA extension. In Clemmys, SGXv2 allows us to significantly reduce the startup time of SGX-enabled functions inside a Function-as-a-Service platform. Finally, in T-Lease we solve the problem of trusted time by introducing a trusted lease primitive for distributed systems.

We perform evaluation of all of these systems and prove that they can be practically utilized in existing systems with minimal overhead, and can be combined with both legacy systems and other SGX-based solutions. In the course of the thesis, we enable trusted computing for individual applications, high-performance network functions, and distributed computing framework, making a vision of trusted cloud computing a reality.

Acknowledgements

First and foremost, I thank my supervisor Prof. Christof Fetzer for the support and the environment without which this thesis would have been impossible. Second, I thank Prof. Thorsten Strufe for providing important comments during the status talk, and for his support as the Fachreferent of my thesis. Last but not the least, I would like to thank Prof. Pramod Bhatotia for teaching me a lot of skills indispensable in research.

I extend my gratitude to Dr. Thomas Knauth, who brought me to the Systems Engineering chair, showed me the wonderful world of systems research from the inside, and who provided invaluable feedback on this thesis. I am also grateful to Dr. André Martin for his feedback on the early drafts of this thesis, which has greatly improved its quality.

They say that the only true brotherhood is of those who have faced the same doom. I would like to thank my brothers and sisters among the past and present members of the System Engineering Chair for all the help, support, and encouragements in hard moments: Oleksii Oleksenko, Robert Krahn, Sergei Arnautov, Dmitrii Kuvaiskii, Do Le Quoc, Saidgani Musaev, Franz Gregor, Wojciech Ozga, Gabriel Pereira Fernandez, Anna Galanou, Ardhi Putra Pratama Hartono, Rasha Faqeh, Samuel Knobloch, Muhammad Usama Sardar, Thordis Kombrink, Martin Nowack, Frank Busse, Frezewd Lemma Tenna, Vesna Nowack. I have learned a lot from you. Your help will not be soon forgotten.

I express my thanks to collaborators and friends from other chairs of TU Dresden, Imperial College London, TU Braunschweig, Université de Neuchâtel, who have brought me outside of my field of study into the wonderful fields of storage and microarchitectural attacks. It has been my pleasure.

I am most thankful to Dr. Irina Karadschow for her invaluable help with the administrative hurdles.

Finally, I would like to thank my parents for providing the support and encouragement when I most needed them, as well as teaching me that education and hard work are the keys to a better future.

The work in this thesis has been supported by the European Union's Horizon 2020 research and innovation program under grant agreements 645011 (SERECA), 690111 (SecureCloud), and 690588 (SELIS), as well as by the Federal Government of Saxony.

Publications

This thesis is based on the following publications:

- SCONE: Secure Linux Containers with Intel SGX. Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, and Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch and Christof Fetzer. In proceedings of the *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
- FFQ: Fast FIFO Queue. Sergei Arnautov, Pascal Felber, Christof Fetzer and Bohdan Trach. In proceedings of *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.
- ShieldBox: Secure Middleboxes using Shielded Execution. Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. In proceedings of the *Symposium on SDN Research (SOSR)*, 2018.
- Clemmys: Towards Secure Remote Execution in FaaS. Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. In proceedings of the *12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.
- T-Lease: A Trusted Lease Primitive for Distributed Systems. Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. In proceedings of the *Symposium on Cloud Computing (SoCC)*, 2020.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Need for Trust in Cloud Computing

Internet services have gained significant popularity over the years: for example, a report by Gartner [15] shows that by 2020, up to 60% of businesses will have migrated their services to the cloud, up from 30% in 2018. This change is motivated by several appealing aspects of the cloud environment: outsourcing hardware ownership to the cloud reduces the total cost of ownership, while the economy of scale and reliance on the infrastructure management by cloud operators allows further reduction of operational expenses. To simplify the migration, companies developed several easy-to-use services, like Amazon AWS and S3, Microsoft Azure, and others [2, 1, 31, 19]. Furthermore, large companies like Oracle and SAP started to offer their software on a subscription basis [38, 41].

To better organize development of cloud applications at scale, a so-called microservice architecture was proposed. To this end, the application is partitioned into a number of small services [122], each performing one task (following UNIX philosophy) and communicating with the others using REST interfaces [124]. Today, cloud-native has become a new mainstream style of development.

In the cloud, microservices are run on a shared infrastructure managed by the cloud operator. The operator's business model depends on the oversubscription of computational resources, achieved by collocation of services of mutually untrusting and uncooperative users. To make this sharing of hardware possible, efficient and easy-to-use solutions for security and performance isolation of guests were developed. Initial solutions to this problem have relied on virtualization technologies. Virtual machines provide a high level of guest-guest and guest-host isolation, however they are increasingly falling out of popularity due to their lower usability and higher overheads [249]. Instead, more lightweight isolation and distribution *container* technologies are gaining traction, exemplified by Docker and Kubernetes. Containers do not rely on the hardware virtualization features, but run the user software on top of the host kernel, relying on the operating system namespacing features to achieve isolation and resource virtualization.

Virtualization and containerization technologies aim to thwart attacks by guests only. Yet, there are significant concerns about trust between cloud users and cloud operators. By giving up the control over the infrastructure to cloud providers, cloud users expose their software to a much larger attack surface than was the case with on-premises computing.

Indeed, NIST has identified several aspects of the cloud computing, where the trust gets conferred to the cloud operator [166], the most prominent being insider access. When the control over software and hardware is transferred to the cloud operator, the circle of the insiders having privileged access to the system expands to the cloud provider's staff and potentially to other customers of the operator. The new threat model allows potentially devastating attacks on the infrastructure, as the customer systems still consider infrastructure the trusted component. These issues are exacerbated by the ongoing migration of financial and medical companies to the cloud, as these companies hold highly sensitive and valuable user data that may increase the incentives for dishonest insiders to access the data.

Additional issues stem from the huge computing stack that forms a foundation of the cloud infrastructure. Recent analysis has identified six layers of deployment stack of typical cloud service: *hardware*, *virtualization*, *cloud environment*, *communication*, *service/application*, *orchestration* [98]. The developer targeting cloud totally relinquishes control over the former three levels and has only limited control over the latter levels. Additionally, the size of the codebases that are used at each level is extremely large. For example, the Linux kernel, which is typically used for container and VM support, has 27 million lines of code in its 5.5 release [28], with an estimate of 1 to 3 million lines of code used in typical deployments. To provide a cloud environment, solutions like Kubernetes (1.9M LoC) are applied. Studies show that the software defect density is typically 10 defects per 1K LoC [215]; at this scale, the software is bound to have exploitable errors (vulnerabilities), which can be used to gain access to the customer data.

All these issues call for efficient solutions to protect cloud users from hardware operators, the most promising of which rely on hardware isolation technologies. Initial attempts to solve these problems relied on TPM, Secure Boot, and virtualization technologies [136]. However, these techniques remained mostly unused due to restrictions of TPM [216, 109] and exploitable bugs in the underlying BIOS code [256]. In addition, these solutions did little to reduce the TCB size, still relying on large codebases and exposing significant attack surfaces. Thus, they failed to get traction in the industry, and have extremely limited software support. Despite this, Intel still maintains TXT firmware components, such as SMI Transfer Monitor technology.

Trusted Execution Environments directly target the outlined threat model: TEEs aim to provide an environment that is verifiably protected from software and hardware attack vectors available to privileged attackers. To this end, they rely on a combination of CPU-embedded primitives: hardware-implemented security monitor prevents privileged software from accessing the created environment, shrinking the TCB to the user code and TEE implementation. Remote Attestation provides the user a method of provably establishing whether the hardware protection mechanisms are active and the identity of code executing under TEE protection. Finally, optional but common cryptographic extensions provide a way to protect against advanced hardware attacks like memory bus snooping. Thus, the general availability of TEEs in commercial off-the-shelf CPUs promises to bring the security of cloud software to a new level.

However, applying TEEs in the cloud environment is challenging, as the existing TEE technologies are typically not well adapted to cloud use-cases, embodied in microservices orchestrated inside containers. The reason for these limitations is that most TEEs have their origins in the CPUs for end-user, embedded hardware. Therefore, to achieve performance, flexibility, and generality, the software and the runtime environment for trusted applications have to carefully consider the available hardware primitives and the requirements of the microservice-based applications. For example, cloud applications typically allocate signifi-

cant amounts of memory, and require efficient communication over the network to provide service to remote users. Additionally, they require support for rich functionality offered by the operating systems, which is generally not available inside the TEEs, and provide native support for execution inside containers, which are the industry standard for software deployment.

Thus, the main question that we want to answer in this thesis is "How can we utilize Trusted Execution Environments in the cloud with low overhead without sacrificing security?" In this thesis, we focus on the Intel SGX-based TEEs as the most widely available at the time of writing. To further understand the features, limitations, and requirements of the Trusted Execution Environments, we consider an example cloud Function as a Service (FaaS) deployment.

1.2 A Running Example

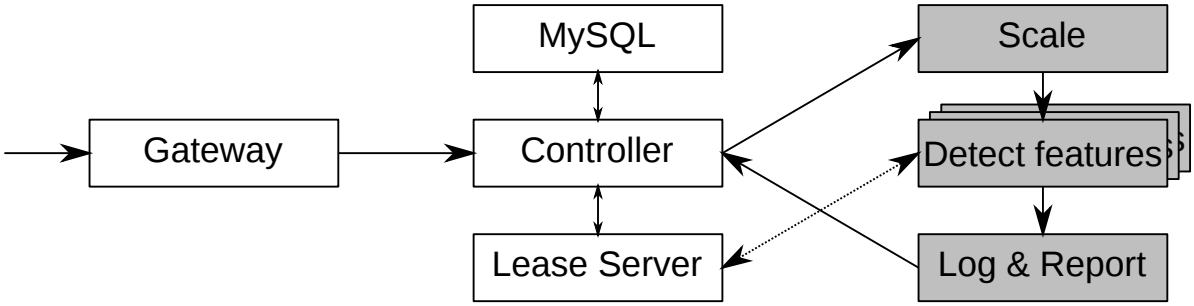

Figure 1.1: An example of a Function-as-a-Service system with Intel SGX.

We explain the challenges that an application based on trusted execution environments faces in real-world cloud deployments using an example. We use a Function-as-a-Service (FaaS/serverless) system as an example because it has recently gained traction as a novel way to deploy cloud software. It should be noted that the same challenges arise in non-serverless cloud deployments and that the results of our thesis can be applied to these use-cases as well.

The high-level overview is presented in Figure 1.1. The user requests for image feature recognition are sent to the serverless provider where user functions run. The requests with images are passed to the chain of user functions that scale down the image, pass the scaled-down image through the neural network that performs feature recognition, and log and report the results. The feature recognition function could perform optical character recognition, or detect street outlines for satellite images, etc. To achieve trustworthy operation, Intel SGX is used throughout the system.

The data processing pipeline has several steps. In the first step, the user requests enter the system via a gateway and get forwarded to the FaaS controller. The gateway implementation can include a high-performance network middlebox that monitors the traffic to the system for network attacks. The FaaS controller schedules the execution of the function on one of the worker nodes. To perform the scheduling, it inspects the message content to spawn the right function on the worker node selected for processing.

Then, the user request gets forwarded to the first processing function (downscale), which reduces the size of the images. To support a large number of input formats, this function may

be implemented as an invocation of the ImageMagick `convert` command inside a Docker container.

The next function in the chain passes the downscaled image through a neural network trained for feature detection. While this function is stateless, the user may want to limit the maximum number of functions of this kind running simultaneously, to avoid a situation where the cloud operator spawns additional instances to reverse-engineer the model by processing her own, specially crafted images.

The last function in the chain is the logging and reporting function, which may store the data in a MySQL database. The response is returned through the controller and the gateway back to the client.

1.3 Challenges and Contributions

To make the system outlined above secure and practical, several issues have to be taken care of. First, most of the currently available Intel SGX support frameworks require the development of the in-enclave software from scratch, which makes their utilization extremely complicated: demanding an SGX-aware rewrite of already available software is not a practical requirement. Thus, given that Linux is the dominant platform in the cloud, it is necessary to ensure that the SGX runtime has native support for POSIX applications. Also, software communicating over the network typically has to sustain extremely large I/O rates, while Intel SGX associates a significant cost with enclave entry and exit which are utilized for communication with the untrusted environment by default.

Another restriction of Intel SGX is important both from the performance and security point of view: the amount of physical memory that can be simultaneously utilized by the enclave is limited to 94–198 MiB in the currently available generations of Intel CPUs. Thus, to increase the amount of memory available to the user application, the runtime should have a minimally practical size. This also improves the security of the system, by reducing the Trusted Computing Base of the in-enclave application.

The second point that requires additional optimizations in the middlebox at the gateway of the system. As the gateway needs to handle traffic from multiple clients, the communication rates that are achievable via system calls may be insufficient, especially when handling small packets at line rate. To overcome this limitation, enclave applications should consider kernel bypass in their design.

The third aspect that needs to be taken into consideration is the architecture of FaaS system. A well-known inefficiency of the FaaS systems is the slow startup of functions [62, 230]. This problem is even more pronounced with Intel SGX and with complex programming language runtimes, as in this case, large amounts of memory have to be added before the enclave can start to support garbage collection. In addition, the attacker in the system has several opportunities to leak the encrypted user data. For example, if the “Log & Report” function is sending data to the unencrypted storage, the attacker has an opportunity to violate the message confidentiality by directing the incoming user message to the last function in a chain, potentially creating an information disclosure. Another FaaS architecture-related trade-off is that the controller is a component that is typically too large to run fully inside an enclave, yet it needs to decrypt the user request to route it to the right function.

Finally, the lease mechanism required to limit the number of concurrently running instances of the “Feature Detection” function must be secured from the attacker that manipulates the system time to break the guarantees provided by the leases. Bringing trust to

leases is challenging because of the significant capabilities of the attacker who can interrupt the execution of a program at arbitrary points of time, change the value and frequency of timers, and control the power management functionality of the system.

In this dissertation, we address the issues enumerated above, thus achieving secure and efficient execution of software in the cloud.

1.4 Thesis Scope and Goals

In this thesis, we propose and evaluate system components of the trusted execution environments in the context of the cloud. The scope of the thesis is limited to the runtime support aspects, ensuring that all the functionality required by the software is supported in full scale without limiting generality and performance.

Our target is to support standard POSIX applications running on top of Linux, because these applications are the most common in the cloud environments, and present generality challenges to the current TEE runtimes [44]. In contrast, small-scale frameworks for developing minimal, single-purpose applications from scratch are generally available: Intel SGX SDK, TrustZone is supported by multiple TEE frameworks, a minimal runtime (Eyrie) is available for RISC-V Keystone enclaves. However, these frameworks require significant extensions to run the applications requiring rich functionality from the host OS.

We aim to achieve three main goals with the systems created in the course of this thesis:

- Generality. Most of the software running in the cloud assumes a POSIX environment with Linux additions. Our goal is to support all applications running on top of this interface without making application-specific changes. To achieve this, we implement a generic runtime based on the standard C library, extending it with library OS-like functionality. All optimizations implemented on top of this runtime are thus available for all enclave applications.

- Performance. Practically-oriented cloud users are wary of using mitigations and security hardening features that exhibit significant overheads. Therefore, it is necessary to maintain near-native performance during the application runtime. To this end, we carefully investigate the requirements that each use-case may present to the enclave runtime together with the available hardware primitives and implement the required functionality in a way that exhibits minimal overhead.

- Security. Unlike normal applications, enclaved applications consider the interface to the OS to be untrusted. Thus, this interface must be protected from, for example, Iago attacks without applying any changes to the application. This functionality is typically implemented in the protection modules (shields). Our enclave runtime system presents the opportunity to implement shielding solutions on top of it. Also, applications implemented on top of our runtime must not suffer from decreased security stemming from design decisions necessary to achieve the first two goals.

1.5 Contributions

To fulfill the goals, we design and evaluate four cloud systems that utilize Intel SGX for achieving trustworthy computations in the untrusted environment: SCONE, ShieldBox, Clemmys,

T-Lease. Each of these systems applies trusted execution technology (Intel SGX) to a new level of cloud architecture: starting with the support of individual POSIX applications in SCONE, we move to the problem of trustworthiness of network middleboxes (ShieldBox), and then finally use Intel SGX to secure a Function-as-a-Service architecture (Clemmys) and leases in the distributed systems (T-Lease).

SCONE. We start this thesis by building the Intel SGX runtime for unmodified POSIX applications inside enclaves. The aim is to allow the wide variety of the commodity off the shelf, well-tested software to utilize Intel SGX protection without requiring any modifications. To achieve a pragmatic trade-off between the TCB size and the supported functionality inside the enclave, we use neither the minimal TCB approach nor the library OS approaches to the SGX runtime. Instead, we implement SCONE as a minimal extension to the standard C library, using system calls as the communication interface between the enclave and the untrusted system. This architecture, coupled with the asynchronous communication and in-enclave execution of only the essential system calls allows SCONE to achieve both high performance and generality. We demonstrate this by evaluating the software commonly deployed in the cloud: Apache HTTP server, Nginx, and Redis, and show that they operate with acceptable overhead.

ShieldBox. After building SCONE, we turn our attention to the problem of building trustworthy network middleboxes. To process network traffic at the line rate, middleboxes require a significantly more efficient communication interface than is achievable with the SCONE system call interface even after the optimizations. To overcome this restriction, we rely on the kernel bypass and directly control the NIC from the enclave, which allows ShieldBox to achieve near-native throughput. To implement ShieldBox, we combine the Click modular router with the DPDK (Data Plane Development Kit) userlevel networking framework, running both components on top of SCONE. Additionally, we implement Intel SGX-specific optimizations that further reduce the SGX-induced latency stemming from the lack of a fast and trustworthy time source inside SGX enclaves. Our evaluation shows that after the optimization, ShieldBox can operate at wire speeds and with near-native latency.

Clemmys. To bring trust to a higher-level cloud infrastructure, we turn our attention to the Function-as-a-Service (serverless) platforms, which are a rising trend in cloud computing. As it is impractical to run all of the FaaS platform components inside SGX enclaves, there is a necessity to develop a message format and a key management scheme that will allow running only the gateway and the user functions inside enclaves. Additional problems stem from the FaaS requirement of having minimal application startup time, which conflicts with the reality of running functions implemented in scripting, GC-enabled languages that require a large heap for operation. To overcome this challenge, we rely on Intel SGXv2 extensions, that allow adding virtual memory to the enclave after it has been initialized. We implement Clemmys using the OpenWhisk FaaS platform and measure its performance, showing that it has minimal performance overhead compared to the native variant as long as the hardware limits are not reached.

T-Lease. After designing Clemmys, we note that the missing component of trustworthy distributed systems is *trusted leases*. Indeed, as all timers exposed by the hardware to the enclave are untrusted, a naive implementation of leases on an untrusted platform is bound to be vulnerable. We overcome this issue by building on Intel SGXv2, which extends the timestamp counter support inside the enclave with Intel TSX and a novel CPU frequency verification routine. To demonstrate the practicality of the resulting system, we extend several state-of-the-art distributed protocols with trusted leases and show that T-Lease adds

minimal overhead to such systems.

2 Background

This chapter introduces the conceptual foundation of our research and the design space of the proposed solutions. We start by explaining the concepts related to trusted execution and trust in computing systems. Then, we briefly explain the available hardware-assisted trusted execution technologies. To finish this section, we give an overview of the main contributions of the state-of-the-art TEE systems.

2.1 Concepts

To introduce the concept of trusted computing, it is necessary to first clarify what is trust in the context of computing systems. Generally, trust is defined as [52]:

> Firm belief in the reliability, truth, or ability of someone or something; confidence or faith in a person or thing, or in an attribute of a person or thing.

In the context of computer science, a system or a component is said to be *trusted* if its failure *can* lead to the security failure of the system. A component that is proven to never fail in the way that compromises the system security is called *trustworthy*. Based on these concepts, we can also define a *Trusted Computing Base* (TCB) of a system as a set of all trusted components in the system: correct operation of these components is sufficient to ensure that the system does not experience a security failure. The NIST definition states that TCB of a system is the *"Totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination responsible for enforcing a security policy."*

For security-critical computing systems, the standard way of ensuring trust has historically been Common Criteria certification, which assigned an Evaluation Assurance Level (EAL) to the evaluated piece of software. There are seven Common Criteria levels, with assurances ranging from minimal examination of the software with its specification and documentation to comprehensive formal verification. However, this certification is practical only in situations where the certified software does not change, which is not the case with the common user software, as subsequent changes to the certified software typically invalidate the certification, and more importantly, is small enough to be amenable to verification. In practice, cloud applications rely on millions of lines of code of the Linux kernel, hypervisors, cloud management software, and so on.

In the cloud, a more practical approach is relying on the novel hardware isolation features of the CPU embodied in the Trusted Execution Technologies, which provide hardware primitives for managing Trusted Execution Environments. The main goal of a Trusted Execution Environment is to create an isolated environment on the untrusted host computer; the user can be assured that the security guarantees of the computations would hold inside the TEE. These guarantees include confidentiality and integrity of code and data executing inside the enclave, but may also be extended to the freshness of on-disk data, confidentiality and integrity of communication, and so on. The brief definitions of these terms are given below:

- *Confidentiality* – The property that sensitive information is not disclosed to unauthorized entities [80].
- *Integrity* – A property possessed by data items that have not been altered in an unauthorized manner since they were created, transmitted, or stored [79].
- *Freshness* – A property that states that replays of old messages/data items are prevented or impossible.

To ensure that these properties hold in practice, a *remote attestation* procedure is typically provided. It is used to produce a remotely verifiable, cryptographically signed statement about the isolation and security properties of the TEE.

Historically, Trusted Execution Environments lacked a specific definition [129, 133, 237, 289, 257]. The most complete and comprehensive definition is provided in [257]:

> Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties. The content of TEE is not static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible.

It should be noted that this definition has an extremely strong threat model, which requires defense against all software attacks and extends the defense to the persistent memory. In practice, the TEE primitives provided by the CPU aim to prevent direct reads and writes to the TEE memory and the register state, and implementing on-disk *sealing of data*, while using these primitives to secure the application and its persistent data to withstand attacks is the responsibility of the TEE application developer. For example, supply chain attacks are an attack vector that only rarely gets solved in the context of TEE technologies [222].

An important attack vector, commonly ignored for the standard applications, but which must be considered in the case of TEEs, is *Iago attacks*. The idea behind Iago attacks is that the operating system is actively trying to break the behavior of user applications by providing incorrect return values for the application system calls. More specifically, these return values break invariants on which the application relies: for example, that return value of `mmap` system call points to the previously non-allocated memory. If the application accepts a pointer to the memory that was previously allocated, it could overwrite some of its data currently in use, leading to data loss, denial of service, or remote code execution vulnerabilities.

Recent breakthroughs in microarchitectural attacks, exemplified by side-channel attacks [263] or transient execution attacks [180, 204, 93, 94] have also been used to successfully attack TEEs. These attacks rely on the fact that TEE execution leaves traces in shared CPU structures, for example caches, and deduct the in-TEE data by timing the accesses to these structures. Mitigating side-channel attacks requires a combined hardware-software approach: for example, TEE hardware vendors can allow attesting hyperthreading state to remove concurrent access to certain CPU structures and fix microcode deficiencies to improve fault processing. The TEE developer should rely on constant-time programming [8], attack detection techniques [233, 139], fuzzing [235], and formal methods and language- and compiler-based techniques [81, 116] to improve the resilience of TEEs to these attacks. In general, these attacks are out-of-scope for this thesis.

2.2 Brief History of Trusted Execution (in the Cloud)

The trust requirements to the software and the corresponding secure remote execution problem did not appear with cloud computing but dates back to the past. The survey of key academic and industrial solutions to the secure remote execution problem is given by Costan et al [104].

Early systems. Arguably, one of the first examples of relying on trusted hardware for the computations was the IBM 4758 cryptographic module [276], which allowed secure processing of banking data without trusting the software stack running on the computer. However, it was not implemented as part of the host CPU, and relied on custom hardware for this purpose.

Several examples of relying on trusted hardware to simplify the design of the distributed systems come from the field of academic research. For example, TrInc relies on a small trusted counter module to simplify the design of Byzantine fault tolerance protocols [200]. Flicker describes a way to significantly reduce the TCB of the software by relying on the late launch and attestation technologies, TPM as a hardware root of trust, and virtualization as an isolation technique. Multiple TPM-based systems were proposed [240, 239].

Intel TXT and Trusted Computing Initiative. One of the first examples of trusted execution technology built into the COTS CPUs was Intel TXT (Trusted Execution Technology), developed by Intel as a part of the Trusted Computing Initiative. The idea of Intel TXT was to boot a verified, trusted operating system alongside the host OS. The communication between the trusted and the untrusted OS would be subject to control by the minimal, formally-verified hypervisor. However, as we have mentioned before, Intel TXT had deficiencies that prevented the use of this approach in practice: TPMs cannot be migrated in the cloud without special extensions, the boot protocol for trusted OS is complex and relies on the bug-free BIOS implementation, which is not a valid assumption in practice [256].

ARM TrustZone. ARM TrustZone is a set of ISA extensions introduced in 2004 to ARM-A and ARM-M CPU architectures. Its main features are the separation of computing resources into two partitions: a *normal*, non-secure one, and the secure partition, both called *worlds*. The context switch between the partitions happens on interrupt delivery or the explicit request. The operations executed during the context switch are controlled by the *security monitor* software. The *secure world* has the same software structure as the normal world, with a separation into kernel mode and user mode, with an ability to run the hypervisor inside.

While the security monitor is responsible for the partitioning of CPU resources (time-slicing), additional optional components allow system developers to partition the RAM and system devices, by assigning them to either of the worlds. RAM and SRAM are partitioned by TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA) correspondingly, by marking it as secure or normal. Software running in the secure world can access either of the memory types, while the normal world can access only the normal memory. TrustZone Protection Controller (TZPC) allows assigning devices to secure or insecure worlds. TZASC, TZMA, and TZPC are optional components, and their configuration and interface is implementation-defined.

ARM TrustZone can be used for implementing TEEs: this is achieved by implementing a required functionality in the application running on top of the operating system in the secure world. There are several standards and commercial implementations of TrustZone-based TEEs: Komodo, OpTee, SierraTEE, Samsung KNOX, and others [121, 35, 42, 40]. These TEEs are commonly found on end-user devices like mobile phones or set-top-boxes (for example, for implementing TPM-like or DRM functionality), and see little use in the cloud.

Intel SGX. Intel SGX is a Trusted Execution Technology introduced in 2015 in the Skylake generation of Intel CPUs. It is implemented as an ISA extension that introduces the concept of *enclave* and a corresponding execution mode. Enclave is a hardware-protected range of memory that contains application data and code, running in a special execution mode. Enclave memory is protected against accesses from non-enclave software, thus achieving confidentiality and integrity, and must be allocated from the secure physical page set, called Enclave Page Cache (EPC). Enclaves run as a part of a userspace process, simplifying the OS support for the enclaved software.

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. These secure pages are called Enclave Page Cache (EPC) and has a limited size. SGX prevents cold boot attacks and snooping on the memory bus using the on-CPU Memory Encryption Engine (MEE), which encrypts all DRAM traffic between enclaves and RAM while ensuring freshness and integrity of the encrypted in-DRAM data by computing a Merkle tree over cache lines. The depth of the Merkle tree limits the maximum amount of physical memory that the enclave can use without secure memory paging. To prevent chosen cyphertext attacks on the in-MEE encryption keys, MEE locks up the memory controller if an integrity check fails. This way, enclave memory has confidentiality, integrity, and freshness guarantees – all memory modification and rollbacks are detected.

Enclaves are subject to several restrictions: several instructions are prohibited inside the enclaves, most notable `syscall`. These instructions cause an illegal instruction exception inside the enclave and have to be handled outside of the enclave.

The memory reads and writes for in-enclave software are still subject to virtual memory permission checks controlled by the OS, however the reads from untrusted mode to the enclave physical memory are short-circuited to return zeros. Memory reads and writes from enclave mode to the untrusted memory are allowed, to pass function call arguments and results. The only exception is the instruction fetches from untrusted partition, as these fetches are forbidden inside the enclave mode. This prevents ROP attacks where the instruction pointer in the enclave mode would be pointed to the untrusted memory. It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data.

Additionally, the amount of physical memory that can be simultaneously mapped into the enclave is limited, to 94 MiB on CPUs starting with Skylake (2015), while starting with Coffee

Lake generation (2019) the EPC has been increased to 198 MiB [25].

The enclave lifecycle relies heavily on the ISA extensions. During system boot, BIOS dedicates a range of physical memory for the enclave memory and SGX bookkeeping data structures. Enclaves are created by untrusted code using the `ECREATE` instruction, which initializes an *SGX enclave control structure* (SECS) in the EPC, using an OS-provided range of virtual addresses that will be dedicated to the enclave. 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 this, a *EINIT token* must be created. EINIT token certifies that the given enclave can be launched with the requested security properties. As these properties involve capabilities like disabling debugging and performance counters during enclave execution and access to sensitive platform keys necessary to implement attestation, an enclave is verified by the *launch enclave* in the process of EINIT token creation. The launch enclave implements the policy which controls the properties available to the newly created enclave.

When an enclave is no longer needed, the OS can deallocate it by issuing a sequence of the `EREMOVE` instructions, which will remove all of the enclave's data pages and internal control structures.

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 *Thread Control Structure* (TCS) along with a dedicated *State Save Area* (SSA).

When an interrupt gets delivered to the in-enclave application, or when it generates an exception, the so-called *Asynchronous Enclave Exit* (AEX) procedure takes place. It saves the enclave state to the State Save Area, replaces the register values with the synthetic values to avoid information leaks. At this point, the thread leaves enclave mode, and the control is transferred to the kernel according to the x86 architecture. In case the kernel generates the signal to the userspace, the control is transferred to the userspace signal handler, which may optionally reenter the enclave to trigger in-enclave signal handling. Otherwise, the kernel transfers control to the AEX handler passed to the `EENTER` instruction. The AEX handler, which typically consists of a single `ERESUME` instruction, reenters the enclave and restores the state from SSA, and continues the execution.

In case the application attempts to access the memory that is currently not mapped into the enclave, SGX firmware will perform the paging of the memory with the assistance of the operating system. As Figure 2.1 shows, this process causes significant performance degradation [71].

To allow the remote user to verify the identity of the SGX software, several versions of the attestation protocol were developed. The initial version was oriented towards checking the identity of the software running on the end-user computer [168], while the latest version is oriented towards attestation of cloud software where maintaining anonymity throughout the attestation process is less of a concern [261].

SGXv2 (documented in 2015, first made available in 2019 in Gemini Lake mobile CPUs) alleviates several of the original restrictions of SGX:

- `rdtsc` and `rdtscp` instructions are no longer forbidden.
- Page permissions can be changed dynamically during the application runtime.

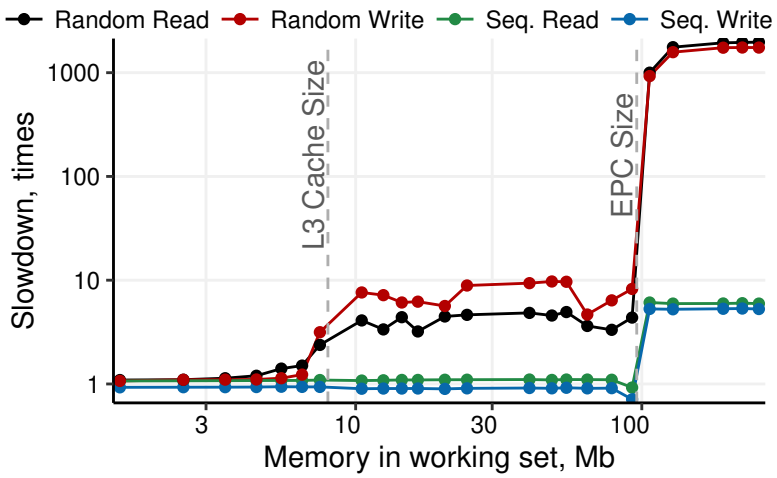

Figure 2.1: Influence of the working set size and memory access pattern on the performance of an SGX enclave.

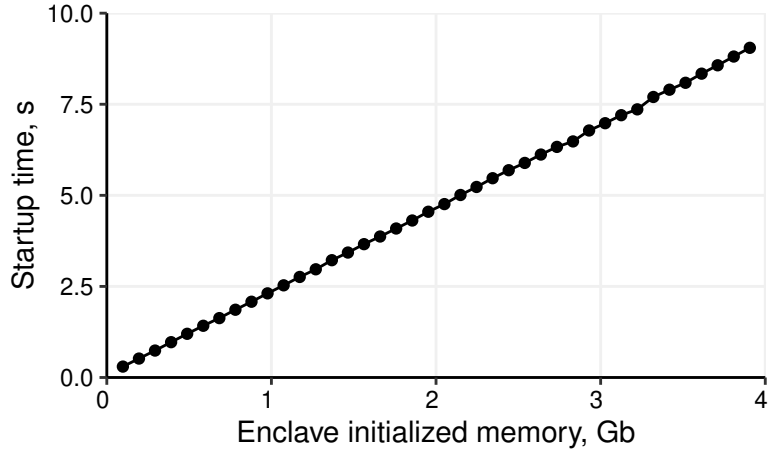

Figure 2.2: Influence of the enclave heap size on the SGX enclave startup time.

- Physical memory that was not allocated to a virtual address during enclave startup can now be added to the enclave through the cooperation of the enclave and the OS.

The dynamic physical memory allocation feature allows delegating the initialization of enclave memory to the runtime. This is critical for deployment of enclaves in scenarios which require the processing of large amounts of data with low latency, because otherwise, enclave initialization time is directly proportional to the enclave heap size (Figure 2.2).

SGX Oversubscription Extensions for easier support of SGX enclaves in virtualized environments were announced in 2017, but are not available in the CPUs yet. The reason for their introduction was significant overhead associated with tracking SGX-related state inside virtual machine monitors, and concurrency issues when both the VM and the VMM perform service operations on the enclave memory. To eliminate these overheads, Intel has added the extensions that allow VMM to query the metadata of an EPC page, and pinning, unpinning, and tracking of Secure Enclave Control Structure (SECS) page. This mechanism allows implementing SGX support in VMMs at a much lower overhead.

Another SGXv2 extension is Flexible Launch Control (FLC). Without FLC, only the enclaves authorized by Intel may be launched in the non-debug mode. Flexible Launch Control allows creating custom architectural Launch Enclaves, which then can allow starting other enclaves with arbitrary properties (access to architectural keys or production, non-debuggable mode).

AMD SEV. AMD Secure Encrypted Virtualization (SEV) is an extension to AMD virtualization technology introduced in 2016. It relied on the off-core Platform Security Processor (PSP, implemented as an embedded ARM core) to encrypt the memory of the virtual machines running on the platform, thus preventing attackers (malicious insiders or outside hackers) from accessing the content of the VM memory. A separate extension SEV-ES (Encrypted State) additionally protects the register state of the VM guests during the interrupts [172]. In this context, the Trusted Execution Environment is formed around the executing virtual machine.

PSP is used in SEV to implement all the key features:

- A remote attestation protocol that allows the remote user to verify that the user-provided image has been correctly launched under SEV protection.
- A secure channel establishment protocol that allows users to launch the VM images or the cloud operator to perform VM migrations.
- All key management features and control of memory encryption modules is implemented as part of PSP software.

AMD SEV has been a target of several attacks: in 2018, a limitation of encryption without integrity protection used in SEV was exploited to completely bypass SEV protection [224, 73]. Additionally, several vulnerabilities in the PSP OS were discovered that allowed arbitrary code execution on the PSP, thus allowing the deactivation of PSP without affecting the attestation state [92].

To address the vulnerabilities stemming from the missing integrity protection, AMD introduced SEV-SNP extensions (Secure Nested Paging) [53]. It allows establishing virtual-physical page mappings that are verifiable by the SEV guest. Additionally, these extensions allowed dividing the VM address space into four hardware isolated abstraction levels, provided functionality for protection from malicious interrupts, and improved the attestation and key derivation procedures.

It should be noted that in addition to SGX, Intel has developed a technology similar to SEV memory encryption primitive, called Multi-Key Total Memory Encryption (MKTME) [162]. It offers weaker guarantees than SEV as it does not enforce integrity and does not support attestation. It is expected that starting from the Ice Lake generation of Intel CPUs, the TME implementation will substitute the current implementation of Memory Encryption Engine, thus allowing developers to create enclaves of unlimited memory size while dropping the freshness protection [258].

It is expected that TME technology will be used to implement another Intel ISA extension, called Intel Trust Domain Extensions (TDX) [163, 24]. It aims to protect the Virtual Machines from the privileged and unprivileged attackers using primitives similar to those implemented in SEV. Intel TDX introduces a special execution mode, called Secure Arbitration Mode, in which a trusted module with management routines (called Intel TDX module) for interaction between the VM and the VMM will be located. Intel TDX module is located in a reserved range of memory, inaccessible to both privileged and unprivileged software as well as DMAs from the devices. To install the Intel TDX module, an Intel TXT-based loader is used. The VMM can

use a special instruction, SEAMCALL, to invoke the functionality in the TDX module, such as create, launch, and resume VMs. Intel TDX relies on the Intel TME to encrypt the VM memory, but will also provide integrity in addition to confidentiality to the memory pages. The VMM must maintain two Extended Page Tables (EPT), the Secure EPT and the Shared EPT, to allow the communication between the VM and the VMM, which are both further translated by the Guest Page Table inside the VM. To control page mapping and TLB states, a data structure similar to that employed by SEV is used. For remote attestation, Intel intends to use the Intel DCAP library along with a special Intel TDX Quoting enclave. Unlike Intel SGX, which hides the security-critical functionality inside the microcode, TDX presents a more open design, by allowing open and third-party-verifiable implementations of the security monitor, as proposed by Komodo [121].

RISC-V Keystone Enclave. Keystone Enclave is a proposed implementation of Trusted Execution technology for RISC-V CPU architecture. To this end, it fully relies on the RISC-V execution mode for firmware (Machine, or M-mode), and on the standard (albeit optional) security primitives available in RISC-V architecture: (I/O) Physical Memory Protection (PMP and IOPMP) [197].

To implement Keystone, M-mode software runs a security monitor software, that configures the hardware protection primitives and facilitates context switches between the untrusted world and enclaves. Security monitor uses Physical Memory Protection feature to restrict the range of physical addresses that the higher-level software, including the OS and the hypervisor, can access. IOPMP, on the other hand, restricts DMA to physical memory ranges, preventing malicious devices from accessing the enclave memory. This implies that M-mode software is a trusted component.

Inside the Keystone enclave, the system structure mirrors the normal system structure - that is, there is an operating system component (enclave runtime), on top of which the actual enclave software is executing. This architecture is similar to the one of the Trustzone and AMD SEV.

Nitro Enclaves. Nitro enclaves are AWS implementation of the TEE technology [6]. From the point of view of developers, Nitro enclaves are significantly restricted virtual machines running on dedicated hardware (without co-tenants), without access to persistent storage. The Nitro enclave communicates with an untrusted runtime in its parent VM via a virtual PCI device. The enclave runs with its own kernel and minimal userspace, initialized at the enclave creation time from an EIF (Enclave Image Format) file. The content of this file is measured by the hypervisor and the security chip implementing the root of trust, and can be used as an attestation record by other AWS services, most importantly by AWS KMS [29].

AWS Nitro enclaves present a slight deviation from the standard TEE trust model. The TEEs provided by Intel and AMD deployed in the cloud make it possible to assume malicious cloud providers, as it is unlikely that the cloud provider can easily compromise the TEE primitives implemented in the CPU by an independent vendor. Nitro enclaves, on the other hand, still require the trust in the cloud provider, as the chip implementing the root of trust is also developed by Amazon.

Conclusions. We can see the trend of having Trusted Execution technologies available in the commodity CPUs, due to the necessity to perform trusted computing in untrusted environments.

However, there are several problems associated with Trusted Execution technologies:

- All of these technologies provide only hardware primitives and an extremely limited runtime, that is insufficient for serving the rich functionality expected by cloud soft-

ware from the host operating system. Today, most cloud software expects to be built and run as a POSIX application, while default runtimes of the TEEs provide only minimal interfaces or even do not provide any runtime at all, making migration of existing software into the TEE challenging. The only exception is AMD SEV that transparently supports launching the existing VMs under SEV protection.

- While container technologies have become a dominant way of deploying software in the cloud, trusted execution technologies have to adapt to this to be usable. However, out of the presented TEEs, only Intel SGX supports transparently running inside containers out-of-the-box; other technologies require significant modifications to their runtimes to bridge the gap between available hardware primitives and the semantics of container interfaces, such as extending Kata containers for AMD SEV support [250].
- Each of these technologies has its own performance-security trade-offs. The most significant example is Intel SGX, which provides significant security guarantees at the high performance and flexibility cost: enclave entries and exits, interrupts, and secure memory paging all cause significant performance overheads.
- It may be impossible to secure the functionality necessary for some use-cases, or it may be generally not available inside the TEE.

In our thesis, we tackle the problems that affect Intel SGX enclaves, while outlining the potential solutions for other trusted execution technologies as future work. This choice is motivated by the general availability of SGX-enabled CPUs in the field of cloud computing during the thesis work period (2016-2020).

2.3 Intel SGX Runtimes

In this section, we explore the design space and the associated trade-offs for building TEE systems for Intel SGX. SGX runtime library supports the operation of an application, by providing functionality in three core categories: memory management, thread management, and communication. The choice of SGX runtime for the application depends on the requirements of the application in these categories, but also on the standard API that the application targets, such as POSIX or Windows.

Memory management, threading, and I/O are typically handled by the operating system in the cooperation with the userspace. Hovewer, due to the changed threat model, this functionality must be implemented fully in userspace inside the enclave. Memory management must be provided inside SGX runtime because the corresponding OS functionality is unavailable inside the enclaves, to protect the enclave against the memory-based attacks. Thread management must be implemented inside the enclave and inside the untrusted runtime likewise, to protect against the attacks which involve malicious thread scheduling, and to overcome SGX limitations. And the mechanisms for communication with the untrusted world are necessary to submit requests to the untrusted environment and return the computation results: the untrusted world has no access to the trusted enclave memory. Different SGX runtimes implement these core systems aspects inside enclave to a different extent, but minimal level of functionality must be provided by all runtimes.

Another important distinction between the SGX support frameworks can be drawn based on their tradeoff between TCB size and the communication overhead. From this point of

view, all Intel SGX support frameworks broadly fall into one of the three categories: *approaches base on a library OS* and the *approaches with minimal TCB* take extreme choices with regard to the enclave code and data, while *partitioning-based approaches* try to achieve a more balanced solution.

Library OS approaches rely on a custom-built or existing library OS, and aim to support rich functionality applications by emulating an existing OS (for example Linux, NetBSD, or Windows) to the extent, to which this is possible. The application communicates with the library OS with the assistance of a C library modified to use standard function call calling convention instead of trapping instruction like `syscall`.

Minimal TCB frameworks aim to increase system security by minimizing the code size inside the enclave at the expense of the functionality of the applications. For example, such frameworks may contain only the simplest memory and thread management infrastructure, coupled with a custom or predefined communication infrastructure.

Partitioning-based approaches employ a (semi-)automatic partitioning toolset for splitting the application into the trusted and untrusted parts. The fundamental assumption is that this way, both the TCB and the communication interface can be minimized, without incurring a significant performance overhead on the unprotected part, and without a requirement to perform application reengineering. To guide the partitioning tools, typically the programmer has to provide only a few annotations on the variables that must be protected.

Thus, these three approaches made different trade-offs between the following two fundamental aspects:

- TCB size: the larger the TCB size, the more functionality can be supported inside the enclave, but this potentially decreases the trust in the system. By minimizing the TCB size, the application becomes easier to audit and verify, allowing increased trust in the application, but limits the functionality inside the enclave and potentially increases the communication interface size.

- Communication interface: larger TCB allows minimizing the interface to the untrusted system, and thus reducing the attack surface by implementing more functionality inside the enclave. On the other hand, delegating the handling of rich functionality to the outside of the enclave reduces the TCB without causing incompatibilities, but makes the interface a lot larger. Additionally, custom-developed interfaces do not suffer from both incompatibility and TCB size drawbacks, but they are not general and must be developed anew for each application.

2.3.1 Library OS based approaches

Haven. Haven was the first system aiming to provide support for rich applications inside Intel SGX enclaves [83]. To this end, it relies on the Drawbridge library OS that emulated a full Windows 8 API, while requiring the developer to implement only 22 hypercalls from the enclave to the host OS. These hypercalls broadly fall into the following categories: memory management, thread management, signal handling, stream I/O, and system time access. Thus, Haven strived for minimal host OS interface and maximal functionality inside enclaves.

An important contribution of Haven is the concept of *shielding*: a reverse sandbox that instead of protecting a trusted environment from an untrusted piece of code, aims to protect a trusted piece of code from the untrusted environment. To this end, Haven introduced shielding modules: as enclaves hypercall interface is untrusted, Iago attacks must be detected and

prevented. Shielding modules implement comprehensive validation of return values from the OS [100].

A characteristic feature of Haven is its huge TCB size: the paper quotes 209 Mb of code, which is much more than the amount of EPC memory even on the latest of SGX-enabled CPUs. Thus, this system cannot be practically utilized. While the authors quote numbers plausible for production use in their evaluation, it is important to remember that it has been performed in an emulator, which does not model the performance degradation from the EPC paging accurately.

Graphene-SGX. Graphene-SGX conceptually follows the design of Haven, but was built to support POSIX instead of Windows applications [287]. It builds on Graphene library OS, which is used to implement a wide range of Linux-compatible functionality [286]. The interface to the host OS is similarly small — Graphene-SGX requires only 41 hypercalls to provide Linux application support on an untrusted Linux[1] host [34].

The emulation of the Linux system call interface inside Graphene-SGX comes at a high cost — the Linux system call interface is extremely hard to faithfully replicate, and the minor incompatibilities that thus arise make application support and debugging extremely hard. To enable many common applications, Graphene-SGX has to be constantly modified and extended to provide feature-by-feature support of Linux system calls (including obscure parts of the Linux kernel, e.g. signal dispositions and flags like MSG_PEEK). This proved to be a tedious affair.

From the performance point of view, evaluations show that Graphene-SGX has approximately 40% overhead compared to the native versions of real-world applications with a significant rate of system calls. As with other frameworks, these overheads are lower for mostly CPU-intensive tasks that fit into the EPC.

SGX-LKL. An alternative approach to the supporting Linux programs inside enclave was taken in SGX-LKL [247]. Instead of emulating Linux functionality, SGX-LKL uses a version of the Linux kernel built as a library OS for a NOMMU architecture (LKL, Linux Kernel Library). This approach allows SGX-LKL to support POSIX binaries without compatibility issues.

SGX-LKL takes further measures to minimize its API to the untrusted system. The external interface is limited to only 7 calls, which comes at the expense of the ability to directly communicate with the untrusted OS via system calls: SGX-LKL supports working only with the file system images on disk and with emulated network devices. This interface is extensively confidentiality- and integrity-protected, and takes significant measures to prevent leaking information about the behavior of the application via its external interfaces.

The evaluation shows that the overhead of SGX-LKL in communication-heavy cases is between 170% and 190%. This significant overhead may not be acceptable when running in the cloud environment, and the lack of integration with the files located on the host OS makes container support challenging.

2.3.2 Minimal TCB Systems

Intel SGX SDK. Intel SGX SDK is the official framework released by Intel for the development of Intel SGX enclaves [157]. It is based on the modified OpenBSD libc, extended with a set of cryptographic libraries and libraries for access to Intel SGX functionality, for example, remote attestation and sealing.

[1] Unlike Graphene-SGX, Graphene can run Linux applications on top of other operating systems.

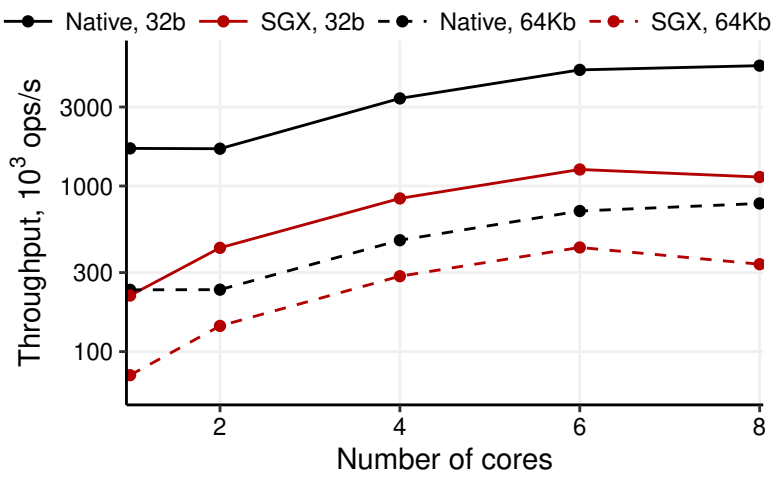

Figure 2.3: Influence of the I/O buffer size and concurrency factor on the I/O throughput of a native POSIX application and Intel SGX enclave.

For communication with the untrusted OS, Intel SGX SDK provides an IDL-based interface generator, that reads user-provided interface description, and uses it to generate the code for entering and exiting enclave, transferring and validating arguments, and their wrappers in the trusted and the untrusted worlds—so-called enclave calls (ECALLs) and outside calls (OCALLs). The drawback of this interface is its performance overhead, which can reach 8x for the system call heavy applications (Figure 2.3). Additionally, context switches between trusted and untrusted worlds can last up to 5000 cycles, further decreasing performance with a TLB flush.

To reduce the context switching overhead, Intel has developed switchless call infrastructure, that relies on an asynchronous queue to pass OCALL arguments to the untrusted world and receive the response inside the enclave, and a thread pool outside of the enclave to read the arguments and execute the request. However, this approach still lacks the generality necessary for supporting general POSIX applications.

Additionally, Intel used to provide (until version 2.8, February 2020) a Platform Services Enclave as a part of Intel SGX SDK, which provided access to the hardware functionality of Intel Management Engine (ME), such as monotonic counters and trusted time source. However, this functionality has been deleted in the further version of SGX SDK, without providing any alternative implementation of these APIs. In practice, developers can use TPMs to implement these features.

Panoply. Panoply is a minimal TCB system for running unmodified POSIX applications [272]. It is based on the two main concepts: delegation of system calls to the host OS kernel and partitioning of application into multiple enclaves (called *microns*) for fault isolation. To ensure that the communication between multiple enclaves of one application is secure, Panoply proposes a protocol that preserves the integrity of messages and the control flow inside the enclave based on the received messages. Additionally, Panoply strives to hide the limitations of SGX from the software, for example, it has provisions for supporting an unlimited number of threads.

For the delegation of system calls and standard C library functions, Panoply uses wrapper functions generated using Intel SGX SDK. Panoply uses a sophisticated shielding module to protect against Iago attacks. Panoply also supports `fork` and `exec` system calls inside the

enclave, thus allowing it to run a wider range of POSIX applications than other platforms. While typically these system calls require a complex implementation to faithfully recreate the OS-provided semantics with respect to state that gets inherited by the child or replacing process, Panoply significantly simplifies their implementation by delegating as much state as possible to the OS.

While the TCB of Panoply is only 20 kLoC, its untrusted API has comparably large size—it includes 254 API endpoints, that is system calls and standard C library functions. Panoply does not provide a toolchain for automatic partitioning the application between different enclaves and delegates this responsibility to the developer. However, the partitioning cost is low as the developer has to only annotate the application source code with pragmas.

Asylo. Asylo is a framework for writing TEE-independent applications: the applications built with Asylo are independent of the *security backend* used by the platform, whether software or hardware one [4]. The software backends may be implemented as virtual machines, while hardware backends would use Intel SGX or AMD SEV. This promises to simplify the deployment for the user and reduce the development effort for the application provider.

For communication, Asylo offers an extensive interface to the untrusted operating system based on delegation similar to the one used in Panoply. For network communication, Asylo offers a framework for implementing gRPC endpoints. To secure the data in transit, it provides a cryptographic protocol where channel establishment includes mutual attestation. In addition, APIs for sealed data access and for access control are provided.

RATEL. Another system that aims to support unmodified POSIX applications is RATEL [271]. Unlike other systems, which require relinking or recompiling the application to run it inside of the enclave with minimum modifications, RATEL develops this research direction further and allows running precompiled unmodified applications inside the enclave. To this end, it relies on the dynamic binary translation system (DynamoRIO), which identifies binary code sections which are incompatible with the restrictions of Intel SGX, and rewrites them with calls to the host operating system, to the in-enclave runtime, or to the untrusted runtime.

While this approach adds the dynamic binary translation engine to the enclave TCB, it relies on the extremely lean runtime after translation—a minimally-extended Intel SGX SDK, thus allowing us to classify it as a minimum TCB system.

2.3.3 Partitioning Approaches

Glamdring. Glamdring is a toolchain for automatic partitioning of user applications for Intel SGX enclaves [203]. Glamdring combines static program analysis techniques with user annotations to automatically partition the security and performance-sensitive components of the application into the enclave and unprotected parts correspondingly. This approach reduces the TCB by pushing the code that violates the principle of least privilege to the untrusted world while taking the performance implications into account.

The user has to specify sources and sinks of sensitive data, that is variables or function arguments, through which the communication with the untrusted world is happening. The static code analysis employed by Glamdring identifies all statements in the program that depend on the annotated lines, and ensures confidentiality by performing static dataflow analysis and integrity using backward slicing approach. Additional functions are moved to the enclave if this improves performance using `gcov` tool.

To protect the communication interface, Glamdring analyses the static invariants of the untrusted code and transforms them into the runtime checks inside enclaves, and verifies

pointer destinations between the trusted and the untrusted code. To protect against the Iago attacks, the arguments and the return values are checked for invariant violations. Replay attacks are prevented by tracking the freshness of requests using nonces.

The results show that Glamding places 22-40% of application source code into the enclave, with 40 to 150 communication endpoints between the trusted and untrusted world. The performance achieved after partitioning is 30-60% of the native application, which is stemming from frequent transitions between the application partitions. The resulting interface size is also bigger than the interfaces achievable with library OS frameworks, and on the same order of magnitude as with minimal TCB approaches.

2.4 Related Work

Memory safety. While Intel SGX provides hardware protection from direct memory reads by privileged and unprivileged software, the in-enclave applications must still be correctly implemented to remain secure: any software defect that causes the application to disclose its secrets renders the protection from Intel SGX useless.

The biggest category of software defects found in the common user software are spatial and temporal memory vulnerabilities, stemming mostly from the insecure use of C and C++ programming languages which lack memory safety [32, 99]. There still exists a large number of applications written in these programming languages, which could be run inside the enclave, and thus need security hardening to prevent the exploitation of these vulnerabilities. One of the promising approaches to these problems is based on the program instrumentation, where the program is enhanced at compile time with additional instructions and data structures necessary to ensure the instrumentation goal—memory safety of the code.

SGXBounds is an instrumentation engine that adds spatial memory safety to the existing C and C++ code inside Intel SGX enclaves [190]. SGXBounds implementation is SGX-aware: because significant utilization of physical memory usage inside SGX enclaves causes large performance overheads, SGXBounds limits the maximum size of enclave virtual memory to 4Gb, and uses the freed bits inside the virtual address to store the upper bound of the object. The lower bound of the object is stored in memory after the end of the object.

While this approach does not provide protection comparable to hardware approaches [231], its significant benefit is a much lower overhead compared to other solutions, including hardware (Intel MPX) and software (Address Sanitizer): only 17% performance overhead on average, compared with 51% for Address Sanitizer, and up to 600% for Intel MPX, which is explained by SGXBounds' high memory locality and low memory consumption.

Storage solutions. We do not consider the work of building specifically a scalable and reliable storage solution to fall within the scope of our project, however it is an important component of every cloud software stack. Thus, it is important to consider the solutions built for support of persistent storage inside Intel SGX enclaves.

The fundamental functionality exposed for achieving storage confidentiality and integrity in Intel SGX SDK is called sealing: in a nutshell, it employs encryption of data using a hardware-derived key tied to the enclave identity. Thus, it is not accessible to the software outside of the enclave or to other enclaves. Sealing as implemented in Intel SGX and SGX SDK allows upgrading both Intel SGX firmware and the software versions inside the SGX enclave.

Storage primitives exposed by Intel SGX SDK contribute only low-level cryptographic functionality, while cloud architecture usually relies on dedicated key-value storage solutions, for

example Memcached, Redis, RocksDB, and so on. An example of a system that adapts a modern key-value store for Intel SGX is Speicher [76]. Speicher is a high-performance networked LSM store that relies on kernel bypass for fast access to the NVMe devices using the SPDK (Storage Performance Development Kit) framework. It implements a novel on-disk LSM data structure that maintains confidentiality, integrity, and freshness of the stored data. While Speicher's direct I/O library can achieve near-native throughputs and latencies, its freshness protection mechanism causes a large throughput overhead, which is approximately 15–30× lower than the throughput without freshness protection.

Speicher's significant overhead is stemming from its use of platform monotonic counters as a primitive for freshness protection. An alternative approach to achieving freshness in storage systems is implemented in ROTE [212]. ROTE protects against a distributed adversary by relying on multiple nodes to hold the freshness information, which requires an attacker to subvert more than a third of all nodes to violate the system properties. By relying on network nodes instead of the platform counters, the latency of freshness state reads and writes can be reduced by a factor of 20, which allows a corresponding increase in the system throughput.

SGX sealing key is the fundamental primitive necessary for the protection of *local on-disk* data. However, at the cloud scale, distributed storage systems are prevalent and thus the SGX-protected solutions that use network storage can be also constructed. PESOS is a storage framework that provides a policy-enabled REST interface to network-enabled Seagate Kinetic disks [185]. By attaching policies to the key ranges, Pesos can flexibly implement several common use-cases, such as mandatory access logging, time-based storage access, and so on. A single Pesos node scales the amount of available storage by adding more Kinetic disks. However, it suffers from the overheads stemming from comparably slow Kinetic disks: the achieved throughput is approximately 1000 IOp/s when the request size is small.

An important factor for the security of applications that rely on the file systems of the untrusted OS for data persistence is its correct use and resilience to Iago attacks [100]. In this context, BesFS proposes a library for accessing the untrusted file system, which is formally specified and verified using Coq theorem prover. This allows BesFS to provide high-assurance interface that resists various attacks on the system call interface while allowing users to chain the provided functions in arbitrary ways.

Remote attestation. Remote attestation is a critical requirement for achieving trust in the in-enclave software. Without attestation, it is impossible to distinguish an execution that happens inside the Intel SGX enclave from the execution inside the emulator of Intel SGX. Attestation is performed during the application startup and setup phase; the verifier sends the secrets necessary for further operation only to the successfully verified applications.

Intel's remote attestation protocol provides a cryptographically signed, remotely-verifiable statement about the enclave software TCB: the enclave software identity, platform configuration, enclave author identity, and additional data. The enclave identity is represented in the attestation message as the enclave *measurement*: a SHA256 hash of all operations that were executed to construct an enclave, taking into account the memory contents of the enclave, locations of the measured memory, and the protection flags of in-enclave pages. The platform configuration includes information on the microcode version, hyperthreading configuration, and so on. This information is needed because each of these components may have vulnerabilities (bugs in the implementation or opportunities for side-channel attacks), thus, the attestation verifier may choose not to trust the platform with these hardware bugs. The enclave author's identity is conveyed as the hash of the RSA public key of the enclave

signer. Additionally, the enclave may choose to transmit arbitrary data along with the signature to facilitate the establishment of a secure channel to the enclave. If any of the system TCB components is discovered to be untrustworthy, there is a *TCB Upgrade* procedure that allows handover of service from the old instance of software to the new, patched one.

Attestation of enclaves is implemented in Intel SGX in two variants: local attestation and remote attestation. Local attestation is performed in a hardware-assisted way between two enclaves running on the same CPU. It allows two enclaves to mutually attest each other by exchanging attestation reports, which include the identity of the enclave communication partner, and additional information to establish a secure channel between the enclaves.

Remote attestation is more complicated and allows the remote party to verify the identity of the enclave and configuration of its platform. It is performed via a local attestation with a special, Intel-provided, architectural Quoting Enclave, which signs the report using the attestation key, following the Enhanced Privacy ID (EPID) cryptographic protocol. EPID is an extension of the Direct Anonymous Attestation used in TPM 1.2 specification. The EPID protocol has the important properties of anonymity (it should be impossible to attribute two signatures to the same or different entities in the group) while providing a possibility for signature-based revocation (it is possible to revoke the signature key based on the signature alone). EPID as implemented in Intel SGX divides the nodes into groups based on the CPU family (i3, i5, i7). To generate the attestation key and verify the signed attestation report, Intel provides an Intel Attestation Service (IAS) infrastructure.

In some cases, implementations of custom, third-party attestation protocols are required. For example, it may be impossible or undesirable to connect to IAS for the attestation quote verification, or if there is a necessity to implement a remote attestation protocol with different privacy requirements. To allow the implementation of these more flexible requirements to attestation, Intel has provided a set of libraries and tools called Data Center Attestation Primitives (DCAP). Combined with Flexible Launch Control (FLC) extension, an independent implementation of a Quoting Enclave can be built. As an example, Intel provides an ECDSA-based Quoting Enclave. It is expected that large cloud providers will use DCAP to implement custom attestation infrastructure.

2.5 SGX Challenges

The limitations of Intel SGX present challenges for a wide range of use-cases, which application developers must tackle to ensure that their software runs inside enclaves securely and efficiently. This is especially true for the some classes of applications developed for the existing operating systems, which use rich OS functionality and rely of low performance overheads of the used interfaces.

2.5.1 SGX Challenges for Network Middleboxes

After investigating the applicability of Intel SGX to common cloud software, we turn our attention to analyzing and solving challenges related to the use of Intel SGX for networking services. As there is a new trend for programmability in the network, exemplified by Software-Defined Networking (SDN), and Network Function Virtualization (NFV) appliances replacing fixed-function middlebox devices, we believe that this is a promising direction. In particular, we consider network middleboxes, which have gained traction in the Internet for a variety of use-cases, for example for caching, traffic optimization, deep packet inspection, and so on.

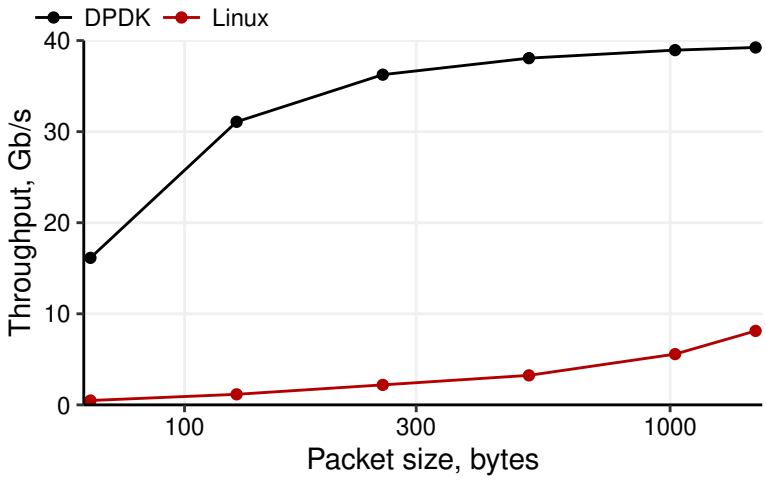

Figure 2.4: Throughput of noop middlebox application for native Linux and kernel bypass packet access methods.

The main challenge of using SGX enclaves for middleboxes stems from an inefficient communication interface between the enclave and the NIC. The fact that modern NICs achieve wire speeds of 40–100 Gb/s puts stringent lower bounds on the system performance. In many cases, even native software that communicates via system calls with the NIC cannot operate at such rates (Figure 2.4). Native applications rely on the combination of the following strategies to solve this problem:

- Implement part of the application in a SmartNIC, either via programmable packet processors (e.g. using P4 [91]), or on-NIC FPGAs.
- Move part of the application into the kernel, which recovers the performance by tightly integrating into the relevant kernel subsystems.
- Implement kernel bypass access to NIC, running the driver in userspace as a part of the application.

The drawback of the first approach in the context of trusted execution is that the on-NIC software in this case does not have the TEE (Intel SGX) protection. In-kernel implementation is challenging because of the high complexity of developing kernel code, missing isolation features, and particularly for Intel SGX because enclaves can run only in userspace (Ring 3). Therefore, only the last option remains available for use by Intel SGX enclaves.

When userspace access to the NIC is used, the partitioning trade-offs for Intel SGX enclaves gain a new dimension: putting the userspace device driver inside the enclave increases the system TCB. On the other hand, leaving the driver run as a part of untrusted runtime requires implementation of an additional memory queue for passing the received packets into the enclave.

One fundamental limitation of Intel SGX in the context of userspace in-enclave device drivers is that the communication channel between the enclave and the device is not protected, and the untrusted operating system may expose the emulated version of the NIC instead of the real one to the enclave. One solution to this problem requires establishing a secure DMA (Direct Memory Access) channel between the NIC and the enclave [131], but

this approach requires that a high-end FPGA is attached to the PCI Express bus on the NIC, which is not a standard design.

Another limitation that reduces the performance of middleboxes with Intel SGX is that middleboxes require a fast clock to periodically schedule the execution of some network functions, for example for a load generator function. However, Intel SGX does not provide a sufficiently low-latency clock, causing latency degradation. Time sources inside the Intel SGX enclave, however, warrant a more detailed discussion.

2.5.2 Time Sources for Intel SGX Enclaves

System time access is an important system service that current SGX frameworks provide via delegation to the OS, as SGX does not have a secure source of time. In this section, we provide a short overview of the timers provided on the 64-bit x86 Intel CPUs.

While most of the timers are implemented in hardware, there is a possibility to construct a software timer. In its simplest form, it is just a tight loop that increments a register and writes it to the in-enclave memory [263]. This timer can be quickly read in the enclave by reading the corresponding memory location. This method has multiple drawbacks: first, it requires a dedicated core for the timer thread that must be continuously running to update the timer. Second, if the timer thread gets preempted, the timer value becomes stale, which the clients of the timer must detect.

Hardware methods use primitives exposed by the CPU or by a peripheral device. Intel CPUs expose two hardware time sources: the Timestamp Counter (TSC) and High-Precision Event Timer (HPET). Timestamp Counter is the timer embedded into the CPU that is incremented at a fixed, power management-independent rate. The internal counter value is exposed to the software via `rdtsc` instruction, subject to the value adjustment: first, the internal value is adjusted using a fixed divider; second, an OS-configured offset is added to the value to produce the final value. Thus, the OS can adjust the timer value by writing the offset or by directly writing the value of the clock into one of the corresponding model-specific registers (MSRs). Additionally, the clock increment rate can be controlled inside virtualized environments using the *TSC multiplier* and *TSC offset* Virtual Machine Control Structure field. Reading the TSC is extremely fast — approximately 20 cycles.

High-Precision Event Timer (HPET) is a timer designed to synchronize multiple event streams with high accuracy. It can be read without the direct involvement of the operating system through MMIO memory. However, it has much higher latency than TSC: around 0.6 μs.

Operating systems typically use either TSC or HPET as a time source, preferring the use of TSC unless it is proven to be unstable. As reading time is a common operation in userspace applications, it is common to implement the time-related system calls in vDSO — a small shared object mapped into the application address space by the kernel. vDSO contains correction variables necessary to transform hardware-tracked time into its real-world value, and code to perform these calculations.

Some peripheral devices contain their own clocks and can expose them to the OS. One common example of such peripheral-provided clocks is the TPM clock, which maintains the uptime of the platform. While it does not track the real-world time, having access to a monotonic time source is sufficient to implement many time-based protocols. Also, devices like NICs often provide a hardware clock for packet timestamping or PTP synchronization. Enclaves could use these on-device timers as the time sources provided by the OS are unreliable or unavailable.

2.5.3 SGX Challenges for Distributed and Serverless Computing

While the mutual attestation is considered out-of-scope for this thesis, it is a required component in the production systems. Therefore, when implementing systems where multiple enclaves communicate together, the communication protocols should include provisions for mutual attestation. Mutual attestation is also of great importance during the distributed system bootstrapping, when the initial distribution of shared secrets and configuration happens.

Mutual attestation on a single computer can be trivially implemented using the SGX-provided local attestation. However, in the context of the distributed system, remote attestation has to be employed to establish trust between the system nodes. In this case, mutual attestation can be implemented in peer-to-peer or transitive variants. In the former case, two enclaves attest each other via direct communication. In the latter case, enclaves establish trust by performing mutual attestation with the special attestation and configuration service.

The benefit of the peer-to-peer approach is that there is no single point of failure in the system. This approach, however, has a chicken-and-egg problem: the enclaves have to know the expected measurements of their communication partners, which either has to be embedded into the binary, thus causing tight coupling between the enclaves, or distributed during startup, which requires a secure channel to a trusted node. This problem is avoided in the case where all enclaves perform attestation with a central attestation service, which transmits the configuration and the secrets to the successfully attested enclaves. To avoid a single point of failure of this design, standard replication techniques can be used.

In our work, we rely on a system called Palaemon [138], which transparently handles the attestation and secret distribution to the enclaves during the application startup. Palaemon represents the system configuration in a declarative policy-based language similar to Docker Compose, and transparently handles key generation and injection into the configuration files. To secure Palaemon itself, it implements board-based policy control (multiple stakeholders have to reach consensus to change the policy stored in a Palaemon instance), and runs in the SGX enclave itself to protect the secrets. Palaemon client is embedded into the SCONE framework and is configured with only a few environment variables.

Independent of the configuration and attestation system used, the enclaves have to establish a secure channel for communication after they perform the mutual attestation. The industry standard for secure communication is TLS, for which SGX-aware extensions exist [179], however in some cases more flexibility is required. As the communication security is out of scope for Intel SGX, in these cases custom communication and key management protocols have to be developed.

For trustworthy cloud computing, two additional challenges are stemming from the usage of Intel SGX and the threat model of an untrusted cloud provider. First, trustworthy accounting has to be provided: users have to be able to independently verify the resource consumption (CPU time, memory, network, and disk I/O) of their functions. One solution to this problem is provided in the S-FaaS serverless system [65]. It features an implementation of a custom timer that measures the execution time inside the enclave and computes the memory-time integral during the program execution. Network and Disk I/O amounts can be measured directly by the enclave runtime.

Second, in the context of serverless platforms, function startup and teardown time should to be minimized, as it has proven to be a bottleneck. There are multiple solutions to these problems, including reusing the enclave instances, relying on language-based isolation between functions, and directly minimizing the function startup time. While our research investigates minimizing the function startup time, TFaaS relies on isolate features in Ducktape

and V8 Javascript engines to minimize the function spawning overhead [90].

3 Efficient Support for POSIX Applications inside Intel SGX Enclaves

As explained in §1.4, POSIX applications are prevalent in cloud, as most common OS used inside VMs and containers are various Linux distributions [44]. In this chapter, we focus on designing and implementing an efficient runtime support framework for running Linux/-POSIX server applications inside Intel SGX enclaves. More specifically, these applications are typically long-running, multithreaded programs with very high I/O rate requirements: they receive a request from the client, and use it to compute the response, which may involve reading in-memory or on-disk data structures. Request handling typically happens in multiple threads to efficiently utilize modern multicore CPUs, especially when CPU heavy cryptographic operations for TLS termination are necessary. Unix-like operating systems provide a set of interfaces, such as standard library functions, system calls and signals, specified in the POSIX standard, which the applications use to implement the required functionality.

Intel SGX enclaves, on the other hand, provide and environment that is largely isolated from the operating system and system libraries by its security-oriented design, making it the responsibility of the developer to implement the functionality specified by POSIX. SGX support frameworks attempt to simplify the developer's work by implementing most common POSIX functionality shared by a large number of applications [285]. More specifically, the following functionality must be provided:

- thread management functions (pthreads interface);
- memory management functions (malloc, mmap, etc.);
- communication interface to the OS (delegation of calls to the libc, or system calls and signals).

At the same time, the enclave TCB should be kept at a minimum: first, to minimize the memory consumption of the enclave framework, as the amount of memory inside the enclave is extremely limited; second, to reduce the attack surface inside the enclave.

SGX frameworks implement these requirements with a different set of trade-offs, which fall into two major approaches of SGX runtime construction: minimal TCB-based, and library

OS-based. However, both approached make unsatisfactory trade-offs between the TCB size, system performance, and the developer effort necessary to use the runtime. In particular, library OS approaches provide comprehensive subset of POSIX functionality by running a large code base inside the enclave, thus increasing the system memory consumption, as well as effort necessary to verify the system. The emulation of POSIX functionality inside library OSes is also error-prone, because POSIX functions have a large number of corner cases that must be correctly taken care of. For example, correct setting of error codes is critical for correct operation of software. On the other hand, minimal TCB approaches can be hard to use due to the necessity to develop a custom interface for each application. Existing approaches rely on SGX-provided primitives for entering and exiting the enclave, which are a source of high performance overhead.

Thus, the questions that we want to address in this chapter, are:

- How can we achieve a better trade-off between the TCB size and the functionality available inside the enclave then both library OS and minimalistic approaches?
- How can we provide an efficient, comprehensive interface to the OS without complex emulation of the OS functionality?

To provide an answer to these questions, we introduce a system called SCONE (Secure Containers), which provides a lightweight, generic libc-based runtime with minimal TCB that can be applied to a wide range of POSIX applications. By building on libc instead of on an existing library OS or minimalistic SGX runtime, SCONE exhibits a minimal TCB without compromising on the functionality available to the in-enclave applications. This is achieved by a two-fold approach: the functionality that must be executed fully inside the enclave, like thread and memory management, is implemented as close to the libc interfaces as possible. On the other hand, the invocations of OS functionality like network of file I/O is fully delegated to the outside of the enclave. To reduce the probability of coding errors and simplify the support of system calls, we have automatically generated the code for forwarding system calls to the outside using an annotated C header and a custom generator. As SCONE's communication interface is fully generic, porting new applications into enclaves can be as easy as recompiling with a SCONE-provided cross-compiler.

To improve the system performance, we rely on *an asynchronous system call interface*, which allows SCONE to avoid SGX-induced performance degradation from frequent context switches, and switch the runtime to M:N threading model to fully reap the performance benefits. It provides two-fold benefits: first, it allows SCONE to support an arbitrary number of in-enclave threads without high performance overheads in spite of SGX limitation (fixed number of TCS pages), second, it allows application threads to submit their I/O requests without exiting and entering the enclave, and without blocking the in-enclave thread.

The implementation of asynchronous system call interface relies on a lock-free concurrent queue pair, which has better scalability than traditional lock-based queues. The threads inside and outside of the enclave are also pinned to sibling hyperthreads to facilitate the communication through the shared cache.

We measure the performance of SCONE using a set of micro- and macro-benchmarks, using common network servers as the focus of the study. Our evaluation shows that SCONE runs with 8-25% overhead in throughput for a wide variety of common network server applications, albeit at the cost of higher CPU utilization.

The structure of this chapter is as follows: in §3.1 we provide a motivation for our system, by using a popular web cache server as an example. In §3.2 we will discuss the design deci-

sions of SCONE. §3.3 will introduce the implementation of SCONE, and we will conclude with the discussion and the conclusions in §3.5 and §3.7 correspondingly.

3.1 Motivation

To understand, what kind of interfaces must be supported by SCONE, consider a Memcached server, commonly used to implement caching and in-memory storage system. Running Memcached inside Intel SGX enclave allows protecting the cached information from unauthorized accesses by the cloud operator insiders, and makes the exploitation harder for other kinds attackers, hence it is an important use-case to consider.

Memcached is a multithreaded application. It has a thread that listens and accepts new connections on the network, and a number of threads that process the requests received from the accepted connections. The threads synchronize using pthread-provided APIs.

The handling of the request involves TLS decryption, examining the request, which may involve reads and writes to memory and to the local file system, preparation of the reply which gets TLS encrypted, and sending it over network. As Memcached is designed to be a user-facing component that provides replies to the majority of user requests, its I/O performance must be as fast as possible: the requirement is to handle a large number of client sessions with maximum throughput and minimal latency.

POSIX-compatible operating systems provide two features to support these use-cases: non-blocking I/O over file descriptors, and a functionality to poll the readiness status of a set of file descriptors. Non-blocking I/O can be enabled by setting the `O_NONBLOCK` flag on the file descriptor at the file descriptor creation time, or by using `fcntl` system call. Polling is provided by system calls like `poll`, `select`, `epoll`, `kqueue`.

Memcached performs asynchronous I/O using the `libevent` wrapper library over the OS asynchronous I/O functionality to handle multiple connections in one thread, and uses a range of functions to invoke the operation system functionality.

Additionally, Memcached sets up a few signal handlers to exit cleanly when the user or the system stops the application. The low-level memory management functionality of the operating system like `brk` and `mmap` are not available as well, making it impossible for the libc to request memory from the operating system.

The Memcached instance would typically be deployed inside a virtual machine instance or a container for performance and security isolation from other peers. However, the fundamental deficiency of container technology is that it aims to protect only 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 [106, 107, 108], or they compromise the credentials of privileged system administrators [303]. Hence, hardware-assisted technologies like Intel SGX started to gain traction in the cloud.

To enhance the protection of Memcached instance using Intel SGX, we develop a runtime system called SCONE, which supports all of the aforementioned functionality required by the common cloud software. However, this task is complicated by restrictions of Intel SGX:

For thread management, the thread synchronization has to be implemented inside the enclave, as otherwise an untrusted operating system could subvert the synchronization, and cause data races to take over the control over the enclave. Additionally, Memcached allows

specifying the maximum number of threads at runtime, while Intel SGX requires to preallocate the Thread Control Structures at compile-time.

For the signal support and memory management, the corresponding functionality has to be implemented inside the enclave as well, as the OS can only deliver signals and manage memory in the untrusted part of the application.

The biggest runtime support challenge lies in the implementation of the communication interface between the enclave and its untrusted environment. First, it is necessary to decide, what the interface could look like: while a custom per-application interface violates the generality property, there is still a choice between the minimal interface of the library OS, and forwarding of system calls of libc function to the outside. Second, it is necessary to ensure that the communication interface is protected against Iago attacks. It should also be possible to *shield* the interface.

The most important requirement for the communication interface is achieving high performance. As Memcached can scale up to one million requests per second on COTS server, there is a necessity to minimize the performance loss due to additional communication level between the enclave and non-enclave parts of the application.

To conclude, cloud workloads exhibit significant challenges to serving them inside Intel SGX enclaves, all of which affect performance of the resulting enclaved software. Our runtime framework, SCONE, implements all these properties with minimal TCB, thus leaving more memory for user data and having less attack surface. We explain the design of SCONE in the next Section.

3.1.1 Threat model

When designing for the Intel SGX enclaves, 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. With Intel SGX, the host OS remains in control of physical resources, thus it can deny the enclave CPU time or delivery of packets, thus preventing the enclave from doing useful work. However, such attacks would lead to visible deviations from SLAs, which allows the cloud tenant to dispute the processing fees. We consider trustworthy resource accounting an orthogonal problem.

In this work, we also consider side-channel and microarchitectural attacks out-of-scope. While recent publications have shown the extreme vulnerability of SGX enclaves without mitigations to such attacks, we expect that most of the vulnerabilities will be fixed in the new revisions of Intel CPUs or with microcode updates, and that the enclave application developer will rely on available countermeasures in form of any of the available compiler-based techniques and constant-time constant-cache-footprint coding techniques for the cryptographic code [139, 233, 234].

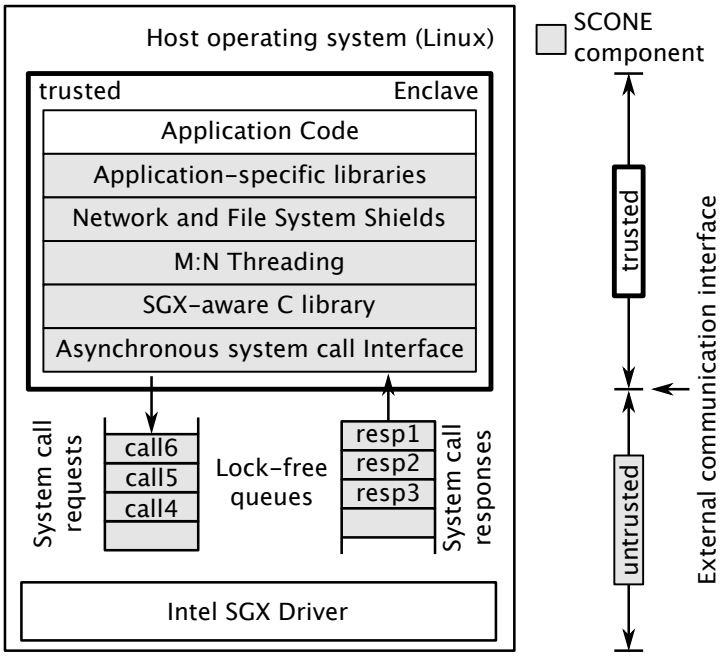

Figure 3.1: SCONE architecture. The highlighted components were implemented as part of SCONE.

3.2 Design

Our objective is to offer trustworthy cloud software on top of an untrusted OS: a secure container must protect containerized services from the threats defined in §3.1.1. We also want to achieve maximal performance with minimal TCB without compromising the system security.

3.2.1 Architecture

Figure 3.1 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 §3.2.5). To protect the integrity and confidentiality of data processed via file descriptors, SCONE allows transparent encryption and authentication of data through *shields*.

(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 §3.2.4).

(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 userspace. Hence, the threads inside the enclave do not have to exit when performing system calls (see §3.2.5).

3.2.2 Trusted runtime

Given the task of supporting POSIX applications, we need to choose and implement a runtime system. Most of applications running in the cloud are either implemented in C or C++, or in a language the runtime of which is implemented in C/C++, Therefore, C and C++ are the languages we aim to directly support with SCONE.

The foundation of the C runtime is the standard C library (libc), along with a few support libraries (libgcc in case GNU Compiler Collection is used, or compiler-rt in case of Clang/LLVM). Standard C library implements the interfaces to the OS kernel, and exposes a standard, POSIX-specified interface to the applications. We aim to provide a standard C library interface to the applications as well, by modifying and extending an the existing standard C library. The main reason why these modifications are necessary is that some functionality, like communication with the OS and, for example, thread management, must be implemented as part of the libc modifications, and cannot be implemented efficiently in the lower level of abstraction with Intel SGX. For example, to implement system call delegation, we cannot let an existing libc invoke `SYSCALL` instruction, which will require several transitions between the trusted and untrusted worlds for emulation. These changes must be made inside the libc.

While the functionality exposed by the libc to the user applications will stay unmodified, the interface between the libc and the OS kernel is more complicated, as some of the functionality provided by the kernel to the non-SGX applications has to be emulated inside the enclave due to SGX restrictions. These low-level system calls are related to the thread management, memory management, and the low-level primitives for communication with the OS. Other interfaces between the kernel and the libc are used for the proper communication with the OS, and are described in detail in the §3.2.3.

There are several approaches for implementing the aforementioned low-level runtime components. For example, library OS systems provide this functionality as a subset of other OS-related components that they comprise. On the other hand, minimal TCB systems like Intel SGX SDK ship with the modified and stripped-down version of the standard C library, extended with a custom memory and thread management code.

In SCONE, support of the libc inside enclaves conceptually follows the minimal TCB approach: we strive for *minimal amount of emulation* of the kernel functionality inside the enclave. We analyze the libc code to find the interfaces necessary for running without kernel support, and implement strictly those interfaces. Unlike minimal TCB approaches, we do not strip rest of the libc interfaces, and delegate them to the OS instead. Unlike library OS approaches, we do not implement any other OS-related functionality inside of the enclave.

3.2.3 External Interface

A choice of an external interface is extremely important from the point of view of achieving the generality goal: SCONE must support running a POSIX/Linux applications as a generic function of an interface, without implementation of any mechanism specifically for any application. This requirement excludes the design choice of *custom interface* for SCONE: by definition, it must be adapted to the functionality that the application needs from the OS and from its library dependencies. Furthermore, partitioning the application to account for this is also application specific.

To better see what design choices are valid for SCONE, we consider how a POSIX application can request service from the operating system. On the lowest level, the application requests service from the OS using so-called *system calls*. The semantics of system calls

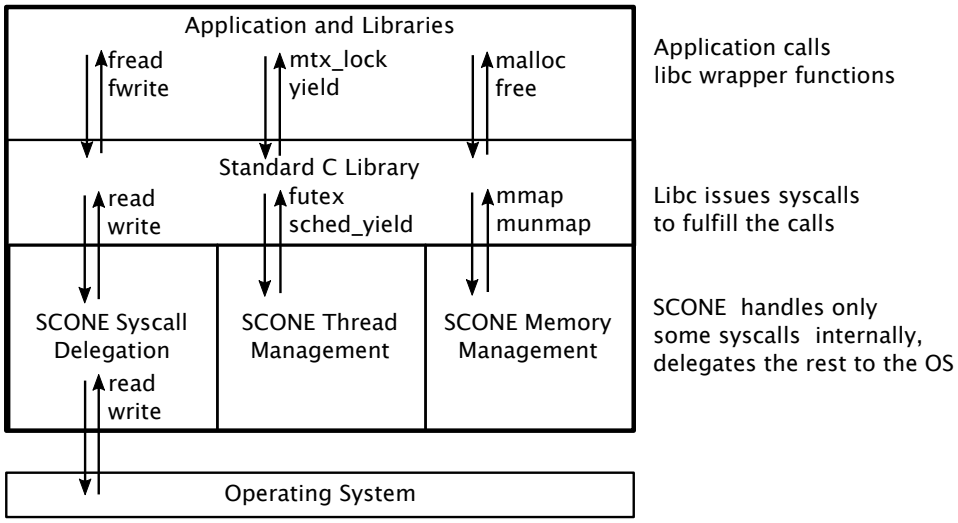

Figure 3.2: SCONE external interface and runtime components.

varies: management of threads and processes, memory management, communication over unidirectional or bidirectional streams, persistent storage of data, and so on. These operations require permission control and multiplexing by the operating system, which validates the correctness and the security permissions of the request. System calls on POSIX operating systems are typically implemented in a synchronous and trap-based way: the thread executing with the user privilege level invokes a trapping instruction which performs a controlled transfer to the interrupt or system call handler in the kernel. To this end, it saves the user processor state, switches the CPU privilege level to kernel, and restores the kernel context. Then, the kernel system call handler validates and executes the request, before reverting the control back to user mode. This state transition is costly even for normal user-level code, especially when the page table switch has to be performed as well (as required by the mitigations to the Meltdown vulnerability).

Applications can invoke system calls directly, but for convenience they are wrapped into functions inside the standard C library or as a part of a programming language runtime. The convenience stems from the fact that the system call calling convention is different from the function calling convention, thus wrappers inside the libc eliminate a significant amount of boilerplate code that would be otherwise required. Additional, arguably more important functionality stems from fact that on Linux, certain kernel versions may implement some of the functionality in non-standard-conforming way due to bugs, which may require additional code to handle such cases, or to make use of more efficient or extended but otherwise semantically compatible system calls that may become available in the future kernel versions.

Another design choice for the enclave external interface is whether to delegate system calls or function wrappers to the untrusted operating system. The alternative to delegation is emulation of the functionality inside, doing as much work of the operating system as possible and then forward only the primitive operation to the operating system, such as block-level disk I/O or L2 network I/O, however this design choice is error-prone and not robust: standard C library and applications relies on POSIX/Linux system call semantics which have a lot of corner cases, such as error codes and return values, behavior on some edge cases (e.g. passing `NULL` pointer arguments). Therefore, delegating this functionality is a more robust solution than emulation. For maximum flexibility, the decision about delegation or emula-

tion can be taken per each system call, as some system calls may not admit delegation, while others may be too complex or error-prone to emulate.

In the context of delegation, it is preferable to choose system calls over function wrappers for the external interface of the enclave. The reasons for this are: first, system calls are already an external interface between the application and the operating system; second, forwarding function wrapper calls to the outside requires application developer to partition the standard C library functionality into parts that must execute inside enclave and parts that can be safely delegated. This task is much easier to accomplish with the system calls then with the function wrappers, as the POSIX/Linux libc interface is larger than the system call interface.

Some of the system call arguments are pointers to the actual memory that kernel will read or write during handling of the system call. As enclave memory (EPC) cannot be read by the privileged software, that is by the operating system kernel, the enclave must copy these memory objects outside of the enclave and update the pointer arguments correspondingly, and copy them back into the enclave after the system call has been processed by the kernel. This copying should also be implemented in a secure way, to protect against Iago attacks, where some of the system call return values could be modified in a way that violates implicit invariants in the code (e.g., returned amount of data read from the network larger than the buffer passed to the system call).

An important system call performance optimization available on Linux is the virtual dynamic shared object (vDSO). It is a shared library mapped into the address space of the application by the kernel, containing implementations of several system calls which can be invoked without entering the kernel. On some architectures it may also contain code for performing system calls in the optimal way. SGX enclaves do not support vDSO as-is, because vDSO invocation would require calling into out-of-enclave code from the context of enclave.

System calls are a communication channel initiated by the application. However, the operating system can initiate the communication itself, via so-called *signals*. On POSIX platform, signals are asynchronous notifications sent by the kernel or a process to another process or thread. It is a limited form of IPC, as it allows to only invoke some functionality identified by a numerical code. Some signals sent to a process are handled by the kernel, stopping or killing the application, and optionally producing a core dump. Other signals can be handled by the application, or ignored altogether. System calls like `sigaction` or `signal` can be used to setup the in-process signal handler.

POSIX specifies *synchronous* and *asynchronous* signals. Synchronous signal is generated whenever an application performs specific action, and is delivered to the thread that executed the corresponding action. On the other hand, asynchronous signals are caused by external effects (for example, alarm expiration), and can be delivered to any thread that does not have this signal blocked.

When the operating system delivers a signal to the process, it interrupts its execution, saving its execution context on the stack, and sets up the stack frame for the signal handler according to the signature of the signal handler (for `sigaction` or `signal`), and passes control to the handler in the context of the signal stack frame. The return value in this synthetic stack frame leads to execution of the `sigreturn` system call, which destroys the signal stack frame and restores the previous execution context.

The main challenge for handling of signals inside Intel SGX is that the signal handling stack switching and destruction has to be reimplemented inside enclave, and integrated with the threading model of the trusted enclave runtime. For full support of the signal handling functionality, handling of nested signals has to be implemented too.

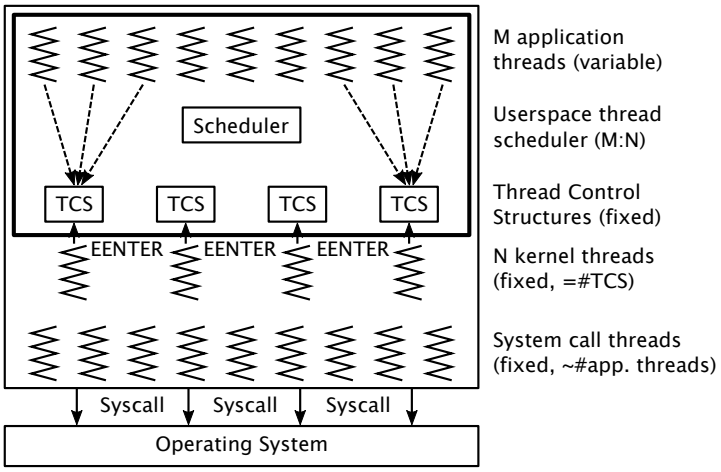

Figure 3.3: Implementation of M:N threading model inside SCONE for use with Intel SGX.

3.2.4 Threading model

Modern operating systems provide abstractions of threads and processes. From the point of view of the operation system, a thread is an atomically scheduled entity that executes as part of the process. The specifics of a threading system implementation classify the threading models into three categories:

- 1:1 threading — each user-level thread has a corresponding kernel-managed thread.
- N:1 threading — there are multiple user-level threads, and a single kernel-managed thread.
- M:N threading — M user-level threads are scheduled across N kernel-managed threads.

In case N:1 or M:N threading models are implemented, a user-level scheduler, which runs as a part of the application, is necessary to multiplex the execution of the user-level threads onto the available kernel threads.

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 [156], SCONE supports a variable number of application threads. Another reason for implementing M:N threading in SCONE is that it allows reaping the full benefits of asynchronous system calls, allowing a quick switch (without leaving the enclave) to a different userspace thread after a system call has been submitted.

As shown in Figure 3.3, multiple OS threads, called *enclave threads* in SCONE, can enter an enclave. Each thread executes a *scheduler*, which checks if: (i) an application thread needs to be woken up due to an expired timeout, signal delivery, 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 userspace threads implement cooperative multitasking: the preemption points for the userspace threads are thread creation and joining, synchronization primitives, and system calls: upon submission of the system calls, the thread blocks, giving opportunity for

a different thread to run on the same in-enclave kernel thread. Creation and destruction of userspace threads is oblivious to the OS kernel, and thus must be fully implemented by the SCONE runtime inside the enclave, as part of `pthread_create` and `pthread_join` implementations. Additionally, to avoid data race attacks by the untrusted operating system, SCONE should also implement the synchronization primitives (futexes) inside of the enclave instead of delegating them to the untrusted world (§3.3.3).

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 userlevel 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. We would like to note that there is no technical barrier to the implementation of preemption.

In addition to spawning N OS threads inside the enclave, SCONE also dedicates a number of OS threads to execution of system call requests passed from the outside. These threads are called *system call threads*. System call threads dequeue requests from the system call request queue, perform system calls, and enqueue results into the response queue (see Figure 3.1). The number of system call threads must be at least the number of application threads to avoid stalling when system call threads block. When there are no pending system calls, the threads back-off exponentially to reduce CPU load. The performance of this approach can be further improved by considering the memory locality: on the hyperthreading-enabled CPUs, the system call threads and the enclave thread can be located on sibling hyperthreads, enabling them efficient communication through L3 cache (§ 4.7).

3.2.5 Asynchronous system calls

Since SGX does not allow system calls to be issued from within an enclave, they must be implemented through calls to functions outside of the enclave: the executing thread must copy memory-based arguments to the non-enclave memory, exit the enclave and execute the function outside 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 §2.5.1, such *synchronous system calls* have acceptable performance only for applications with a low system call rate.

Instead, to achieve the performance goal, SCONE provides an *asynchronous system call interface* [277] (see Figure 3.4). Conceptually, this interface consists of a pair of 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. A thread inside the SCONE untrusted runtime receives and processes these requests. When the system call returns, the OS thread places the result into the response queue.

As shown in Figure 3.4, an application thread first copies memory-based arguments into the memory arena 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 memory arena are allocated statically for each userspace thread.

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,

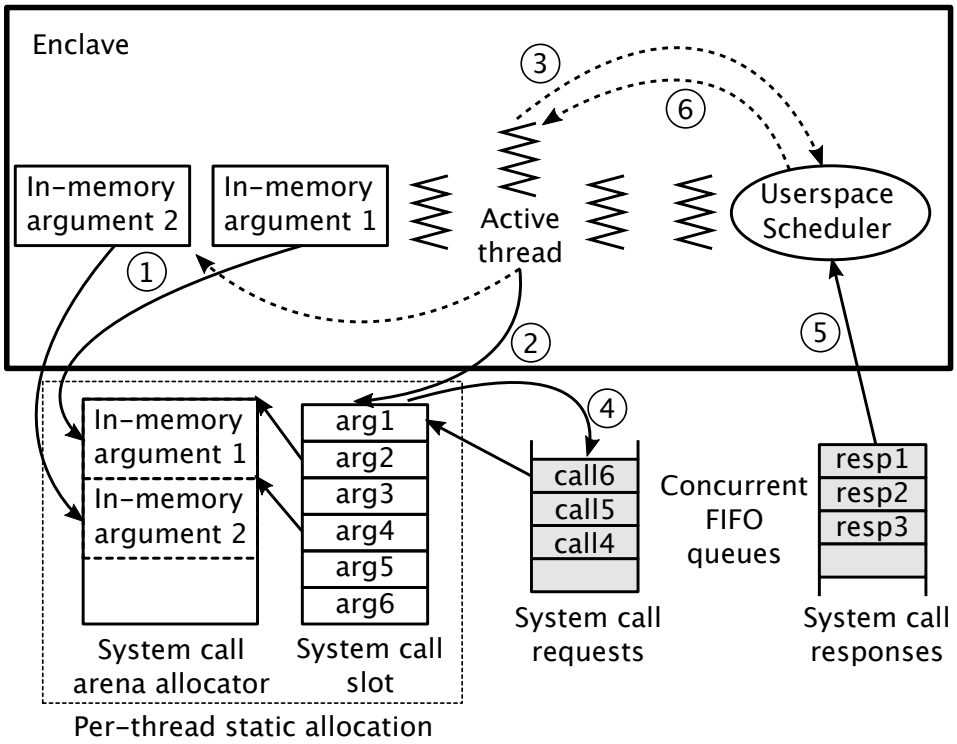

Figure 3.4: An example of execution of an asynchronous system call in SCONE.

there are checks of the buffer sizes, ensuring that no *malicious* pointers referring to the outside of an enclave can reach the application, and that excessive amount of data is not copied into the enclave. Finally, the associated application thread is scheduled again ⑥.

Due to the pointer destination check and the buffer size checks, SCONE is protected against memory-based Iago attacks [100]. These checks are performed in each code that directly consumes data outside of the enclave.

One or more *system call threads* running outside the enclave consume entries from the request queue. When a system call returns, a new entry is pushed into the shared *response queue*. *Enclave threads* consume entries from the response queue, to receive results for previously issued system calls.

The execution of the system call is performed fully in userspace. To achieve lower latency and higher throughput, the system call threads can be configured to run with realtime priority. Further performance improvements come from the core-local allocation of the quests and pinning of the enclave and system call threads to the sibling hyperthreads.

The request and response queues are lock-free and are implemented using atomic instructions, with a custom-tailored concurrent queue algorithm, called Fast FIFO Queue (FFQ). It exists in the single-producer multi-consumer (SPMC) and multi-producer multi-consumer (MPMC) variants used by a system call and return queues respectively. FFQ replaces the generic MPMC queue used in the original publication [291, 71]. More details about FFQ are available in the §4.

3.3 Implementation

3.3.1 Trusted runtime foundation

SCONE is built around a standard C library to provide a transparent shielding support to POSIX applications. Thus, we must choose a standard C library that will serve as a basis for our implementation.

There is a large number of different libc implementations (glibc, musl libc, bionic libc and others, [120, 45, 9, 11, 47, 30]), but a majority of them are targeted at embedded applications or specialized systems, supporting general purpose POSIX software only after patching, and often missing the required functionality. While the comprehensive overview of libc features is provided by Eta Labs [117], two implementations stand out as sufficiently complete to be used as the foundation of SCONE: glibc and musl libc.

Glibc. Glibc is the de facto standard C library on desktop and server Linux distributions. It extensively supports available POSIX and standard C functionality, is portable across various operating system kernels and architectures, and contains a high-performance implementation of common functionality. This large set of features also makes it large, complex, and hard to modify.

musl-libc. musl-libc is a C/POSIX library implementation focusing on providing a correct, reliable and simple implementation of a standard C library for Linux. It offers high-performance implementations of most basic functions, and supports a big subset of Glibc functionality. While it is portable across several hardware architectures, musl-libc is mostly implemented in C and is easily modifiable. It is well-tested as the default standard C library in several Linux distributions and is conformant with the most popular server applications.

Due to cleaner and smaller amount of code, and the focus on correctness, we have chosen to use musl-libc as a foundation of SCONE. However, our fundamental design decisions are generic and could be applied to glibc as well. The core of the SCONE runtime is modular and allows easy updates of the used musl-libc version.

To produce the ready-to-use enclave, we need to build the application with our modified musl-libc. While C-based application can be built with a system C compiler, as long as it is configured to use musl-libc headers and to link with the provided `libc.a`, to support C++, we need to rebuild parts of the C++ runtime with our modified libc. To this end, we produce a cross-compiler by building on the musl-cross-make project [33], which automates the GCC compilation process for musl-libc. The resulting cross-compiler can be used for building C and C++ applications and their library dependencies into in-enclave application image — ELF shared library.

However, an untrusted runtime is required to actually instantiate these images inside an enclave. We build such untrusted runtime as a shared library, and provide a compiler wrapper that automates the linking between the untrusted runtime and the in-enclave application image. To automate the linking, we have developed a wrapper around the cross-compiler gcc binary, which detects the cases when the final linking is performed, and embeds the in-enclave application image into the starter application as another section. It also automates the creation of SGX-mandated launch data structures.

3.3.2 System calls

To generate the code for serializing system calls, we create a C header file with declarations of all system calls supported for forwarding (229 out of 332 in total that SCONE recognizes). We annotate the system calls declarations and their arguments with additional information. System calls definitions can be annotated to be ignored, if in-enclave emulation is required, or can be turned into a stub definition, that exits setting an errno. For the arguments, there are more annotations:

- Size: provides an argument name that holds the size of the pointed-to object argument in bytes.
- Array: modifies the semantics of Size annotation to count the object size in the number of elements.
- Read/Write: if the pointer is used for reading and/or writing of data to the kernel.
- Deep copy: the annotated pointer argument contains pointers and must use a custom function for copying.
- String: the size of the object must be obtained using `strlen` function.
- Check: the size of the object is provided through a pointer, which may be `NULL`. Its validity must be checked before dereferencing the size.

This annotated header file is passed to the generator script, that parses the declarations and generates a C source file with the necessary serialization and deserialization code. The generator emits code that is secure against memory-based Iago attacks.

The pointer arguments of a system call must be copied to untrusted memory from the enclave. To allocate the untrusted memory, we create a memory arena allocator per-thread using a `mmap` call outside of the enclave. This allocator has the benefits of simple allocation routine (pointer bump), high data locality, and very sparing memory usage outside of the enclave.

Some system calls have different semantics depending on the first argument (command), for example `fcntl`, `ioctl`, `semctl`. For these system calls, we write the system call serialization wrapper manually. `ioctl` system call is particularly challenging as there are no tables describing all possible commands and the corresponding argument structures.

The system calls arguments also need to be provided to the kernel. Given that our implementation of system calls is asynchronous, we pass them to the kernel without leaving the enclave, via shared memory. System call number and arguments are written in a structure with the corresponding fields, which is statically allocated per userspace thread out of a global array. The index into this array is inserted into the concurrent queue, as explained in §3.2.5.

The implementation of concurrent queues has originally used a single MPMC queue [291] shared between all enclave and system call threads. However, to improve the performance and scalability, we have since replaced it with our own optimized queue, FFQ, which works in SPMC mode for the submit queue, and MPSC mode for the return queue. Due to this, SCONE requires allocation of a concurrent queue pair per enclave thread.

To achieve communication between enclave and system call threads through L3 caches on CPUs with hyperthreading, we pin threads that share a single queue pair to sibling hyperthreads. This behavior needs to be configured, along with other SCONE runtime parameters, through a configuration file (`/etc/sgx-musl.conf`).

To reduce the CPU utilization in case the enclave application is idle, we have implemented a backoff procedure in both system call threads and enclave threads. If there are no new system call submission for a configurable number for dequeuing attempts, the system call thread will start sleeping with the exponential backoff. The same is true for the enclave threads in case all userspace threads are blocked on system calls or locks.

3.3.3 Thread management

To reap the benefits of asynchronous system calls, SCONE relies on userspace M:N threading. In contrast to the traditional kernel threading, userspace threading multiplexes multiple in-enclave threads onto a fixed number of kernel-level threads that are spawned by the untrusted runtime during the application startup. To implement userspace threading, we follow the minimal emulation approach inside the enclave and implement only the functionality that Linux kernel cannot otherwise provide to the libc and the enclave.

We implement a simple userspace thread scheduler, that is executing in the context of an *enclave thread*. This scheduler checks the scheduler data structures in loop to see if there are any runnable userspace threads, and runs these threads. There are several data structures that are relevant for the scheduler:

- System call return queue is used to wake up a thread which has an outstanding system call completed.
- Runnable queue is used to schedule a thread that yielded its execution or has been just created. There is a single runnable queue for all schedulers.
- Futex hash table is periodically scanned to wake up threads which have expired timeouts.

Additionally, userspace thread scheduler exits the enclave in case of prolonged idleness to reduce the CPU utilization. The implementation is based on the `lthreads` library [64], but over time was extensively modified in all aspects.

Implementations of POSIX threads have to follow the System V ABI specification for its thread data structures, called *Thread Control Block* (TCB). For example, on Linux, TCB contains assorted information, such as

- self-pointer for supporting thread-local storage;
- locks that must be released by the kernel after the threads finishes to execute;
- signal masks to allow delivering signal only to specific threads;
- current locale;
- pointer to the thread stack and its size;
- `errno` value.

In our implementation, this information is located in the userspace thread control block, which is managed by our M:N threading library. In addition this control block contains:

- a register state storage for performing thread switch in userspace;
- userspace thread system call slot index and the pointer system call arena;

- additional pointers necessary for correct communication between the scheduler and the userspace thread.

On the other hand, the scheduler TCB contains only register storage for switching to userspace threads, stack base pointer, scheduler system call slot index, the pointer to system call memory arena, and the pointer to the system call queue pair. It also contains the pointers to the currently executing userspace thread and its system call slot and memory arena pointer.

SCONE threading library does not perform any special actions during the execution of user code. The yield points for each userspace thread are at system calls, and thread creation and synchronization functions. It is also possible to switch threads on timer interrupts (which can be emulated using timers), or using compiler instrumentation, however this functionality is currently not implemented. Thus our implementation of threading is non-preemptive, but we did not encounter any application for which this implementation detail would cause incorrect execution or starvation.

To reduce the latency and improve the throughput of concurrent queues, it is possible to pin enclave threads and the related userspace threads to sibling CPU hypercores. However, this functionality must be configured by the user through a simple configuration file. In this file, the user can specify the number of concurrent queues to allocate, and mapping of enclave threads and system call threads to the logical CPU cores.

A crucial component of thread management is synchronization, which on Linux is provided by the futex primitive. SCONE provides an implementation of the most commonly used futex operations inside enclave:

- `FUTEX_WAIT` and `FUTEX_WAIT_BITSET`;
- `FUTEX_WAKE` and `FUTEX_WAKE_BITSET`;
- `FUTEX_REQUEUE` and `FUTEX_CMP_REQUEUE`.

The implementation of futexes relies on spinlocks to synchronize the access to the in-enclave futex hash table. Note that the spinlock to access the futex hash table needs to be taken only in the case of contention, while in the uncontended case the standard lock-free futex operation sequence is followed.

Our implementation of threading in SCONE supports Thread-Local Storage (TLS). This is possible by always compiling the application and library source code in General Dynamic TLS Model (in contrast to Initial Exec model, which can be used by the non-library programs), where accesses to TLS are mediated through the runtime-provided function `__tls_get_addr` [205]. This leaves us the flexibility to modify it to access the Thread Control Block of the userspace thread, not of the enclave thread. With this change in place, TLS implementation requires only management of thread local storage memory on thread creation and destruction.

3.3.4 Memory management

POSIX specifies different ways for application to allocate memory from the operating system. The highest-level interfaces are functions like `malloc`, `free`, `posix_memalign`, and their wrappers, implement a significant part of their functionality in userspace, with thread-level caching, caches per size class, and other performance optimizations. These functions are typically either provided by libc, or superposed by linking the application with an external library that implements the allocator code.

These high-level function, in turn, rely on low-level memory allocation functions, that allocate memory in *pages* from the operating system. The system calls that allow page-granularity memory allocation are *mmap*, *brk*, and *sbrk*. In addition, POSIX provides a family of functions for reallocating virtual memory (`mremap`), changing its MMU access permissions (`mprotect`), or returning it to the operating system (`munmap`).

Following the minimal emulation approach, SCONE implements the lowest-level system interface for allocating memory on POSIX systems: the `mmap` system call. We do not support functions that manipulate the program breakpoint (`brk` and `sbrk`), because these system calls are rarely used in practice, complicate the design of allocator data structures, and are optional in musl-libc.

One of the restrictions of Intel SGX is that virtual memory mappings of an enclave must be preallocated during the enclave creation, preventing SCONE from following the normal POSIX workflow of managing them at allocation and deallocation time. To overcome this issue, our allocator uses a single preallocated virtual memory range, managed by a first-fit bitmap allocator. It has a low memory overhead, as it does not maintain information on virtual memory ranges or page permissions. We have also implemented the functions of the `mremap`, which reallocates the storage inside of the enclave, if possible. SCONE creates enclaves of maximum possible virtual memory size (64Gb).

The protection flags requested through `mmap` flags or `mprotect` are honored as far as the page tables are concerned, but are not reflected in the EPC permissions, as SGXv1 lacks the primitives for modifying the EPMD. This restriction is lifted only in §6, where we implement support for EDMM features of SGXv2. As our memory allocator does not track page permissions, we conservatively reset page permissions to read-write before zeroing the pages.

SCONE memory management subsystem also allows forwarding `mmap` calls to the untrusted runtime. In this case, the allocated memory is not SGX-protected. We use this functionality to, for example, create memory arenas for system call buffers. As this functionality is security-critical, we expose it only to the SCONE runtime.

POSIX standard specifies that `mmap` system call can be used not only to allocate anonymous memory from the OS kernel, but also to map file contents into the address space of the application. SCONE supports this functionality by forwarding the corresponding call to the operating system, mapping the file outside of the enclave. We argue that this is the right choice, as `mmap` is typically used with large files that won't fit into the EPC, causing paging, and also these files must be cryptographically processed inside of the enclave, so allocation of file contents outside of the enclave does not eliminate memory copies.

3.3.5 Signal handling

To allow SCONE applications communicate with the operating system using signals, we have implemented support for signal-related system calls inside of the enclave (e.g. `signal` and `sigaction`), and the mechanism to forward signals from the untrusted world to the enclave.

During the application startup, SCONE untrusted runtime installs a generic runtime for all signals that are not internally used by a standard C library implementation. When a signal handler is installed inside the enclave, it is associated with the signal in the global signal table.

When a signal is delivered to the enclave, an AEX procedure is triggered, which ultimately passes control to the generic signal handler in the untrusted runtime. Signal handling is different for synchronous and asynchronous signals.

For synchronous signals, the runtime enters the enclave and schedules the in-enclave

signal handler: it copies the register state of the currently interrupted userspace thread from the SSA into a dedicated storage area, and the register state necessary to enter the in-enclave signal handler is set up. After that, the thread yields the execution. On the other hand, the SSA register state is modified to enter the userspace thread scheduler. In case the current thread cannot handle the signal due to, for example, `sigprocmask` system call (which is also handled inside of the enclave), or missing signal handler, then the trusted runtime will terminate the application.

SCONE by default handles the Illegal Instruction signal (`SIGILL`) for several instructions that are common in the user software, such as `CPUID`, `RDTSC`, and `RDTSCP`. These instructions are executed outside of the enclave without SGX protection.

For asynchronous signals, SCONE implements a concurrent MPMC queue, which operates similarly to the system call queue pair. The untrusted signal handler enqueues the signal information into the queue, which is polled by the scheduler. Upon receiving a signal notification, it schedules the execution of the signal on the passing userspace thread, or adds the signal back to the pending list if no passing userspace thread is found.

To return from the signal handler, the enclave must enter the code running in the context of the scheduler (in contrast to the userspace thread context), as modifying a runtime state of the current thread is not safe. To this aim the enclave sets up the initial stack frame of the signal handler to return to the function that contains a special predefined invalid instruction sequence. When the exception is raised, the AEX transfers control to the OS and to the untrusted runtime, which enters the enclave to determine the exit reason. Upon detecting that the aforementioned instruction sequence was executed, enclave restores the original thread stack frame that was stored during the signal delivery, and returns control to the untrusted runtime, which in turn reenters the enclave, continuing the execution.

3.3.6 Limitations and Future Work

In this section, we will outline a few of the limitations of SCONE.

System call support. While current SCONE version implements both `fork` and `exec`, their correct implementation puts burden on the enclave runtime, as the state of the enclave memory and its resources, including shielded ones, has to be transferred to the child process. Thus, the author of this thesis has chosen to not implement these system calls, and they were subsequently implemented by other team members. Ongoing discussion about usefulness of these system calls is provided by Baumann et al. [82]. Furthermore, a generalized form of `clone` is not supported either and is challenging to implement inside of enclave.

Memory safety. It should be noted that while SCONE is a security-critical component of an enclave application, it may contain bugs that can lead to information disclosures [95]. As SCONE runtime is written in C, it may contain integer and buffer overflows, and thus would benefit from a memory-safety hardening, for example SGXBOUNDS [234]. On the other hand, formal verification could be applied to a part of SCONE runtime or even user enclave code to find and eliminate latent bugs and increase the resilience to Iago attacks [275, 273].

Improvements to the scheduler. SCONE userspace thread scheduler is extremely simplistic and could be improved for achieving higher performance. Currently, userspace threads that do mostly I/O are bound to a single enclave thread, and cannot be rescheduling to an idling enclave thread. As newly created userspace thread are added to global queue, there is no guarantee of fairness in work distribution, and thus our scheduler is not *work conserving*: some enclave threads could be idling while others have userspace threads ready for

scheduling. Scheduler modifications that improve fairness without excessive reduction of latency could further improve SCONE's performance in applications that spawn a large number of threads, for example Memcached.

Preemptive scheduling. While we have discovered no application that would not run correctly on the non-preemptive scheduler, there is no technical impediment to implementing it. The possible implementation strategies include inserting Varys-like compiler instrumentation [233], modifying untrusted AEX handler to periodically reschedule userspace threads, as it is implemented in SGXKernel [281], or by relying on POSIX signals to interrupt the execution of enclave threads.

Varys relies on compiler instrumentation to detect AEX event, giving the application an opportunity to execute a *hook function* after a specified number of exits. To implement preemptive scheduling, the hook function could execute `sched_yield` function, returning control to the scheduler. In contrast, SGXKernel modifies the userspace handler for AEX, provided to the enclave during the entry into the enclave: this handler periodically checks the running time of the current thread, and in case the thread has exhausted its budget, reenters the enclave to schedule a different userspace thread to run. These two approaches make different performance-compatibility trade-offs: Varys makes the preemption implementation protected, but distributes the cost of checking for AEX over all application code, slowing down its execution, which is not the case with SGXKernel. However, our experience has shown that modifying AEX handler has adverse effects on the debug tooling for SGX enclaves, requiring intrusive changes into the `gdb` plugin that must be tightly synchronized with the changes to the AEX handler. Finally, sending periodic signals to the enclave from the kernel is conceptually similar to the solution of SGXKernel, but much simpler to implement. However, it suffers from compatibility issues, as it requires reserving one of the signals typically available to userspace.

Additional experiments are necessary to ensure that the preemptive scheduling does not cause significant performance overheads.

Instruction emulation. While SCONE simulates several instructions by forwarding them to the untrusted world already (`CPUID`, `RDTSC`, `RDTSCP`), it could also be extended to simulate `SYSCALL` instruction, by invoking the system call wrapper function inside the enclave instead. This change would simplify porting applications that invoke system calls directly via inline assembly, instead of using system call wrappers.

Shared EPC memory. One of the SGX restrictions is that multiple enclaves cannot share EPC memory. Thus, if two enclaves want to setup a confidential, integrity-protected shared memory region for communication, software solutions must be used. Currently, SCONE does not provide any mechanisms for achieving this.

CPU utilization. The untrusted runtime spawns a number of system call threads to serve the requests from enclave threads, which back off only when no system calls are submitted for a large amount of time. In practice, system call threads could cause 100% CPU utilization on the cores they were pinned to.

vDSO support. While code inside Intel SGX enclaves cannot jump to the vDSO code directly, it is still possible to use vDSO code by relocating it into the enclave. It is only necessary to update the offset to the vDSO data page, and verify that vDSO code page does not contain malicious code. This change would improve performance of programs that invoke `clock_gettime` system call at a very high rate.

3.4 Evaluation

We evaluate SCONE on Intel SGX platforms split in two parts: first, we discuss the evaluation methodology and setup. Then, we evaluate performance of Nginx, Redis, and Memcached, comparing them to native versions. Finally, we discuss results of a system call overhead microbenchmarks.

3.4.1 Methodology

We perform all experiments on an Intel SGX-enabled machine with Intel Xeon E-2186G CPU with 6 cores at 3.80 GHz and 12 hyperthreads (2 per core), and 12 MB L3 cache. This machine has 32 Gb RAM and runs Ubuntu 18.04.5 LTS with Linux kernel version 4.15. 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 18.04.5 LTS with Linux kernel version 4.15.

We disable dynamic frequency scaling to reduce the interference with the measurements. The reported data points are based on ten runs, averaged using the arithmetic mean for throughput and latency and with geometric mean for CPU utilization.

3.4.2 Application Benchmarks

To showcase the performance of typical cloud software with Intel SGX,We evaluate the Nginx web server [253], Memcached [125], and Redis [252]. This are popular I/O-intensive servers, which are commonly used to store and serve data, and even to build application servers (Nginx with OpenResty). The goal of these measurements is to establish whether the design choices of SCONE, especially its system call interface, which is a major departure from the classical 1:1 threading model, are adequate for real-world software. An additional goal of the measurement is to establish the impact of the additional overheads that Intel SGX introduces into the real-world applications.

We compare the performance of two variants of each application: one build with into the native code with the system compiler and the GNU C library (`glibc`), and one built with SCONE cross-compiler. We compare to Glibc-build native software because it is a standard C library in most Linux distributions, and constitutes a more conservative baseline then musl-libc. In our experiments, applications compiled with Glibc perform with same or better performance than the musl-based variants.

To establish the impact of the system call interface, we modify SCONE to include a *synchronous system call execution mode*: after submitting a system call, the enclave thread leaves the enclave, executes the system call, and reenters the enclave. Importantly, after entering the enclave, the thread *enters the userspace scheduler*, thus retaining the M:N threading model. Thus, to reduce the performance effects of M:N threading, we always configure the number of enclave threads to be equal to the number of application threads. As this operation mode adds userspace scheduler overhead to each system call invocation, it exhibits worse performace than the 1:1 thread model.

In each application experiment, we configure the number of threads to give the best performance for each application and variant. We determine the best configuration empirically. We summarize the thread configuration in Table 3.1. Because experiments done with native Glibc, Graphene-SGX and synchronous SCONE runtimes either use 1:1 threading or do not

Application	Enclave Threads	Sthread Cores	Sthreads per Core	Application Threads
NGINX	-	-	-	1
NGINX (SCONE)	1	1	5	1
Redis	-	-	-	1
Redis (SCONE)	1	3	5	1
Memcached	-	-	-	12
Memcached (SCONE)	4	8	4	24

Table 3.1: Thread configuration used for the SCONE application benchmarks.

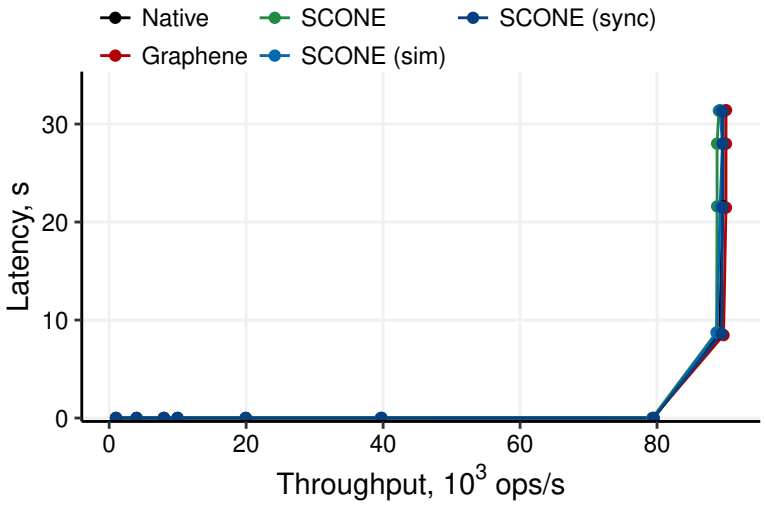

Figure 3.5: Throughput-latency plot of Nginx using multiple processes and cores to serve a 1Kb-sized file over HTTPS.

oversubscribe cores, Application Threads metric is the only meaningful one. In each experiment, we allow the benchmark to use all 12 cores of the machine, subject to explicit thread pinning in SCONE asynchronous system call interface.

Nginx. Nginx is a popular HTTP web server. We benchmark it using the `wrk2` benchmarking tool for measuring throughput and latency of the server while changing the rate of requests and the number of concurrently running client sessions, where each session consists of downloading a single 1Kb-sized file. We configure the benchmarked Nginx to serve the files via HTTPS, using self-signed certificates (certificate checking is ignored in the client).

Nginx uses a multi-process model with a single main process and a set of worker processes. As SCONE supports both `fork` and `exec` system call, we have initially ran the experiment in multi-process configuration. However, we have discovered that in this mode, Nginx easily saturates the 1G network link (Figure 3.5) in all configurations.

Thus, to present a more meaningful evaluation of SCONE's performance, we have ran the same experiment with a single worker thread, disabling the master process. The measurement results are available on Figure 3.6 and 3.7.

We can see that one core of the native Nginx reaches the line rate with 80k requests per second, and 76k req./sec before a significant increase in latency. SCONE in simulation mode exhibits slight overhead, reaching the maximum throughput of 68k req./sec (40k before latency increase). This slowdown is attributed to additional memory copies, which the CPU has to perform when serving the file. With SGX protection, these numbers futher fall to 60k

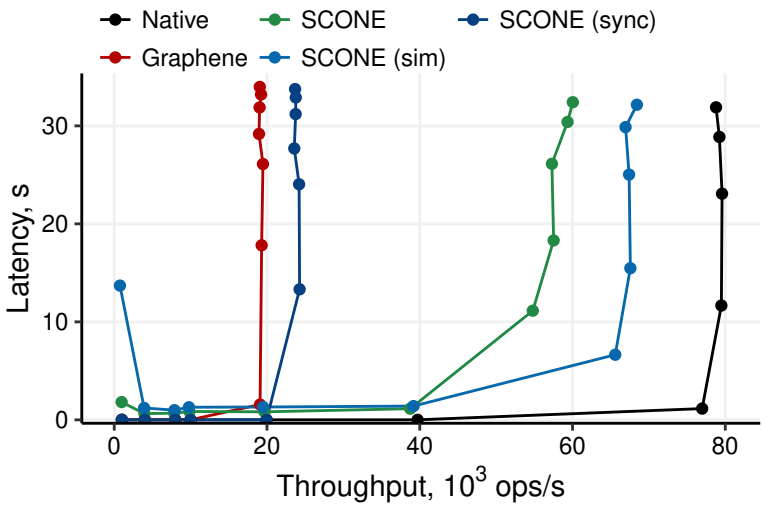

Figure 3.6: Throughput-latency plot of Nginx using a single worker thread to serve a 1Kb-sized file over HTTPS.

and 38k req./sec correspondingly. In this case, performance drop is likely caused by the increased cost of enclave entry and exit, which are triggered, for example, on timer interrupts, both on the enclave thread core and the system call thread cores.

Comparing these SCONE results, obtained with asynchronous system call interface, with the synchronous SCONE results, we can see that the performance results drop by 50-60% to 23k and 19k req./sec correspondingly. These results show that asynchronous system call interface is beneficial even to single-threaded applications. Please note that the same order of magnitude of performance increase could be achieved by using the core dedicated to the system call threads to another Nginx process instead.

Graphene-SGX exhibits performance similar to that of synchronous SCONE, but slightly lower. However, this is likely caused by the extra overheads of Graphene's PAL, which are mostly absent in SCONE. In the Nginx benchmark, this overhead amounts to 15% throughput drop, but only in overloaded mode; otherwise, the performance is mostly the same.

On the other hand, we can see that higher performance of SCONE is achieved through its increased CPU utilization, which is doubled compared to the native version. System developers must be aware of this trade-off when deploying SCONE services to the cloud.

Redis. Redis is an in-memory data structure store, used as a database, cache, and message broker [252]. It is a single-threaded application[1], that relies on event notification interfaces like epoll in its operation.

We benchmark Redis using the `memtier_benchmark` tool developed together with the Redis server. We run measurement in two steps: first, we run the benchmarking tool executing only `SET` operations, which initializes the database key-value pairs. This step is not included in the measurement. After that, we execute the second step, which executes `GET` operations on the same keys. The value size is set to 1Kb.

The results of measurement are presented on Figures 3.8 and 3.9. The results are similar to that of Nginx. The native version achieves 120k req./sec before saturating the CPU. On the other hand, SCONE both with and without SGX protection show 8% less performance, with

[1] Redis spawns a service thread to support its persistence features. This functionality was disabled for the experiments.

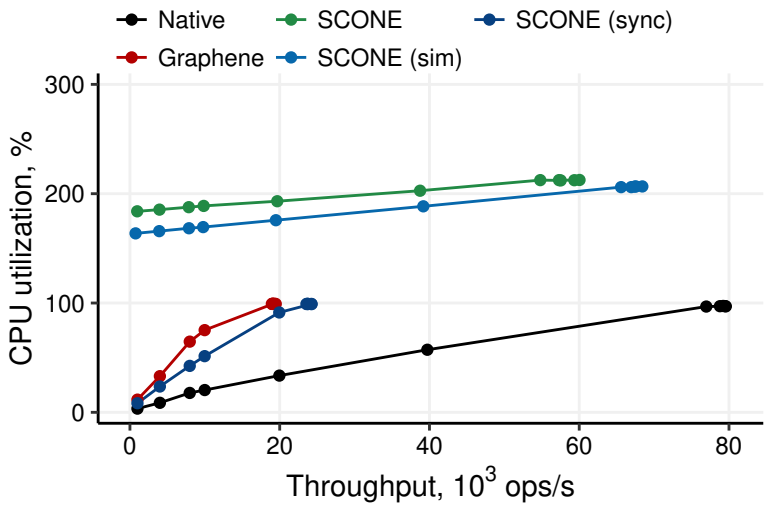

Figure 3.7: CPU utilization of Nginx using a single worker thread to serve a 1Kb-sized file over HTTPS.

a very minor reduction in performance for the protected variant. We expect the difference between these two variants to increase with further increase in the input request rate.

Also similarly to Nginx, Redis running with the synchronous SCONE runtime has nearly 50% less throughput then the asynchronous SCONE variant. This again shows that even single-threaded applications can benefit from SCONE: unlike Nginx, there is no easy way to run multiple Redis processes on the same machine, so spending more cores for system call threads on the machine is more sensible than in case of Nginx. Graphene shows even lower performance, reaching 30-40k req./sec without saturation, and 54k req./sec with saturation. We expect that this slowdown happens because Redis often invokes some costlier PAL functionality than in the case of Nginx.

Memcached. Memcached is an in-memory key-value commonly used as a caching system [125, 229]. Unlike previous benchmarks, it is a multithreaded application, that spawns a number of worker threads, along with some service threads, to process the user requests.

We evaluate Memcached using `memaslap` benchmark, with 128 byte key, 1024 byte values, 1:9 ratio of `SET`:`GET` operations, and 70% rate of overwrites. We have carefully tuned the benchmark parameters to avoid the EPC paging. The results of the experiments are presented on Figures 3.10 and 3.11.

As in the case of Nginx, native version reaches the line rate (120k req./sec). The same throughput is achieved by the SCONE version running without the SGX protection. On the other hand, SCONE version running inside an enclave runs with 8% performance drop comparing to the native and simulation versions, showing that SCONE's runtime is highly efficient for workloads like that of Memcached. On the other hand, the synchronous runtimes, like Graphene-SGX and SCONE in synchronous system call mode, run with approximately 20% overhead, reaching their maximum throughput of 93k and 98k req./sec. On the other hand, it is necessary to take into account the CPU utilization of SCONE, which allows it to reach near-native performance levels: while native version has low CPU utilization of CPU cores, reaching a CPU utilization of 1.02 when running on 12 cores, SCONE version running with Intel SGX has much higher utilization of 9.12, which may be a limiting factor when attempting to colocate the enclaved memcached with other microservices on the same machine. It is

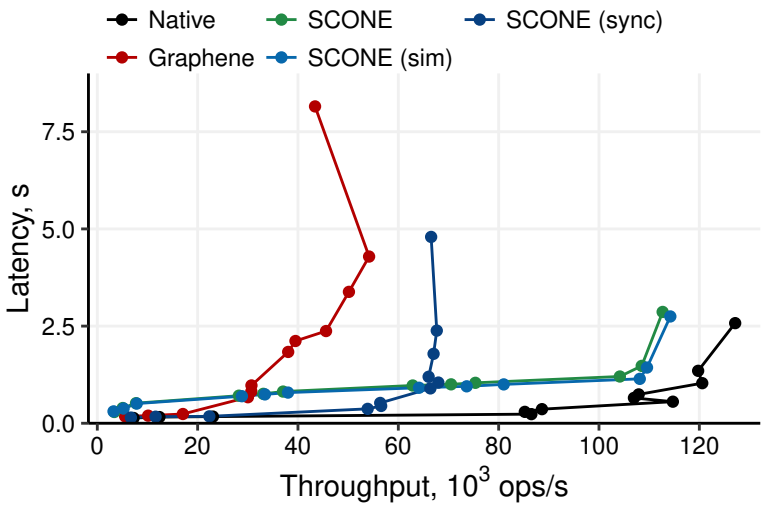

Figure 3.8: Throughput-latency plot of Redis running as native applications and with different SGX runtimes.

interesting to observe that the synchronous runtimes also have high CPU utilization in spite of not requiring system call threads to run, and thus these variants are likely to have the same issues with co-location.

3.4.3 Asynchronous System Calls

We separately measure the performance improvement from using asynchronous system calls instead of synchronous system calls using the application benchmark. To this end, we implement a small microbenchmark applications that executes a `pwrite` system call to a files on `tmpfs` in a loop, measuring the throughput of system calls similar to Figure 2.3. This benchmark runs with multiple threads, one per dedicated writer core, and each core is writing to a different file to prevent contention in VFS when accessing the same file. SCONE version runs one userspace thread per writer core, but with a sibling hyperthread (not included into the writer core count) dedicated to system call threads, thus not getting any benefit from M:N threading. We run 5 system call threads on each of those dedicated cores.

The results are presented on Figure 3.12. We can see that with small buffer sizes, SCONE runs with a performance close to native, 24% faster than native in the best configuration and 35% slower than native in the worst. The performance degrades when the number of writer cores reaches 8 for SCONE because in this case there is a contention between system call thread and enclave threads, as the benchmarking machine has only 12 cores. The same benchrmark running with SCONE in synchronous mode consistently shows throughput that is 4x-7x lower than that of the native version. These results prove that for system calls which do not involve significant amount of data copying, asynchronous system call is a critical optimization.

On the other hand, with large `pwrite` buffers, the overall performance difference between the native version and SCONE running both in synchronous and asynchronous mode. SCONE runs within 35–84% of the native throughput, while in synchronous modes the performance is comparable (39–80%). Most of the overhead of SCONE in this case comes from buffer copying, negating the benefits from the asynchronous system call interface, which

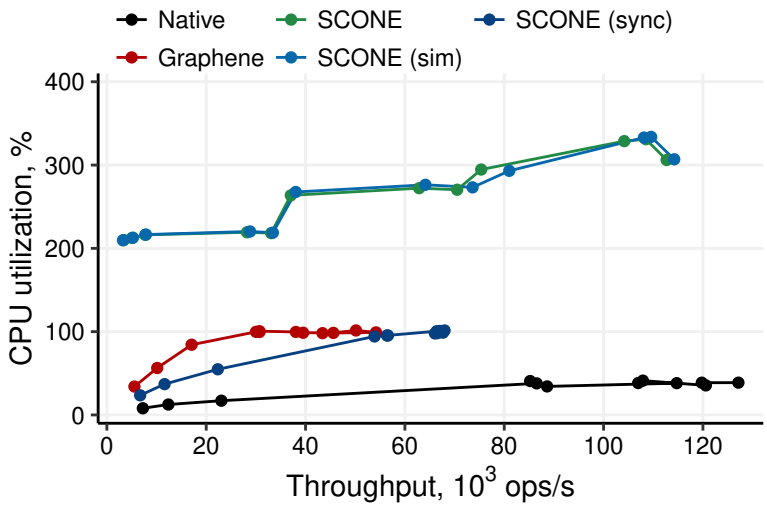

Figure 3.9: CPU utilization of Redis running as native applications and with different SGX runtimes.

requires extra cores to run. Furthermore, when there is contention in asynchronous system call interface, as can be seen with 8 threads, the synchronous interface with 8 threads actually outperforms the asynchronous version.

The results of these and similar measurements can be used to guide the design of a hybrid system call interface. However, we leave the design and implementation of such an interface to future work.

3.5 Discussion

SCONE as library OS? In SCONE, we have implemented only a small number of system calls inside the enclave to gain the required runtime support. These include system calls for thread management and scheduling, memory management, and signal support. As this functionality is commonly implemented inside the operating system kernels, it could be argued that the line between the approaches to enclave construction based on library operating systems and based on the libc is extremely blurry. It is thus possible to classify SCONE as a very small library OS.

On the other hand, it would be a very minimal library OS, as its minimal emulation approach does not allow some use-cases, that are possible with SGX-LKL and Graphene-SGX, the main of which is portability. SCONE can be used to run POSIX applications on Linux only, while the aforementioned runtimes, by the virtue of their design, allow executing the same enclaves on other operating systems, like FreeBSD. It should be noted, however, that given the wide reliance on *Linux containers* for deployment of software makes this use-case not widely pursued.

Other SCONE features. We have not described all features available in SCONE, only focusing on those, that were developed with an active participation of the author of this thesis. Some of the other features of SCONE include:

- Network shield that transparently wraps network connections with TLS encryption.

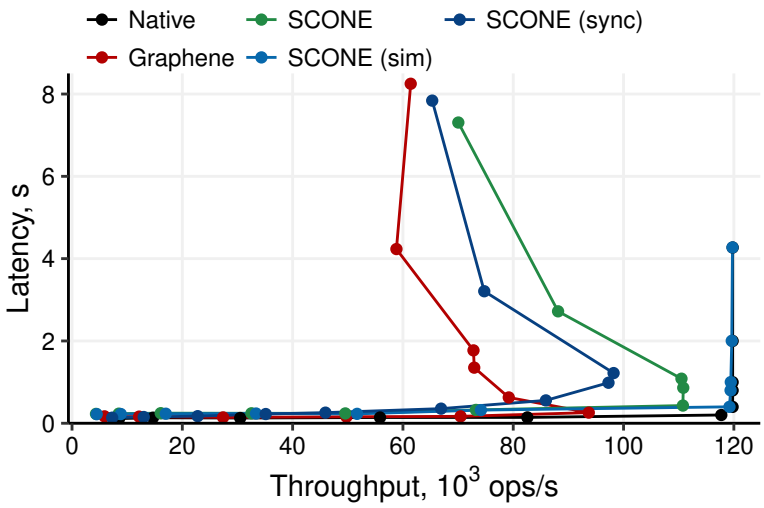

Figure 3.10: Throughput-latency measurement of Memcached running as a native application and with different SGX runtimes.

- File system shield implements emulation of VFS interface inside SCONE [178]. This allows creating virtual file system hierarchies visible only to the enclave, which in turn can be backed by enclave memory or encrypted and integrity-protected on-disk file.
- SCONE contains enclave loading and signing infrastructure implemented independently from Intel SGX SDK.
- Secure bootstrapping of Intel SGX enclaves requires provisioning secrets to them after successful attestation. To simplify this task for SCONE-based enclaves, Palaemon, a Configuration and Attestation framework has been developed [138]. It integrates with the file system shield, and allows performing in-memory updates to configuration files present on disk or inside shielded file system images.
- `fork` and `exec` system call support was implemented.
- System call threads can be spawned and terminated dynamically to achieve lower CPU utilization in the periods of idleness and to remove the need to configure the number of pre-spawned system call threads.

SCONE Kernel Module. Originally, we have implemented a kernel module that executed system calls inside the kernel continuously, without entering the userspace for a prolonged periods of time. However, we have gradually deprecated this kernel module because of several issues we have discovered in practice:

- SCONE kernel module was unlikely to be accepted into the mainline Linux kernel, and shipping it to user machines was incompatible with container deployments.
- Performance speedup from switching to FFQ algorithm and a system call queue pair per core had much bigger effect on performance than the system call kernel module.
- SCONE kernel module implementation did not support multiple enclaves running simultaneously on the same machine.

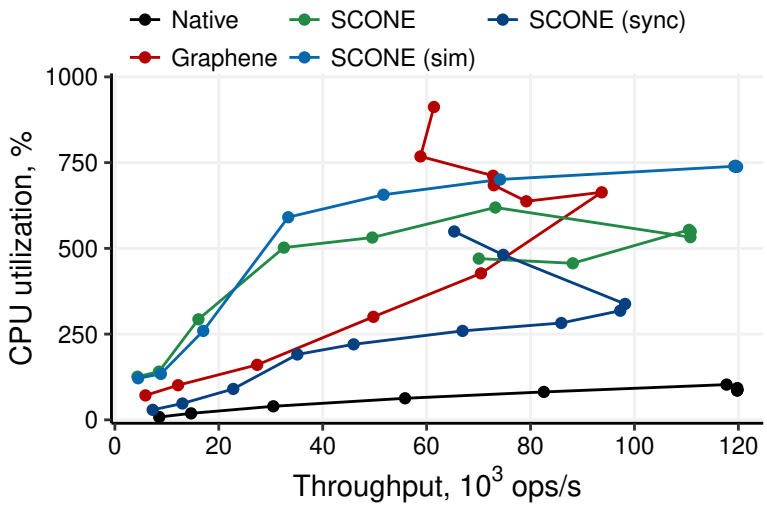

Figure 3.11: CPU utilization of Memcached running as a native application and with different SGX runtimes.

- SCONE kernel module made debugging more complicated, as it was incompatible with the system call tracing utilities like `strace`.

As a result, we have gradually deprecated the kernel module, so that the system call threads invoke the forwarded system calls in userspace, and switch to the kernel using the standard libc code.

Performance optimizations. One of the sources of overhead that is unavoidable with Intel SGX is the extra memory copy between the enclave and untrusted environment that happens on system calls. It has been shown that this cost can be eliminated for RISC-V-based Keystone enclave by extending its implementation in the M-mode [302]. However, as Intel SGX does not use physical memory regions to build a security boundary between the enclave and untrusted world, and relies on TLBs for this purpose instead, mechanisms developed for opimizing performance of microkernels may be a better fit for SGX [112].

3.6 Related Work

We have already presented the fundamental background on Intel SGX in the §2.4. In this section, we will provide a short background on the asynchronous system call interfaces.

The paper that directly served as an inspiration for SCONE was FlexSC [277]. It is based on a modified Linux kernel extended with communication mechanisms (mailboxes) and in-kernel system call threads. The runtime of applications is extended to operate both synchronously and asynchronously, switching between these two modes depending on the workload, and reducing the system CPU load from the kernel system call threads. Futhermore, FlexSC allows system call batching, and wakes the system call threads only when several outstanding calls accumulate. However, some of the design choices of FlexSC are unsuitable for SCONE: SCONE strives to be practically usable in Linux containers, which precludes modification of Linux kernel. Additionally, SCONE is not explicitly optimized in the area of CPU utilization: while system call threads can back-off in the idle state, we have not explored further optimizations for CPU utilization like the adaptive mode, which would be beneficial as enclave

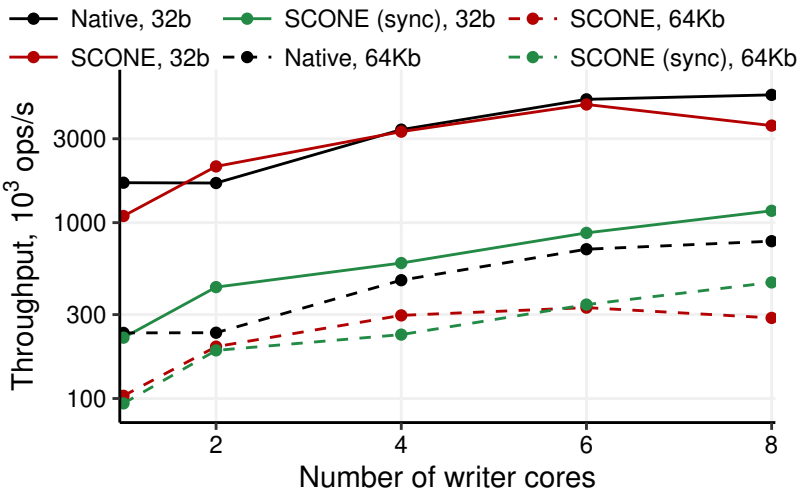

Figure 3.12: Throughput of a `pwrite(2)` microbenchmark with different number of writer cores and write buffer sizes.

applications are likely to constitute only a small share of all applications running on the cloud node.

A system similar to FlexSC is MegaPipe [146], which provides an API for writing efficient and scalable networking applications which handle short messages and sessions. Similarly to FlexSC, MegaPipe relies on batching of system calls to eliminate per system call overhead. On the other hand, it requires application changes to use a new API, and cannot be used by legacy applications transparently. Compared to FlexCS, MegaPipe does not support batching of arbitrary system calls, and does not rely on asynchronous communication, retaining the synchronous system call nature. It forms an interesting point of comparison with SCONE, as these two approaches give most benefits for two opposite classes of applications: SCONE is most suited for applications that spawn a thread per connection, or at least a large number of threads, while MegaPipe is designed for applications that spawn a thread per core or run single-threaded—otherwise, it loses the benefits of system call batching.

These two approaches are reconciled in the modern system call API available in the recent Linux kernel versions, namely `io_uring` [74]. Like MegaPipe, it requires an explicit use of the new API, forcing modifications to the legacy applications. However, it allows spawning an in-kernel thread for system call processing, avoiding userpace-kernelspace round-trip. Unfortunately, the number of operations that are currently supported by this API is still limited, focusing on network and file system operations; however, the list of supported APIs is increasing from version to version. In future, SCONE may forward some system calls to `io_uring`; however, this would introduce extra complexity to the system call interface, and lock SCONE to the most recent Linux kernel versions.

3.7 Conclusions

In this chapter, we introduce SCONE, an Intel SGX enclave runtime for unmodified POSIX applications. We present the design decisions that allow it to avoid limitations of Intel SGX: asynchronous system call interface allows avoiding enclave entry and exit overhead, while M:N

threading eliminates the need to preallocate TCS for in-enclave threads. Minimal emulation inside the enclave allows to keep the TCB of SCONE slim while maintaining the correctness of executed applications. SCONE achieves 8-25% overhead comparing to native applications, and compares favourably in performance with another framework we have chosen for comparison (Graphene-SGX). SCONE is the foundation on which all further work in this thesis is built.

We have seen that asynchronous system call interface is one of the key features that makes SCONE a viable foundation for the high-performance cloud software. However, the initial design [71] was initially using an off-the-shelf MPMC queue that was posing as a performance bottleneck. In the next chapter, we present the design of the concurrent queue that is powering the current SCONE version.

4 FFQ: Fast FIFO Queue

In the previous chapter, we have shown how SCONE uses an asynchronous system call interface to improve its latency and throughput. As we have mentioned, initially, SCONE used an off-the-shelf multiple producer multiple consumer (MPMC) queue for its system call interface. In this chapter, we explain the motivation, design, and implementation of the concurrent single producer multiple consumer (SPMC) queue that replaced the original MPMC design, aiming to address the problem of *how to maximize the throughput of a concurrent, lock-free FIFO queue*.

4.1 Introduction

SCONE relies on the concurrent communication between enclave threads and system call threads to sidestep the performance impact of entering and exiting the enclave. As software serving the cloud user is often I/O-bound, the performance of this communication mechanism should be maximized: the throughput should be as big as possible, and the latency as low as possible.

As strawman solution to this solution could involve an array of per-userspace-thread structures with system call arguments and return values, protected by a POSIX thread mutex. However, this method has a number of drawbacks:

- To use a pthread mutex, enclave still has to submit a system call, such as `futex` on Linux, causing chicken-and-egg problem.
- System call threads concurrently scanning an array and enclave threads writing to it would experience low cache locality and cause superfluous cache coherency traffic between cores.
- Finally, the scalability of mutexes is limited and is likely to become a bottleneck on the modern multi-core CPU architectures.

A combination of these factors also disqualifies a large number of other, better designed data structures, such as mutex or spinlock protected ring buffers, list-based queues, and so on.

In this context, concurrent lock-free queues are one of the most important mechanisms to consider, as their easy-to-use interface makes them a default choice for a large number

of high-performance applications. Due to their lock-free nature, their performance scales better with the number of cores than lock-based queues, and they do not require issuing of system calls for their operation.

A typical application that utilizes concurrent queues contains threads that put new information into the queues (*producers*), and threads that read information from the queues (*consumers*), and either process it or put it into the further queues. A typical requirement is that the items in the queue are enqueued and dequeued in order: such queue is said to be First-In First-Out (FIFO) queue. In SCONE, both system call threads and enclave threads work as consumers and producers, because each thread that enqueues items into the system call submit queue, dequeues the processing results from the return queue, and vice versa.

Additionally, the implementations of concurrent queues are classified by whether they allow multiple producers and consumers to perform operations on the queue simultaneously. Thus, it is possible to implement multiple- or single-producer queues (MP or SP-queues), and likewise for consumers, it is possible to have multiple- or single-consumer queues (MC or SC-queues).

There are several aspects that affect the performance of the concurrent queue. First and foremost, it is the contention over shared cached lines for the multiple consumer and producer variants. Modern multi-core CPUs implement a cache coherency protocol between cores, which synchronizes the information stored in each of the CPUs with the contents of the main memory. Each write to a cache line that is *shared* or is in *exclusive* ownership of another core causes a dispatch of an invalidation message to other cores. These invalidations slow down the execution of both writer and victim cores.

A more complex example of performance loss stemming from cache coherency protocol is so-called *false sharing*. In the case of false sharing, even though the writes and reads a targeting not the same, but merely adjacent addresses, the conflicting accesses still target the same cache line, causing the same performance degradation. More importantly, false sharing can appear not only due to direct accesses to the same cache lines, but also due to the operation of prefetcher on each of the cores.

Additionally, more complex MPMC queues typically involve execution of more instructions, which may also include some rare, complex and unoptimized instructions, such as double compare and swap, which atomically and conditionally swaps the contents of two memory locations of double register width. Another important factor is cache misses, which can slow down a single producer or consumer when dequeueing the items.

It is important to take all the abovementioned aspects into account when designing a new queue algorithm. However, it is also important to consider application-specific aspects, that may allow us to relax some of the restrictions of the concurrent queue interface.

In SCONE, we have originally used an off-the-shelf generic MPMC queue. However, we observe that in MPMC concurrent queues, items that are submitted by different producers are ordered, which reduces the scalability by forcing different threads to synchronize on enqueueing. In SCONE, however, the items enqueued by *different* producers do not necessarily need to be FIFO ordered.

To illustrate the scalability issues of the MPMC queue, we have created a simple microbenchmark (Figure 4.1). Several enclave threads, each pinned to a different core, submit `getppid` (get parent process id) system call in a loop. We measure the number of system call submissions per second over all cores, while changing the number of the cores dedicated to the enclave threads (the higher rate the better). This experiment shows that even for a small number of OS threads executing `getppid` system calls running inside of an SGX enclave, the throughput is already lower than the native system call performance of glibc. Even worse,

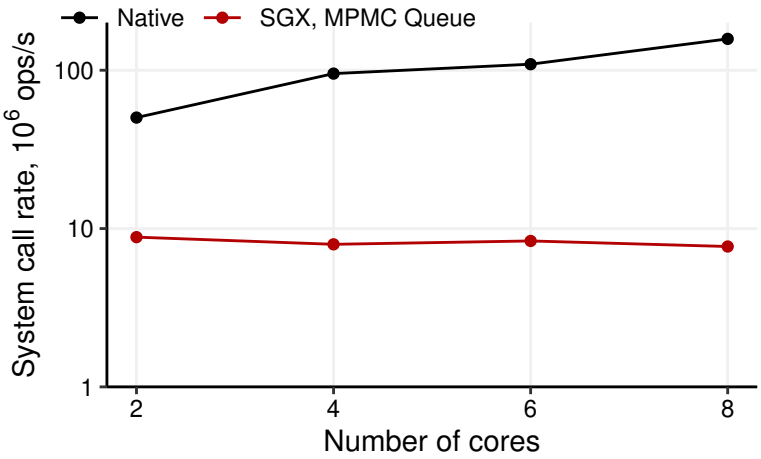

Figure 4.1: System call throughput of a native application and SCONE enclave with MPMC queue. Using an MPMC FIFO queue to submit system calls does not scale with the number of threads, while the native system calls (executed outside of enclaves) scale well.

when adding more OS threads, the throughput does not increase. While there are inherent costs associated with running inside SGX enclaves, our objective is to reduce the overhead introduced by the FIFO queue as much as possible, and in particular, ensure scalability with the number of threads. Furthermore, both state of the art wait-free queues like the ones evaluated in [298], as well as queues using hardware transactional memory (HTM), showed limited scalability in our application context. For example, wait-free queues tend to require complicated memory reclamation schemes, as they are designed for general use-cases and fail to take into the specific workload characteristics of SCONE.

Since existing solutions did not help to solve our performance problem, we propose and evaluate a new FIFO queue, FFQ. Our design decisions were guided by our application context and our main objective was to maximize throughput. Nevertheless, we strongly believe that our solution and the insights that we gained are sufficiently general that FFQ can be applied in other contexts.

Our optimizations are based on the following observations about SCONE:

1. *Single producer, multiple consumers.* The system calls do not need to be ordered between different application threads. Hence, we can use a single-producer/multi-consumer (SPMC) FIFO queue instead of a more generic multi-producer/multi-consumer (MPMC) queue to issue system calls. We need, however, to support multiple consumers since some of them can be blocked while executing a system call.

2. *Implicit flow control.* Each application thread can have at most one outstanding system call. Hence, we can dimension the length of a FIFO queue such that, for each enqueue, we are guaranteed to find an empty slot in the queue. By default, SCONE system call queue has 1024 entries.

3. *Wait-free enqueue, lock-free dequeue.* It is important that all system calls are executed with minimum delay. Hence, we would like a producer to enqueue elements in a wait-free manner. However, it does not matter which of the consumer threads actually

executes the system call. We therefore only require the dequeue function to be lock-free.

4. *Progress with intrinsic read-modify-write operation.* Some intrinsic operations, such as in particular "get-and-increment", are wait-free and can hence be used to guarantee forward progress, despite their relatively limited synchronization power.[1]

As a consequence, we propose and evaluate a new single-producer/multi-consumer FIFO queue algorithm (FFQ). While the motivation and the design of our algorithm are based on our application context, we argue that our algorithm is applicable also in other contexts in which a high throughput queue is needed. Moreover, we contribute to the state of the art in the following ways: (a) we study and exhibit the bottlenecks of existing algorithms, and we use these observations to optimize the performance of our design; (b) we perform a comprehensive evaluation of our algorithm and compare it with existing approaches; and c) we provide key insights in what matters most, and what is less critical, for achieving good performance.

Our presentation of FFQ is structured as follows: we review related work in §4.2 and we introduce our algorithm in §4.3. We then describe low-level performance optimizations in §4.4. We evaluate the algorithm and compare it with other state-of-art concurrent queues in §4.5 and finally conclude in §4.6.

4.2 Related Work

There has been a number of concurrent queues presented starting with Lamport's basic ring buffer [192]. We present an overview of the most influential designs that are used in practice.

The FastForward [132] single-producer/single-consumer (SPSC) queue was designed to improve the performance of pipeline-parallel applications. It uses temporal slipping to avoid cache thrashing and hardware cache prefetching, and supports systems with a range of memory consistency models. In practical terms, however, slipping requires system-specific tuning and causes thrashing by touching queue head and tail pointers. Unlike FastForward, FFQ is an SPMC queue and has no system-specific parameters.

MCRingBuffer [198] is an extension of Lamport's basic ring buffer with the goal of improving cache locality of control variables. This is achieved by batching updates to control variables. MCRingBuffer is data-generic and has no special data values that are used for control purposes.

BatchQueue [246] was designed to improve performance of OpenMP pipeline parallelism extensions [245] by replacing native MPMC queues with a specialized SPSC variant. It simplifies the design of MCRingBuffer by using fewer control variables. BatchQueue avoids false sharing by isolating the producer and the consumer in different parts of the queue.

B-Queue [292] improves the design of FastForward and MCRingBuffer by adding a backtracking algorithm for deadlock detection due to producer and consumer batching. It avoids using parameters that require system-specific tuning, simplifying its usage in real-world applications. In contrast to FFQ, it is a batching SPSC queue, while FFQ is designed to be used in practice in the SPMC configuration without batching.

[1] For instance, get-and-increment has a consensus number of only 2, i.e., it can solve the wait-free consensus problem for no more than two concurrent processes [148, 149].

Lynx [221] is an SPSC queue that focuses on removing check overheads from the enqueue and dequeue fast path. To that end, it inserts pages with special page fault semantics within section boundaries and at the end of the queue. This specialized design would have high costs for our target application scenarios when deployed inside an SGX enclave: signal delivery would cause an enclave exit—which takes up to 50,000 cycles.

Michael and Scott [220] provide a non-blocking list-based unbounded MPMC queue algorithm. It has a simple design that relies on compare-and-swap operations in the non-blocking variant. This queue does not scale well in practice due to contention on the tail and head pointers.

David [110] proposes a single-enqueuer wait-free queue implementation that shares similar design goals with FFQ but is mostly of theoretical interest. In particular, it has unbounded memory requirements as it relies on a two-dimensional infinite array of *swap* objects and a one-dimensional infinite array of *fetch-and-increment* objects. Even though the author gives some hints on how to reduce the memory footprint, the design is not practical and, to the best of our knowledge, has not been used for actual queue implementations.

CC-Queue [119] is an extension of Michael-Scott's queue that uses combining synchronization [118] instead of locks in the two-lock variant of the algorithm. This technique allows better scalability than compare-and-swap operations and traditional locks. However, it is a bad fit for SCONE where the dequeuing thread may block on the system call, breaking the combining construction.

LCRQ [225] is an unbounded MPMC queue that improves performance and scalability over Michael-Scott's queue and CC-Queue by using fetch-and-add atomic operations. This ensures that each operation on the queue makes progress.

WFQueue [298] provides a wait-free, unbounded MPMC queue that also relies on fetch-and-add operations, hence avoiding CAS retries. WFQueue has a lower performance overhead than other wait-free queues in most cases. It uses a fast-path/slow-path approach and can be tuned based on several control parameters.

Maffione et al. have presented improved variants of the Lamport queue and the Fast-Forward queue [206]. The fundamental idea of the improved variants is that the queue performance depends on the cache misses at producers and consumers, and minimizing their number can significantly increase the queue throughput. The techniques for minimizing cache misses include batching, embedding control information into the queue slots, performing operations lazily.

FFQ has been designed primarily for SPMC scenarios with the assumption that the maximum number of elements in the queue is known beforehand, which is the case for SCONE. This allows us to improve performance owing to a simpler algorithm and specialized optimizations, while still achieving high throughput in MPMC settings.

4.3 The Algorithm

In this section, we describe our *fast FIFO queue* (FFQ) algorithm. We first introduce the key idea underlying the single-producer variant (FFQ^s), before discussing in depth its operating principles, implementation details, and various optimizations. We subsequently extend the algorithm to also support multiple producers (FFQ^m).

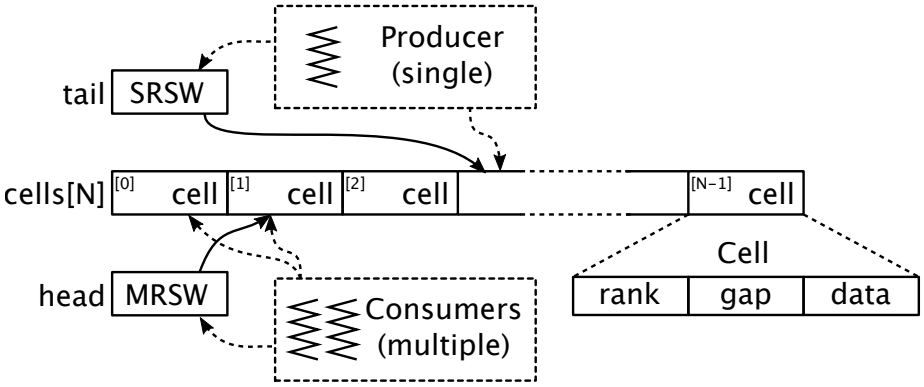

Figure 4.2: Data structures of the FFQ algorithm. The tail of the queue is Single-Reader Single-Writer, while the head is Multiple-Reader Single-Writer.

4.3.1 Single Producer

FFQ implements a concurrent FIFO queue. The basic version has been designed for a single producer and multiple consumers. The rationale behind the single-producer assumption is to maximize speed and limit—as much as possible—the synchronization overheads. Furthermore, the queue has been developed for use in a context where the processing time of consumers that dequeue data largely dominates the time necessary for producing a new item, hence the single producer never represents a bottleneck. It is, however, important that the queue never blocks the producer, i.e., the enqueue operation should ideally be wait-free, or at least lock-free.

To support the FFQ algorithm, we rely on a set of data structures as illustrated in Figure 4.2. Shared data is stored in a bounded array whose size N is large enough so that *there will always be some empty slot for the producer to enqueue a new item.* The motivation for this assumption is our observation that in SCONE, we have implicit flow control that ensures that consumers dequeue items sufficiently fast such that the array will never fill up completely.[2] The array is managed as a circular buffer, i.e., indexes are computed modulo N such that the last element logically precedes the first one. Furthermore, the provisioned queue is typically much larger than the number of threads, and the threads that have submitted a system call are blocked until they receive the reply. Thus, if the number of cells is larger than the number of in-enclave threads, there will always be a free slot in the queue.

Two integer variables are used to keep track of the head and tail of the queue. They behave as monotonically increasing integer counters that hold the *rank*, i.e., the insertion number, of the first and last data items currently in the queue. No two items can have the same rank but, in some situations (discussed later), some ranks may be skipped hence leaving *gaps*, i.e., unused values, in the numbering. Ranks are mapped to array elements using modulo arithmetic: the item with rank k is located in the element at position (k mod N). To denote ranks, we only use numbers such that $|rank| \geq N$, so that we can reserve lower values to indicate special cell states. We currently only use value 0 for a special purpose, hence the size N of the array must be at least 1.

As there is a single producer thread and items are inserted at the tail of the queue, the *tail* variable is not shared (single-reader/single-writer register). In contrast, several consumers

[2] If this assumption would be violated, the producer would spin until a slot becomes available, i.e., the *enqueue* operation would not be wait-free anymore.

Algorithm 1 — FFQ^s: single-producer FIFO queue

```
 1: Type cell is:                                          ▷ Cell for holding data
 2:     data ← NULL                                       ▷ Actual data (initially empty)
 3:     rank ← 0                                          ▷ Rank of item (or -1 if cell unused)
 4:     gap ← 0                                           ▷ Gap in numbering (skipped item)

 5: Variables:
 6:     cells[N] ← array of cell                          ▷ Bounded array of cells (N ≥ 1)
 7:     tail ← N                                          ▷ Tail counter (monotonically increasing)
 8:     head ← N                                          ▷ Head counter (monotonically increasing)

 9: function ffq_enq(data)                                ▷ Enqueue (single-producer)
10:     success ← FALSE
11:     while ¬success do                                 ▷ Find empty cell...
12:         c ← cells[tail(mod N)]                        ▷ Try next cell
13:         if c.rank ≥ N then                            ▷ Cell used?
14:             c.gap ← tail                              ▷ Yes: skip it (gap in rank)
15:         else
16:             c.data ← data                             ▷ No: grab it
17:             c.rank ← tail                             ▷ Remember rank
18:             success ← TRUE
19:         tail ← tail + 1                               ▷ Move to next cell

20: function ffq_deq                                     ▷ Dequeue (multi-consumers)
21:     rank ← fetch-and-inc(head)                        ▷ Get rank of next item
22:     c ← cells[rank(mod N)]                            ▷ Check associated cell
23:     success ← FALSE
24:     while ¬success do                                 ▷ Find next used cell...
25:         if c.rank = rank then                         ▷ Cell used for rank?
26:             data ← c.data                             ▷ Yes: get item
27:             c.rank ← 0                                ▷ Recycle cell
28:             success ← TRUE
29:         else if c.gap ≥ rank ∧ |c.rank ≠ rank| then
30:             rank ← fetch-and-inc(head)                ▷ Cell skipped: ...
31:             c ← cells[rank(mod N)]                    ▷ ...move to next cell
32:         else wait()                                   ▷ Back off (producer still writing cell)
33:     return data                                       ▷ Return item
```

may concurrently access the head of the queue and, hence, *head* is a shared atomic variable (multi-reader/multi-writer register).

Each cell consists of three fields: *data* holds a reference to the actual data enqueued by the producer; *rank* corresponds to the rank of the item stored in the cell, if any, zero if the cell is unused, or a negative value if the item is being inserted; *gap* may hold the rank of an item that has been skipped, i.e., a gap.

A gap can occur in the following situations: Producers enqueue elements sequentially at the tail and consumers dequeue them from the head. FFQ uses a bounded buffer implemented as a circular array and, despite our assumption that there will always be *some* empty slot for enqueuing an element, the next slot where a producer will try to store its element at rank r_1 might not be free because a consumer started but did not complete dequeuing an older element at rank $r_2 < r_1$ with (r_2 mod N) = (r_1 mod N). In such a case, the producer will simply skip r_1 and move to the next rank ($r_1 + 1$), hence creating a gap that is announced in the cell. At that point, both the *rank* and *gap* fields of a cell can be set to valid rank values. As a matter of fact, the same cell might be skipped multiple times due to a very slow consumer, in which case *gap* is set to the last rank that was skipped.

The pseudo-code of FFQ is shown in Algorithm 1. ffq_enq() is designed to be simple and

fast, with as little synchronization as possible. Since there is a single producer and the array is assumed to always have some empty slot, the algorithm is relatively straightforward. In contrast, ffq_deq() may be called concurrently by multiple consumers and, hence, necessitates additional synchronization operations to order and manage conflicts between threads.

Enqueuing Items

To enqueue a new item, the producer first tries to insert its new data at the tail of the queue. To that end, it checks if the tail cell is free by reading the *rank* field of the cell. There are two cases to consider: *(i)* either *rank* has a zero value indicating that the cell is free and any data it previously held has been consumed, *(ii)* or it contains a valid rank value indicating that the cell holds data that was not yet consumed.

In the first case, the producer stores a reference to the enqueued data and then sets the *rank* field to the current value of *tail* to indicate that the cell has been used (Lines 16–18). Note that the order of the two operations is important as the latter announces the availability of data to consumers and represents a synchronization (and linearization) point.[3]

The second case is subtler. Because of our assumption on the array to always have some empty slot and the FIFO nature of the queue, it means that a slow consumer has started dequeuing this item but did not complete its operation.[4] Therefore, we cannot use the cell for storing a new element at rank *tail*. We simply skip the cell, hence, creating a gap in the numbering. The key insight in our algorithm is that, for performance reasons, we do not want to change the efficient "modulo" mapping of ranks to array cells. Instead, the producer announces that the current rank is unused by setting the *gap* field of the cell to the current value of *tail* (Line 14). This will let consumers know, upon dequeue, that they have to move to the next rank to find the following element in the FIFO sequence.

Finally, as the *tail* variable is not shared, the producer can safely increment it without synchronization. The ffq_enq() operation returns successfully if it has managed to insert the data, otherwise, it continues looking for an empty cell by traversing sequentially the array.

Dequeuing Items

The ffq_deq() function can be called concurrently by multiple threads, hence, special care has to be taken for synchronizing consumers. First, the *head* variable is atomically incremented to provide each distinct consumer a unique rank number where to look for the next item to dequeue (Line 21). The consumer locates the cell associated with its assigned rank number (Line 22) and checks if it contains data for that rank (Line 25). Note that multiple consumers might be accessing the same cell because of the asynchrony of the systems and the bounded array size, but at most one consumer will have a rank equal to the value stored in the cell's *rank* field, and only in that case the consumer may dequeue and modify the cell.

If the cell contains the expected element, i.e., $cell.rank = rank$, the consumer reads the associated data and resets the cell's rank to the special value 0 to allow the producer to reuse it (Lines 26–28). Note again that the order of operations is important as resetting the cell's rank represents a synchronization (and linearization) point.

If the cell's rank differs from the expected rank, this may indicate that the cell has been skipped by the producer as its content was still being dequeued by a slow consumer at the

[3] Ordering is enforced in the actual implementation using memory barriers.

[4] Note that, at that point, the following relation holds: $cell.rank < tail \wedge (cell.rank \bmod N) = (tail \bmod N)$.

time of insertion. In that case, the cell's *gap* field must be equal to the expected rank, or possibly to a higher rank (if the original gap announcement has been superseded by other announcements N positions apart after rolling over the end of the buffer). The consumer hence checks for this condition and, if true, moves to the next available cell by acquiring again a unique rank from the atomic *head* variable (Lines 29–31). Note that, at Line 29, we need to verify again that $|cell.rank| \neq rank$ because the producer might have inserted the expected element after a slow consumer has performed the check of Line 25 and, while the consumer was idle, quickly inserted many new elements to eventually skip the same cell upon subsequent array traversal (hence announcing a gap for a higher rank). It is necessary to compute absolute value of *cell.rank* because we use negative rank values to allow producers to indicate that they are in the process of inserting an item.

Finally, if the cell does not indicate that it contains the expected element nor that the considered rank is a gap, this means that the next element to be dequeued has not yet been (completely) enqueued. Hence the client backs off and waits for the element to be available or the cell to be skipped for the considered rank.

Proposition 1. *Assuming the queue is not full, the* ffq_enq() *operation is wait-free.*

Proof (sketch). Consider that there is a single producer iterating through the array and the only situation when it cannot immediately store its data is when the next available cell is still busy, i.e., the item has not yet been dequeued. Under the assumption that there is always some empty spot in the array, i.e., consumers are sufficiently fast at dequeuing items, this case can only arise if a consumer has started dequeuing the item but not yet finished. Hence, from our assumption there must be at least one other cell in the array whose item has been fully dequeued and that is empty. Since we have a single producer, it will eventually reach this cell and enqueue its item, independently of the actions of the other threads (which are all consumers). □

Proposition 2. *Assuming there are elements to dequeue, the* ffq_deq() *operation is lock-free.*

Proof (sketch). Observe first that the code only uses atomic fetch-and-increment operations, which are non-blocking. It does not use locks and the only condition when it can block is if it repeatedly executes the loop at Lines 24–32 (the wait operation at Line 32 is not blocking, it may simply delay the thread for a few nanoseconds). The loop is repeated either if a producer is still writing to the cell (Line 32) or if the cell was skipped (Line 29). In the first case, the consumer backs off and wait. Since we have a single producer, this can only happen if all previous elements have been (or being) consumed and the queue is empty, hence contradicting our assumption that there are elements to dequeue. Therefore a producer that stops making steps after line 16 (or elsewhere, or even crashes) cannot block a consumer that dequeues an element with a lower rank. In the second case, the cell was skipped and the consumer moves to the next cell in the array. Because of our assumption that there is always some empty spot in the array, and hence the producer can always enqueue new items, it is not possible for all consumers to encounter only skipped cells. Hence some of the consumers will manage to dequeue an element and the ffq_deq() operation is therefore lock-free. □

Proposition 3. *The FFQ object is linearizable.*

Proof (sketch). To show that the FFQ object is linearizable [150], we identify the point(s) in ffq_enq() and ffq_deq() where the operation atomically takes effect. In ffq_enq(), this point is

Algorithm 2 — FFQ^m: multi-producer FIFO queue

```
 1: function ffq_enq(data)                                  ▷ Enqueue (multi-producer)
 2:     success ← FALSE
 3:     while ¬success do                                   ▷ Find empty cell...
 4:         rank ← fetch-and-inc(tail)                      ▷ Get next rank...
 5:         c ← cells[rank(mod N)]                          ▷ ...and associated cell
 6:         while (g ← c.gap) < rank do                     ▷ Unless overtaken...
 7:             if (r ← c.rank) ≠ 0 then                    ▷ Cell used?
 8:                 double-compare-and-set                  ▷ Yes: skip it
        .....(⟨c.rank, c.gap⟩, ⟨r, g⟩, ⟨r, rank⟩)            ▷ ⇒ Set gap
 9:             else if double-compare-and-set              ▷ No: use it
        .....(⟨c.rank, c.gap⟩, ⟨0, g⟩, ⟨−rank, g⟩) then      ▷ ⇒ Set rank
10:                 c.data ← data                           ▷ Store data
11:                 c.rank ← rank                           ▷ Remember rank
12:                 success ← TRUE
```

the time where the new element becomes visible to consumers. This happens at Line 17 when the producer updates the *rank* field after having previously stored the data. Indeed, consumers use the *rank* field as synchronization point to detect availability of an element (Line 25) or a gap (Line 29).

In ffq_deq(), the linearization point is not attached to a single line of the code. The last fetch-and-increment operations (Line 21 and Line 30) of a ffq_deq() call logically order the consumers. If the element that ffq_deq() returns is already enqueued at the time when the fetch-and-increment operation takes effect (i.e., linearization point of ffq_enq() has already happened), the fetch-and-increment operation is the linearization point. If the linearization point of the matching ffq_enq() operation has not yet taken effect, we define the linearization point of ffq_deq() to happen immediately after the linearization point of the matching ffq_enq(). Since the ffq_deq() will spin until that consumer provides an element, the linearization point is indeed before ffq_deq() returns. □

4.3.2 Multiple Producers

We now present the modifications to the basic algorithm for supporting multiple producers. Obviously, the resulting algorithm, FFQ^m, will incur extra synchronization overheads, which will translate into lower performance.

The new version of the ffq_enq() function, shown in Algorithm 2, essentially differs from the single-producer variant in that it now uses an atomic increment to acquire a unique rank where to store a newly produced item (Line 4). Yet, the atomic increment is not sufficient by itself as one can run into subtle race conditions that affect correctness. Consider the case of two producers, p_1 and p_2. Assume p_1 acquires a unique rank r_1, verifies that the cell is unused, and goes to sleep (e.g., due to its thread being preempted). After some activity on the queue with elements produced and consumed, p_2 acquires a unique rank $r_2 > r_1$ that maps to the same cell, i.e., (r_2 mod N) = (r_1 mod N), and stores its data in that cell. Finally, p_1 wakes up and proceeds with updating the cell, essentially overwriting p_2's data without noticing the conflict.

Another problem is that producers might actually enqueue elements "in the past". Consider a similar scenario of two producers, p_1 and p_2, which respectively obtain ranks r_1 and $r_2 > r_1$ that again map to the same cell. Assume that p_2 executes first, observes that the cell is used ($c.rank = r$ with $0 < r < r_1$), and thus skips it by setting $c.gap$ to r_2. Then consumer c_1 with

rank r_1 comes, observes the gap, and hence skips it since $c.gap\ (\equiv r_2) > r_1 \wedge c.rank\ (\equiv r) \neq r_1$ (Line 29 of Algorithm 1). Subsequently, the "slow" consumer c with rank r completes its dequeue operation and clears the rank (Line 27 of Algorithm 1), allowing p_1 to enqueue its element with rank r_1 that was skipped by c_1. This ultimately results in the production of an element that will never be dequeued. To solve this problem, one should disallow producers from enqueuing items in the past, i.e., with $rank \leq c.gap$.

We need therefore additional synchronization to handle such conflicts between producers. To that end, we use an atomic "compare-and-set" operation to update the *rank* field of the cell, by attempting to atomically change it from the expected value of 0, which means that it is free, to another value indicating that it is used. The problem is that, if we directly set the new value to the rank, we might run into another race condition—but this time with consumers. Indeed, synchronization between producers and consumers relies on the former to first update the *data* field, and only then the *rank* field so as to let the latter know that the element can be read. Hence we use another special value (–*rank* in the pseudo-code) as the new value for rank in the double-compare-and-set to synchronize the producers, before subsequently setting the *rank* field to its final value. Note that we cannot use a special constant here (–1, –2, etc.), because this could lead to a subtle race condition where one fast producer would catch up with a slow producer and both will deadlock trying to insert data in the same cell. Since no two producers can get the same rank, this guarantees that these negative values will be distinct for two different producers.

Still, this is not sufficient to solve the second problem as another producer might concurrently change the *gap* field of the cell to a value higher than the acquired rank, which is exactly the scenario we need to avoid. We therefore use a double-word version of the compare-and-set operation to ensure that no gap is created while we update the rank (Line 9). Note that double-compare-and-set can be supported by simply using a 128-bit version of the compare-and-set operation (available on most modern processors) and placing the *rank* and *gap* fields consecutively in the same cache line. If the compare-and-set succeeds, the algorithm proceeds with updating the cell, otherwise it means that another producer has taken over the cell or inserted a gap in the meantime and we need to retry acquiring the cell.

If the cell is used, i.e., its *rank* field is non-negative, we skip it and create a gap in the numbering. To update the *gap* field of the cell, we also need to use a double-compare-and-set operation to avoid setting its value "back in time" in case it has since been updated to a larger value by another producer, while at the same time making sure that no other producer has concurrently enqueued an element (Line 8). Note that if the double-compare-and-set operation succeeds, i.e., the gap is created, the condition at Line 6 becomes false and the thread proceeds with acquiring a new rank, essentially restarting the whole procedure.

One should finally point out that, in the MPMC variant, the ffq_deq() function is not lock-free anymore. Indeed, since we do not have the assumption of a single producer anymore, a producer that stops taking steps might prevent a consumer from progressing even if the queue is not empty. Furthermore, ffq_enq() is not wait-free as multiple producers can repeatedly hamper the progress of one another, but it is lock-free under the assumption that there is always some empty spot in the array.

4.4 Implementation and Optimizations

As our focus is on "raw" performance, we take special care of optimizations and fine-tuning. We discuss in this section the implementation details and the various optimizations that we

add to FFQ. Our evaluation shows that the throughput of a badly tuned vs. a well-tuned algorithm can differ by an order of magnitude.

4.4.1 Memory Mapping

To achieve the best performance, it is important to carefully place data structures in memory. Basic optimizations such as alignment of structures to word-sized addresses are typically performed transparently by the compiler. Coarser-grained mapping must, however, be done explicitly by the programmer.

In particular, one should avoid *false sharing*[5] between shared variables that are mapped to the same cache line. There are several ways to prevent this problem from happening.

We support the four combinations of two memory mapping approaches in our implementation. (1) With *dedicated cache lines*, queue cells are explicitly placed in different cache lines, hence avoiding consumers and producers to experience false sharing when they concurrently access the queue. (2) With *address randomization*, data is placed in the queue in such a way that neighboring cells in the shared array are mapped to distinct cache lines. The rationale is that false sharing is most problematic when consecutive elements of the queue share the same cache line, because consumers and producers access these elements sequentially.

Dedicated cache lines are efficient and easy to implement, but as a downside, they may significantly increase the size of shared data structures and hence reduce the effective capacity of the caches. In contrast, address randomization is slightly more complex to implement and requires several CPU instructions to compute, but it limits the impact of false sharing without memory overhead as each cache line can still hold several cells.

In our implementation, we can enforce the placement of cells in dedicated cache lines using compiler annotations. Note that this can also be achieved by inserting "padding" in data structures to increase their size to match cache line boundaries.

Address randomization can be implemented using various techniques ranging from simple bitwise manipulation of the addresses to more sophisticated transformations like minimal perfect hashing. In our implementation, we rotate the bits of the index by 4, effectively placing two consecutive cells 16 positions apart in memory, which will place them in distinct cache lines.

4.4.2 Thread Affinity

Besides optimizing the placement of data in memory, a complementary approach to maximizing performance consists of optimizing the thread placement on cores. This is typically achieved by defining the "affinity" of threads with specific cores. On modern CPUs, two hardware threads (HT)—or *hyperthreads* in Intel terminology—share a core. Hardware threads enable better utilization of a core and can increase core throughput in this way by up to 30 percent.[6] Operating systems like Linux permit us to set the affinity of a thread T by specifying a set of hardware threads on which T can run [7].

[5] Two threads accessing distinct variables sharing the same cache line will contend and invalidate each other's cache lines, hence generating unnecessary cache coherence traffic.

[6] `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`

[7] In the untrusted operated system setting, the operating system can ignore the affinity or even maliciously force a thread migration during the queue operation. However, in a correct queue implementation this can lead to the availability violation at worst, which is out of scope for TEE systems like SCONE.

Setting the affinity of threads to cores is a double-edged sword. On the one hand, we can ensure that producers and consumers can communicate via the same cache, without the need to migrate cache lines between cores or between CPU chips sitting in different sockets.[8] On the other hand, if producers and consumers need more CPU cycles than available on a given core or socket, we should not slow down the computation by overloading a single core or a whole CPU.

When maximizing the throughput of a system, we need to ensure that we optimize the usage of the cores and the caches. In our implementation, We support four different strategies for thread placement. We first force the producer and its consumers to all execute on a single hardware thread, i.e., they all compete for the same processing resources. Second, we place the producer on one hardware thread and its consumers on the second hardware thread on the same core, i.e., they can execute concurrently and share the same core cache. Third, we schedule a producer on one core and its consumers on a different core. Finally, we let the operating system schedule threads without any affinity being specified. We evaluate these different strategies in §4.5 to gain insights into the impact of thread placement on performance.

4.4.3 Queue Length

Another important tuning parameter is the queue length. By increasing the queue length, one can decouple the producer and its consumers. This will ensure that a temporary speed reduction of a producer will not slow down its consumers and vice versa. Moreover, by having a longer queue, we might see less false sharing of cache lines since the consumers and producers might naturally access different cache lines.

However, if the queue becomes too long, the cache hit rate might degrade since we reach the capacity of the cache. Cache lines will need to be written to memory and later be read again. This will not only increase the memory traffic but also limit the maximum throughput one can achieve. We provide the measurements that can help the user to pick the queue size in §4.5.3.

4.4.4 Implementation Notes

Our implementation is written in C with some assembly code for atomic operations and memory barriers. Memory alignment is implemented using compiler directives. Thread management is supported using the `pthread` library. The code is written for a native 64-bit word size, although it could be trivially adapted to 32-bit.

Our code currently supports Intel's x86 and IBM's POWER8 architectures. The main reason for benchmarking on these two architectures is that they both also support hardware transactional memory (HTM) extensions [154, 196], and we can therefore readily compare against state-of-the-art HTM-based concurrent queues in our evaluation.

[8] For simplicity, we indifferently use the terms "CPU", "CPU chip", and "socket" to denote the whole multi-core processor sitting in a socket.

4.5 Evaluation

We evaluate the performance of FFQ using a set of generic and SCONE-based micro-benchmarks. We also compare the performance of FFQ to alternative queue designs. To explain the impact of different optimizations applied in FFQ, we start with a sequence of generic micro-benchmarks.

4.5.1 Methodology

We evaluate our algorithms on 3 different servers:

- **Skylake.** An Intel Xeon E3-1270 v5 (4 cores at 3.6 GHz, 8 hardware threads, 8 MB cache) with 64 GB RAM, Ubuntu 14.04.4 LTS, gcc 6.1.0.
- **Haswell.** An Intel Xeon E5-2683 v3 (two 14-core CPUs at 2 GHz, 56 hardware threads, 35 MB cache, NUMA) with 112 GB RAM, Ubuntu 15.10, gcc 5.2.1.
- **P8.** An IBM POWER8 8284-22A (10 cores at 3.42 GHz, 80 hardware threads, 512 KB L2 and 8 MB L3 cache per core) with 32 GB RAM, Fedora 21, gcc 4.9.2.

We use a micro-benchmark that simulates the SPMC asynchronous system call interface. The benchmark spawns a predefined number of producer and consumer threads. The consumers are statically assigned to producers and there is always at least one consumer per producer.

Producer threads have a state that consists of an SPMC submission queue and an array with SPSC response queues for each of the consumers assigned to the producer. Producer threads insert a number of 64-bit integers into the submission queue and loop through the response queues for dequeuing values. Consumers repeatedly retrieve a value from the submission queue and enqueue a 64-bit integer into the associated response queue, using respectively the SPMC and SPSC FFQ algorithms.

Note that, by default, we run the micro-benchmarks on the Skylake server, whereas comparative tests are executed on all servers. All three architectures support hardware transactional memory (HTM) extensions. The benchmarks are written in C and compiled using gcc [126] with optimizations enabled (`-O3`). The reported results represent the average of 10 runs.

4.5.2 False Sharing

We evaluate the impact of false sharing on the throughput of FFQ using our micro-benchmark. In particular, we evaluate the effectiveness of dedicated cache lines vs. address randomization. To do so, we measure four different configurations the throughput of FFQ:

- **Not aligned.** The *cell* data structures are not cache aligned. Each cell requires 24 bytes in the cache.
- **Aligned.** The *cell* data structures are cache aligned and each requires 64 bytes in the cache.
- **Randomized.** The *cell* data structures are not cache aligned and each requires 24 bytes in the cache. The ordering of the cells is pseudo-randomized (as explained in §4.4).

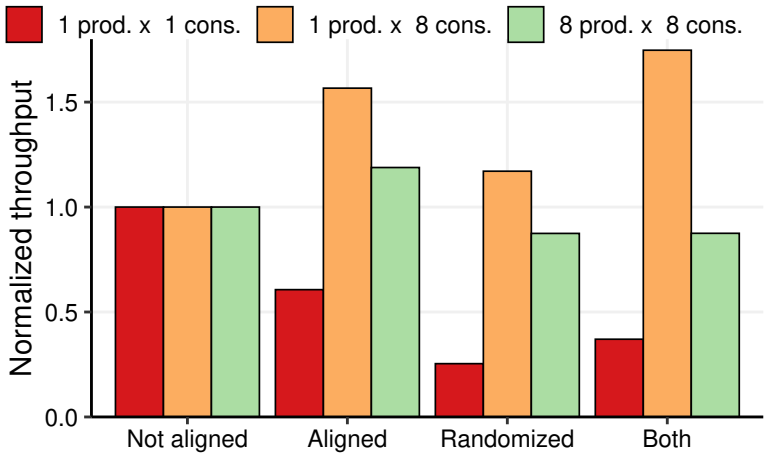

Figure 4.3: Impact of alignment and randomization on throughput with the MPMC variant of FFQ for a single producer and consumer, one producer with 8 consumers, and 8 producers with 8 consumers per producer. Throughput is normalized to the non-aligned variant.

- **Both.** The *cell* data structures are cache aligned and randomized. Each cell requires 64 bytes in the cache.

All experiments were conducted with the MPMC variant of FFQ. In the case of 8 producers, we use 8 distinct queues with 8 consumers each (i.e., 64 consumers).

Our first measurement shows that, for a single producer and a single consumer, neither alignment nor randomization improves throughput (see Figure 4.3). This can be attributed to several factors: we need less space in the cache for the cells without alignment and, hence, have a better cache hit ratio; alignment would only help if the consumer is always faster than the producer and, in this way, competing for the same cache line; and randomization adds some overhead for address computation upon every access to a cell.

When we increase the number of consumers per producer, a producer and a consumer will more likely compete for the same cache lines. Moreover, the consumers will also compete for the same cache lines. Our measurement shows that, when we have multiple consumers per producer or multiple producers, alignment of the cell data structure improves throughput. Randomization helps in the case of a single producer with 8 consumers, but when increasing the number of producers it becomes counter-productive (likely related to eviction patterns in the 4-way associative L2 cache). One can observe that the combination of alignment and randomization provides the best throughput in the case of one producer with 8 consumers.

4.5.3 Queue Size

We have seen that a reduction of the data structure size can—in the case of a single-producer single-consumer setup—improve the throughput. Hence, we next investigate the impact of the queue size on the throughput of the queue. By increasing the queue size, we can decouple the producers and the consumers since producers can continue to produce items while the consumers might be temporarily delayed. However, if the queue size becomes too large to fit in cache, the throughput of the queue may also decrease.

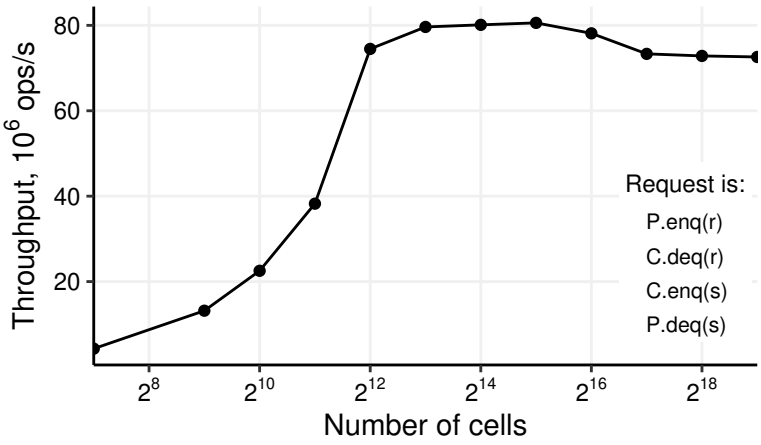

Figure 4.4: Influence of the queue size on FFQ throughput (Skylake). In a single-producer/single-consumer configuration, when reaching 64k entries, the throughput starts to decrease.

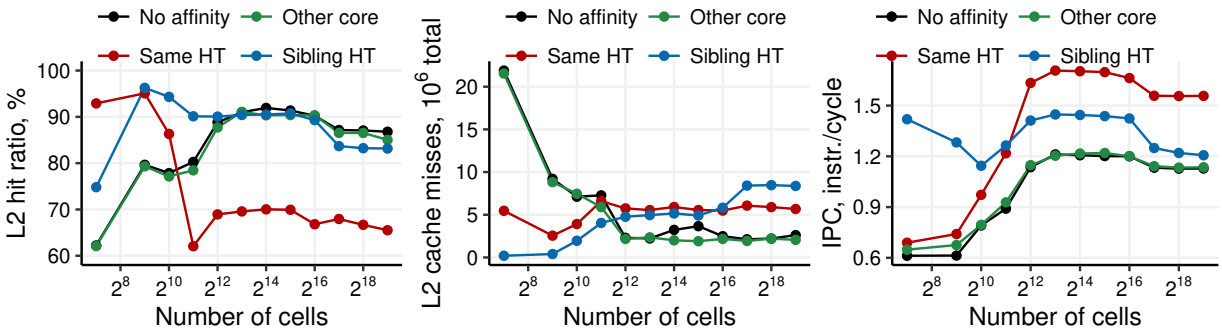

Figure 4.5: L2 cache hit ratio, L2 misses, and IPC (instructions per cycle) for a single-producer/single-consumer configuration (Skylake).

We measure the impact of the queue size for a single producer and a single consumer, each running on a separate core (see Figure 4.4). One can see that, in this measurement, we reach the maximum throughput for a queue with 64k entries.

In a later measurement (see Figure 4.7), we see that if the producer and consumer share the same core, a much smaller queue size can actually yield better throughput. The optimal queue size depends on the mapping of the threads to individual cores, i.e., if we use a L1/L2 cache attached to the core or a more remote (L3) cache. Hence, we investigate the impact of using a local L2 cache vs. a remote L3 cache next.

4.5.4 Cache Locality and Thread Affinity

Modern CPUs have a L1 and L2 cache that is attached to the core and a L3 cache shared amongst the cores. The L3 cache is partitioned such that some parts are closer to some cores than others. By pinning a producer thread and the consumer threads to the same or different cores, we can enforce producers and consumers to communicate either through a local L1/L2 cache or a remote L3 cache. However, caching is not the only factor that impacts

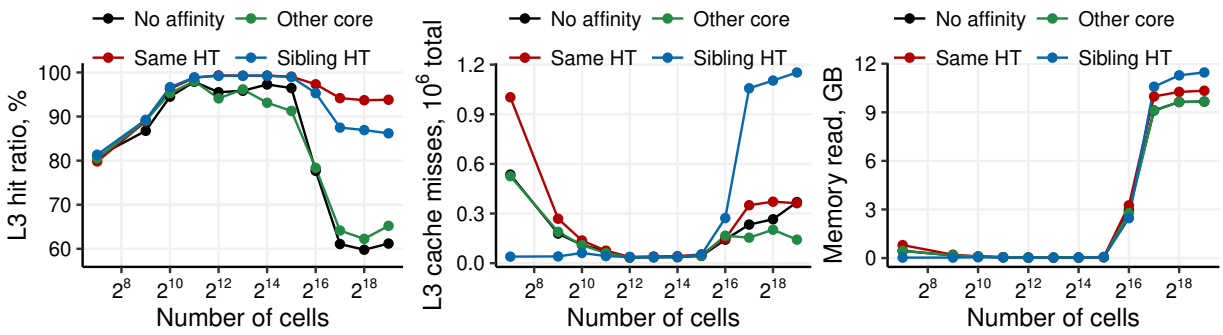

Figure 4.6: L3 cache hit ratio, L3 cache misses, and memory access bandwidth for a single-producer/single-consumer configuration (Skylake).

performance. There are other mechanisms at work like the likelihood that a single core goes into turbo mode, or the degree of the instruction level parallelism.

To understand the factors that affect performance, we perform a simple micro-benchmark: to simplify the understanding of the results, we measure a configuration with a single producer and consumer thread. During the benchmark execution, we record different performance counters that keep track of performance metrics like cache hit ratio, memory access bandwidth, and core frequency. All cells were cache aligned.

In the context of this micro-benchmark, we evaluate different queue lengths and four different affinity policies.

- **Sibling HT.** We place the producer and the consumer on the same core but different hardware threads.
- **Same HT.** We enforce the producer and the consumer to share the same hardware thread.
- **Other core.** We place the producer and the consumer on two different cores on the same socket.
- **No affinity.** We run the benchmark without setting the affinity, i.e., we let the Linux scheduler determine on which hardware thread to place a thread.

The measurements (see Figures 4.5 and 4.6) show that *other core* and *no affinity* have almost the same behaviour, which tends to indicate that Linux schedules the producers and the consumer on different cores. Hence, we focus only on the *no affinity* measurements. One can see that with increasing queue size, the hit ratios of both L2 and L3 are increasing. However, if the queue size does not fit in the L3 cache anymore, the L3 hit ratio drops and cache misses increase. Furthermore, the memory access bandwidth becomes higher, which reduces the IPC (instructions per cycle).

Executing the producer and consumer on the same core but different hardware threads (*sibling HT*) has better L2 and L3 cache hit ratios than the other alternatives, except for very large queue sizes. It also exhibits good IPC even for small queue sizes. However, for large queue sizes, it has a larger number of L3 cache misses and requires a larger number of memory accesses, apparently caused by the larger L3 cache requirements due to cache contention between the producer and the consumer.

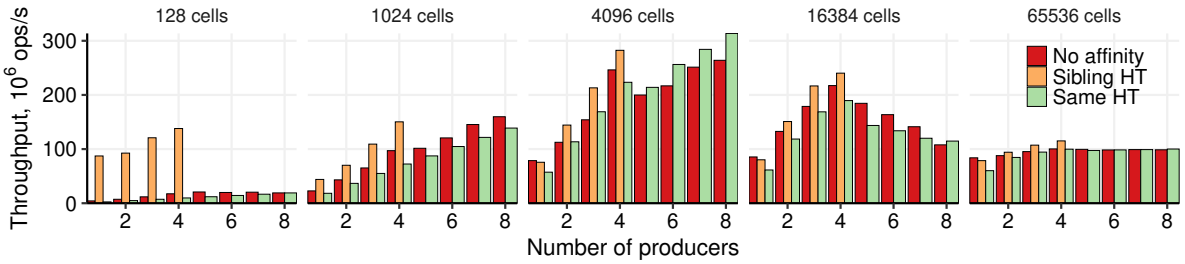

Figure 4.7: Throughput for different the queue sizes and affinity settings (Skylake). When executing on two hardware threads on the same core, the performance decreases with increasing queue size. When running on different cores, the queue benefits from large queue sizes (that decouple producer and consumer) and the additional cycles of the cores.

For reasonably small queue lengths, *same HT* has the highest L3 cache hit ratio and, hence, also the highest IPC. This might indicate that it performs best under resource constraints. We investigate this next.

4.5.5 Maximizing Throughput

In SCONE, the FIFO queue is a bottleneck that limits the throughput of the application. Hence, we want to ensure that we can maximize the throughput of the FIFO queue. To maximize the throughput, we need to ensure that we are as resource-efficient as possible. To that end, we can use IPC as it represents a good indicator of resource efficiency.

The measurements of the last subsection have shown that, for very small queue sizes, *sibling HT* exhibits the best IPC. For larger queue sizes, *same HT* shows the best IPC. However, since there is some back-off involved, the best IPC does not always mean also the best throughput.

We measure the throughput for various configurations in which we increase the number of producers. For each producer, we start one consumer. As our Skylake CPU has four cores and a total of 8 hardware threads, for *sibling HT* we limit the number of producers to 4. For the other two policies, we also oversubscribe the cores, i.e., schedule up to two threads per hardware thread.

The micro-benchmark results (see Figure 4.7) show that *sibling HT* performs best both for small and large queue sizes. However, for medium queue sizes that maximize L2 and L3 cache hit ratio, *same HT* actually performs better (with respect to the number of cores used). *Sibling HT* benefits from the consumer and producer accessing similar memory regions and, hence, from a better L2 cache hit ratio. *Same HT* benefits from a better IPC and, hence, provides the best throughput as long as queue sizes are sufficiently large to keep both producer and consumer busy without waiting.

4.5.6 Application Benchmark

We have integrated FFQ into SCONE and measure its performance using a benchmark application. The benchmark spawns threads that execute `getppid(2)` in a loop. This system call was chosen because it executes fast and involves no costly system call argument copying,

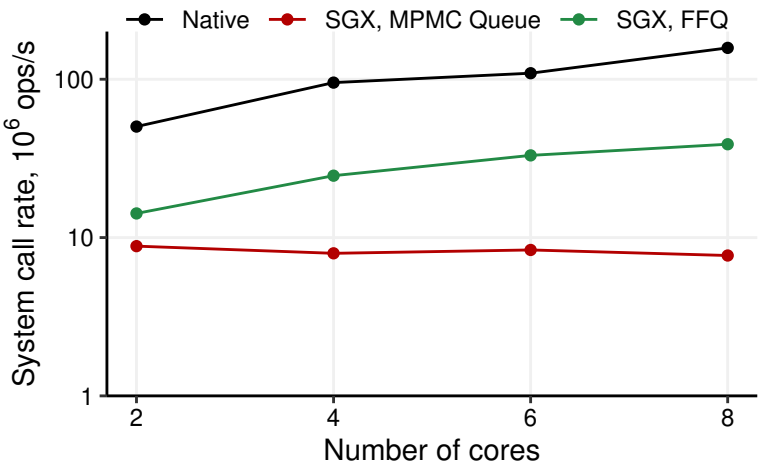

Figure 4.8: Throughput of the SCONE system call benchmark application with different number of available cores on the Skylake server.

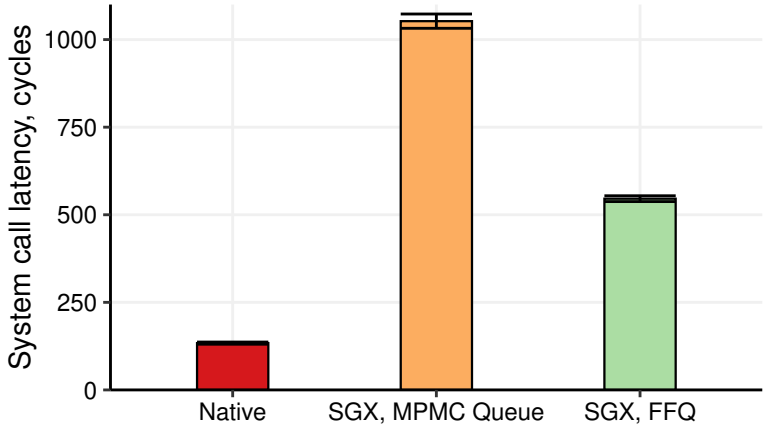

Figure 4.9: Latency of the `getppid` system call with different queues on the Skylake server.

making system call queues a bottleneck. The application records throughput (system calls per second) and average latency (CPU cycles). The benchmark application is built in three variants: native version, SGX enclave with the originally used MPMC queue[9], and SGX enclave with FFQ (SPMC or MPMC variants depending on the number of producers and cores). Running inside the SGX enclave causes additional overheads when the enclave memory is removed from the CPU cache due to memory encryption operations.

Figure 4.8 shows the scalability and performance gains from FFQ. The amount of application threads spawned is proportional to the amount of available cores[10] and is fine-tuned for each variant of the binary to provide the best throughput. In contrast to the MPMC variant, the binary with FFQ achieves a 5 times higher throughput and scales linearly.

Figure 4.9 shows the end-to-end system call latency in the benchmark application. The

[9] `http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue`

[10] We limit the number of application threads that produce system calls, but these threads share the cores with other runtime and system threads.

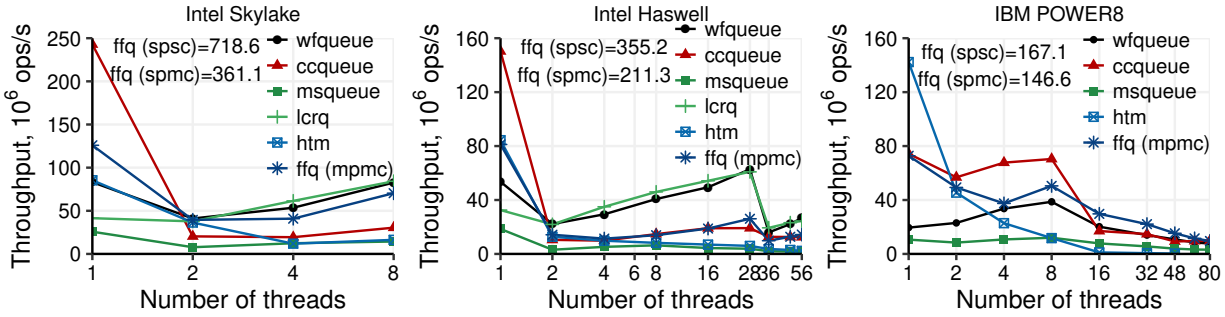

Figure 4.10: Throughput of the benchmark from [298] with our three servers: Skylake (left), Haswell (center), POWER8 (right). The throughput values indicated for SPSC and SPMC are for single-threaded runs.

measurement was done with a single application thread to prevent thread multiplexing in the SGX variants. The latency of a native system call provides a baseline for comparison. The system call latency of FFQ is almost twice as low compared to the MPMC variant. The latency is higher than the baseline because it involves a ping/pong of request and answer between two threads.

4.5.7 Comparative Study

In our last set of experiments, we compare FFQ against other state-of-the-art concurrent queues. To this end, we add our algorithm to the benchmark framework developed and kindly provided by the authors of [298]. We compare FFQ to Yang and Mellor-Crummey's *wfqueue* (fast WF-10 version) [298], Morrison and Afek's *lcrq* [225], Fatourou and Kallimanis's *ccqueue* [119], Michael and Scott's *msqueue* [220], and a simple concurrent queue algorithm that uses hardware transactional memory (HTM) extensions of Intel and IBM CPUs. This last algorithm is based on a bounded circular buffer and simply executes the enqueue and dequeue operations inside hardware transactions. In the benchmark, all threads repeatedly execute pairs of enqueue and dequeue operations on a single queue, for a total of 10^7 pairs partitioned evenly among all threads. We hence use the MPMC variant of FFQ to support concurrent accesses of both producers and consumers. Between two operations, the benchmark adds an arbitrary delay (between 50 and 150 ns) to avoid scenarios where a cache line is held by one thread for a long time.

The results are shown in Figure 4.10 (to be compared with Figure 2 of [298]). Note that the P8 results do not include *lcrq* as it is not supported by the benchmark framework for the POWER architecture. We also indicate in the graphs the performance of the SPSC and SPMC variants of FFQ when running with a single thread. The SPSC variant of FFQ removes the need for an atomic increment operation. In particular, given that our main focus is to maximize the performance of the SPMC implementation, this allows us to estimate the overhead of supporting multiple consumers.

We can first observe that, on each system, FFQ is consistently among the most efficient implementations for each thread count even though we are using the MPMC variant. In sequential runs, *ccqueue* usually performs best (except on P8) because it reuses the same node for every enqueue/dequeue pair and does not experience cache misses without contending thread, but performance drops quickly with more threads. The *wfqueue* algorithm performs

well with an increasing number of threads, mainly on Intel processors, thanks notably to its efficient fast path that does not use compare-and-set operations. In most cases, *lcrq* is slightly slower than *wfqueue*, which can be explained by the higher number of memory fences. Note that *lcrq* and FFQ^m use a double-word compare-and-set, which is only available on a few high-end CPUs. The *msqueue* algorithm performs worst because it manipulates the queue's head and tail pointers with a compare-and-set operation inside a loop that can repeat many times under heavy contention. Finally, one can observe that the HTM-based implementation does not compete with the fastest queues, especially when increasing the number of threads, because transactional operations and retries are costly. This is particularly clear on the P8 architecture, where performance becomes extremely low as we increase concurrency. It is noteworthy, however, that HTM performs best with a single thread on P8—on par with the SPMC variant of FFQ—which tends to indicate that the overhead of transactions is low on that architecture when there are no conflicts.

In terms of processor architectures, FFQ performs comparatively better on P8 than on Skylake and Haswell, even though the raw throughput values are slightly lower. Furthermore, performance does not drop too sharply on P8 after 8 threads, i.e., when exceeding the number of cores, whereas it degrades dramatically for *ccqueue*.

Finally, one can appreciate the benefits of the SPMC variant over MPMC, with a gain of more than 50% on all architectures. This is particularly important as FFQ was designed and optimized for applications with a single producer and multiple consumers. The SPSC variant of FFQ, which is less interesting from a practical perspective, runs two to three times faster than MPMC.

4.6 Conclusion

In this chapter, we provide the extended design, implementation, and evaluation of the concurrent queue that was developed to implement the performance-critical component of SCONE—its asynchronous system call interface. Unlike the original bounded MPMC queue, FFQ in its SPMC variant is optimized for the case of independent producers. With FFQ, SCONE enclave threads can submit system calls without synchronization, and needs less atomic instructions.

The comparison with other state-of-the-art queues shows that our SPMC variant has a 50% higher throughput than other state-of-the-art FIFO queues for a single producer configuration. For multiple independent producers, our evaluation also shows that FFQ has excellent scalability—reaching more than 1.2 billion FIFO operations per second on a 4 core Skylake CPU. Our MPMC variant of FFQ is competitive with the best state-of-the-art FIFO queues. The higher performance of SPMC over MPMC can be attributed to the SPMC variant needing fewer atomic operations. However, we also observe that the correct tuning is an essential factor for performance. We show in the context of the SPMC variant that a good data structure alignment, queue length, and thread affinity can result in an order of magnitude higher throughput. The combination of FFQ and these optimizations allows SCONE to run POSIX applications with low overheads.

5 Securing Middleboxes using Shielded Execution

In the previous chapter, we have shown how we improved the performance of SCONE asynchronous system call interface using an improved concurrent queue implementation. However, for some applications, even the native performance of system calls is insufficient, often due to the suboptimal interface presented by the operating system. In this case, utilizing the full capabilities of hardware becomes impossible, and alternatives have to be developed [254].

An example of software that faces these challenges are applications are software that processes network packets at line speed, for example IDS systems, network traffic optimizers, firewalls, programmable switches, and so on — so-called middleboxes. Typically they operate on-premises; however, recent research has shown users can significantly reduce the maintenance cost of middleboxes by outsourcing them to the cloud. In this context, it becomes important to extend the protection offered by Intel SGX to these middleboxes, and provide middlebox developers a toolset for the construction of secure middleboxes, as they may be processing confidential user data, and thus may become targets for attacks by the hackers or privileged insiders.

Therefore, in this chapter, we solve the problem of securing a network middlebox inside Intel SGX enclaves, while avoiding the overheads of both native system calls and the SCONE asynchronous system calls. We achieve our goal by building on the kernel bypass network I/O library (DPDK), a state-of-the-art modular router (Click), and the SCONE framework introduced in the previous chapters.

5.1 Introduction

Modern enterprises ubiquitously deploy network appliances or “middleboxes" to manage the networking infrastructure. These middleboxes are extensively used to maintain a wide range of workflows for improving efficiency (e.g., WAN optimizers), performance (e.g., caching, proxies), reliability (e.g., load balancers, monitoring), and security (e.g., firewalls, IDS). Due to their widespread usage, they incur significant deployment, maintenance, and management costs [267].

To overcome these limitations, many enterprises are contemplating outsourcing middle-

boxes to the cloud [267, 238]. Cloud computing offers economies of scale for computational resources with the ease of management, elasticity, and fault tolerance. Realizing the vision of middleboxes as a service in the cloud is strengthened by the advancements in network function virtualization (NFV) [211]. NFV offers a flexible and modular architecture that can be easily deployed on commodity hardware. Thus, NFV is a perfect candidate to reap the outsourcing benefits of the cloud infrastructure.

However, middleboxes that process confidential data cannot be securely deployed in untrusted cloud environments. In cloud environments, an accidental or, in some cases, intentional action from a cloud administrator could compromise the confidentiality and integrity of the middlebox execution. These threats of potential violations to the integrity and confidentiality of customer data are often cited as a key barrier to the adoption of cloud services [259]. Furthermore, cloud providers are increasingly offering edge computing resources in collaboration with third-party ISPs and CDN operators to meet stringent low-latency performance requirements (SLAs) of modern online applications [128]. Since the underlying infrastructure is operated by multiple third-party providers, such a hybrid cloud-edge computing infrastructure further exacerbates the secure deployment of middleboxes.

To securely outsource middleboxes in the cloud, state-of-the-art systems advocate network processing over encrypted traffic [268, 194]. However, these systems support only restrictive types of functionality and incur prohibitively severe performance overheads since they require complex computations over encrypted network traffic.

These limitations motivated our work—we strive to answer the following question: *How to securely outsource middleboxes on the untrusted third-party platform without sacrificing performance while supporting a wide range of enterprise NFs?*

To answer this question, we present ShieldBox—a secure middlebox framework for deploying high-performance network functions (NFs) on untrusted commodity servers. The architecture of ShieldBox is based on four design principles: (1) *Security* — we aim to provide strong confidentiality and integrity guarantees against a powerful adversary, (2) *Performance* — we strive to achieve near-native throughput and latency, (3) *Generality* — we aim to support a wide range of network functions (same as plain-text processing) with the ease of programmability, and (4) *Transparency* — we aim to provide a transparent, portable, and verifiable environment for deploying middleboxes, without major changes to the systems source code and deployment procedure.

To achieve these design goals, ShieldBox leverages the components presented earlier in this thesis, that is hardware-assisted secure enclaves based on Intel SGX [103] for providing strong security properties, and SCONE (§3)—a shielded execution framework to securely process network packets on commodity untrusted infrastructure. However, the architectural limitations of Intel SGX present a significant challenge for middleboxes requiring high-performance network I/O.

To achieve high performance despite the inherent limitations of the SGX architecture, we have designed a high-performance I/O library for shielded execution using Intel DPDK [22] to efficiently process packets in the userspace secure enclave memory.

For the developers, ShieldBox provides a flexible and modular framework to build a rich set of NFs by adapting the Click [182] architecture. In this way, ShieldBox supports a wide range of NFs with the ease of programmability using Click's out-of-the-box elements and C++ extensions. Finally, ShieldBox builds on the Docker container technology with a remote attestation and configuration service (an early version of Palaemon [138]), which provides network operators a portable and cryptographically verifiable deployment mechanism.

Furthermore, we have designed several important end-to-end features required for secure

middleboxes:

- New Click elements for secure packet processing.
- Efficient shared memory packet transfer mechanism in the multiple SGX enclaves setup for NFVs chaining [173].
- Secure state persistence layer for fault-tolerance and stateful migration of middleboxes [266].
- On-NIC PTP clock as a time source for the SGX enclaves.
- Memory safety mechanism to defend against DPDK-specific Iago attacks [100].

We implement the aforementioned security features as well as several SGX-specific performance optimizations in ShieldBox. Lastly, we evaluate the system using a series of microbenchmarks, and two case-studies: a multiport IP Router, and IDS. Our evaluation shows that ShieldBox achieves near-native throughput and latency.

5.2 Background and Related Work

Network Function Virtualization (NFV) is an approach to the networking infrastructure that involves a consolidation of independent network appliances on the standard COTS servers by relying on the virtualization technologies [140, 144]. As explained in the original NFV whitepaper [140], the reasons for the adoptions of the NFV approach are:

- Difficulties with deploying a high number of custom hardware components to the data centers, related to the power and space consumption.
- High capital investments necessary to obtain and to integrate the hardware into an existing network.
- Rapid obsolescence of the hardware appliances, their low scalability and elasticity.

By relying on the standard virtualization and isolation technologies available in the modern operating systems, such as software fault isolation technologies, containers, and virtual machines, it is possible to improve multiple aspects of the network [144]:

- Reduce equipment cost and power consumption, exploit the economy of scale in the IT industry.
- Improve development speed of network functionality.
- Enable multi-versioning and multi-tenancy in the network.
- Improve elasticity of network appliances.

The use-cases for NFV include routers, mobile network components, tunnels and VPNs, Deep Packet Inspection (DPI) appliances, monitoring systems, billing components, traffic optimizers, security gateways, and much more [140]. As a result, each of these components, previously implemented as a hardware *middlebox*, is transformed into a piece of software that can be deployed in the network using standard orchestration technology, such as OpenStack, Docker Swarm, or Kubernetes.

Software-Defined Networking (SDN) is related but distinct concept to NFV. The idea behind SDN is a separation of *forwarding plane*, responsible for high-throughput low-latency packet forwarding, and *control plane*, responsible for making decisions about the forwarding destination of each packet, in separate entities, either software or hardware [186].

The motivation for this change is a significant difficulty of configuring and reconfiguring the network in response to overloads, component failures, or changing requirements. Without SDN approach, this reconfiguration would require individual, low-level reconfiguration of each networking device (router or switch), which is an error-prone process, while the lack of standardized APIs was preventing the application of automation to this problem.

The SDN approach was developed to solve these problems. It logically centralizes the control plane (a configuration of the network) in several *SDN controllers*, while switches and routers essentially become programmable *forwarding elements*. This separation allows easy implementation and programmability and automation in the network.

To promote interoperability in the SDN-based networks, a common protocol is necessary for the communication between the SDN controllers and the forwarding elements. The industry standard for this is the OpenFlow protocol [218]. It allows a single control plane to manage multiple forwarding components.

In our work, we consider the problem of network management out-of-scope, thus focusing on the NFV technologies.

Kernel Bypass Technologies. There are several requirements to the networking middleboxes, the main of which are high performance and flexibility. These requirements are challenging to achieve even without Intel SGX, which exacerbates the implementation challenges even further.

Achieving high performance has become challenging because of the ever increasing capabilities of the NICs. While common POSIX-compatible operating systems and the networking software running on them can process data at the line rate of the 1G and 10G Ethernet NICs, recent upgrades to 25G, 50G, and recently to 100G Ethernet has shown that the available OS interfaces are inadequate for high-throughput low-latency operation at such speeds [241]. This change has forced the developers to consider alternative ways to implement middleboxes, while still retaining all features and compatibility with the dominant operating systems.

This improvement in hardware together with the design of the common general-purpose operating systems force a dilemma upon middlebox developers: they can either implement their application as a userspace program, or as a part of the OS kernel. Userspace implementation allows easy access to COTS libraries, simple update process, and guarantees of kernel API stability. On the other hand, the raw packet methods that, for example, Linux provided to userspace applications were insufficient for most low-level (L2/L3) middleboxes. Furthermore, even standard in-kernel functionality sometimes proved to be insufficiently performant due to its low scalability and general purpose design [209].

On the other hand, direct in-kernel implementation of the middlebox functionality allows high-speed packet access, but restricts the programming model, prevents the use of common libraries, and complicates the middlebox support, which has to be adapted to the APIs changes in the new kernel versions. Additionally, the functionality already available in the common OS kernels is not flexible enough to implement complex middleboxes, for example, BPF [214] can only be used to implement filters for packet capture. There are attempts to make in-kernel network programming more flexible and efficient using technologies like eBPF and AF_XDP [290, 84]. Unfortunately, while AF_XDP can be efficiently used to implement increasingly flexible middlebox functionality, it does not admit SGX-protected implementa-

tion, as Intel SGX can only be used with userspace code.

The standard Linux interfaces for raw packet acquisition is AF_PACKET, which can be combined with PACKET_MMAP to gain a zero-copy access to multiple packets at once. However, in this case the packets still pass through several kernel subsystems (for example, queueing disciplines), which slow down the packet processing.

As an alternative to this packet acquisition method, kernel bypass technologies were developed. They either implement the device driver for the NIC in the userspace, directly accessing the PCI MMIO ranges using the OS interfaces typically utilized by the VMMs, or use custom kernel modules to expose a uniform, ring buffer-based interface to different NICs.

The DPDK (Data Plane Development Kit) framework is an example of the former approach: it provides userspace drivers for popular high-performance NICs, and a set of well-optimized libraries for their efficient utilization [22]. To provide a certain level of generality to the DPDK-based software, it provides so-called Environment Adaptation Layer (EAL), a library that manages the acquisition of NICs from the operating system, creation of worker threads and their pinning to cores, either automatically or based on the command-line arguments. DPDK achieves high performance by avoiding synchronization between cores as much as possible, using lock-free and cache-efficient data structures, as well as several software-based approaches: preallocation of memory for the packet buffers, polling for new packets, and pipelined processing. A combination of these features allows DPDK to achieve line rate processing with modern high-speed NICs.

Netmap [254, 255, 207], implemented in FreeBSD and available as a third-party patch on Linux, implements a latter approach, requiring Netmap-specific changes in each of the device drivers to be available through the Netmap interface without emulation overheads. Netmap exposes a ring buffer structure per NIC queue that the userspace applications can use for zero-copy, batched packet access. Control of the packet transmission (interface configuration, polling, etc.) is still performed through the common POSIX system calls. This offers the developers and administrators easy-to-understand control mechanisms: configuration with `ifconfig`, synchronization via `poll`, at the cost of slightly lower performance due to system call overhead (e.g., DPDK uses busy polling). In addition to this, Netmap allows performing zero-copy forwarding between different Netmap ports.

It should be noted that modern non-POSIX operating systems commonly implement similar kernel bypass interfaces, for example Arrakis [241] and IX [85], to achieve high performance and low latency.

Most kernel bypass libraries implement only packet acquisition and L2/L3 processing functionality. In case processing on higher levels of the network stack is required, for example interaction with TCP streams, the middlebox developer can add the required functionality via third-party libraries. For example, rump kernel is a special build mode of NetBSD kernel that allows the reuse of unmodified kernel components as part of userspace application [171]. In particular, it allows the reuse of the kernel TCP/IP stack. An alternative to NetBSD rump components are library network stacks developed for embedded or high-performance use, such as LwIP [115, 248] or TLDK [46].

Click Modular Router. Click modular router [182] is a framework for flexible construction of internet routers and firewalls. The developers customize the behavior of the router by configuring Click: it provides a declarative and safe domain specific language for describing a packet processing graph. The graph consists of so-called *elements*, which are small, atomic, self-contained units of functionality (following the Unix design philosophy). In case an element that provides the necessary functionality is missing, the developer can extend Click by implementing the element in C++, for which a high-level interface is implemented.

```
PollDevice(eth1, true)
  -> SetTimestamp
  -> Queue(8)
  -> DelayShaper(10)
  -> BandwidthShaper(131072B/s) // 1Mb
  -> ToDevice(eth2);

PollDevice(eth2, true)
  -> SetTimestamp
  -> Queue(8)
  -> DelayShaper(10)
  -> BandwidthShaper(131072B/s) // 1Mb
  -> ToDevice(eth1);
```

Figure 5.1: An example of Click router configuration. Each of named entities (`PollDevice`, `SetTimestamp`) are an instance of a corresponding Click *element*, while the arrows connect them into a packet processing graph. This Click configuration builds a simple forwarder between two interfaces with a delay and bandwidth shaping [181].

Click-based middleboxes. Click's extendable architecture and easy-to-understand graph language has made it a to-go tool for implementing production and research middleboxes [211, 70, 89, 201, 170, 170]. Our work also builds on the Click architecture, but unlike the previous work, ShieldBox focuses on securing the Click architecture on the untrusted hardware.

As Click lacks the elements for supporting TCP and higher-level protocols, most Click-based network appliances operate at the L2-L3 layer, with the notable exception of CliMB [195]. CliMB implements TCP support as a set of more than 40 different elements. The corresponding TCP streams are available to user applications via POSIX-like API; it can be used to communicate with user applications in a blocking and non-blocking manner.

To support flow-based abstractions, many state-of-the-art middleboxes [165, 264, 67, 208, 66] support comprehensive applications and use-cases. Since both Click and DPDK are geared toward L2-L3 network processing, our current architecture does not support L4-L7 NFs. As part of the future work, we plan to integrate a high-performance user-level networking stack [167] in the SCONE framework to support the development of secure higher layer network appliances.

While initially Click has provided a custom kernel module for accelerating packet acquisition and processing, it has also gained support for Netmap and DPDK frameworks for fully userspace operation, contributed by the authors of FastClick [78]. However, virtualization provides another interface that can be used for high-performance networking: *ClickOS* [211] is "a high-performance, virtualized software middlebox platform", that utilizes a modified Xen hypervisor to deploy multiple Click-based VMs on commodity hardware. To improve the performance of Xen networking to the line packet rates, it has several optimizations implemented: polling for packets, use of a high-performance software switch [255], and direct mapping of per-port ring buffers into the VM address space (which implements kernel bypass and zero-copy packet transfers). Just like ClickOS, ShieldBox achieves high performance by directly exposing NIC buffers to the userspace application.

Secure middleboxes. APLOMB [267] is one the first systems to showcase that it is a viable alternative, performance- and cost-wise, to outsource middleboxes from the enterprise en-

vironment to the cloud. However, APLOMB did not consider the security implications of outsourcing in the cloud. To overcome the limitation of APLOMB, the follow-up systems, namely Embark [194] and BlindBox [268], advocate network data processing over the encrypted traffic. In particular, BlindBox [268] proposes an encryption scheme based on garbled circuits to support string matching operations over encrypted traffic. However, Blindbox supports only a restricted type of functionalities, supporting only the basic network functions for DPI. To overcome this limitation, Embark [194] extends BlindBox to support a wider range of network function. However, Embark suffers from the prohibitively low performance as it involves complex cryptographic computations over the encrypted network traffic. In contrast, ShieldBox supports a wide range of NFs (same as plain-text) and achieves a near-native throughput and latency.

The several workshop papers [105, 176, 145] have elaborated the challenges and potential usages of SGX in the network applications. In the domain of network-intensive applications, SGX-Tor [175] is one of the first systems to use SGX to enhance the security and privacy of Tor. In a similar vein, CBR [242] leverages SGX to support privacy-preserving routing. Both these systems rely on Intel SGX SDK and kernel-provided sockets for their operation. Similarly, the ShieldBox project builds the first comprehensive system using Intel SGX to secure the middleboxes, but it is built using Click abstractions and with a kernel bypass technology.

It is possible to improve the middlebox throughput not by relying on the kernel bypass, but by outsourcing the network packet processing functionality to the clients. This approach is taken by Endbox [135], a framework that aims to reduce the physical centralization of the network processing, and eliminate a single point of failure in the network. To ensure that the client cannot violate the network processing policy, the packet processing happens under the protection of Intel SGX, and the packets are passed through a VPN which is accessible only via the Endbox enclave. Endbox is built using Click and OpenVPN. The measurements performed by the authors show that it is possible to improve the throughput 2.6-3.8x by using EndBox instead of a centralized middlebox. Unlike EndBox, ShieldBox is designed to operate as a centralized service in the network.

The two other concurrent research projects also investigated secure deployment of NFs: First, SafeBricks [244] is a system for outsourcing NFs to the untrusted cloud. It has high isolation and safety properties with minimal overhead stemming from the Rust type system, and implements least privilege principle for NFs. Similar safety properties are provided to Click by its DSL. Secondly, mbTLS [227] presents a modification to TLS v1.2 protocol that allows seamless and secure integration of middleboxes into connections between two peers. It leverages Intel SGX to authenticate the middlebox and has a high level of backward compatibility with legacy peers.

Timers for SGX-based middleboxes. As we will see in this chapter, timer access is a crucially important piece of the operating system functionality, which is used for scheduling execution of network function elements, timestamping packets and log entries, and by default is available only via a system call, like `clock_gettime` or `gettimeofday`; these system calls are well-optimized on Linux, up to the point of not requiring a context switch [55]. However, for Intel SGX enclaves, they are available only with significant performance loss due to cache flushing (in case of enclave exit) or with high latency in the case of SCONE asynchronous system calls. We present our own solution to this problem in §5.5.5, and the alternative solutions used by state-of-the-art systems—below.

SEC-IDS [188] is one of the first systems that has paid attention to this problem in the context of porting Snort, a popular enterprise intrusion detection system (IDS), into an Intel

SGX enclave. As Snort timestamps each packet for the purpose of logging, getting the time from the OS using `clock_gettime` system call at least twice per packet. Thus, for the initial implementation of this function using Graphene-SGX OCALL (outside call), the authors report 10× overhead over the native version. To eliminate this overhead, they rely on an efficient, high-resolution software timer presented by Schwarz et al. [263]. This optimization has reduced the timer access latency to native levels, at the cost of dedicating a core to the timer thread.

An alternative solution to the problem of low-latency in-enclave timer is provided in SafeBricks and Lightbox [244, 114]. Both of these systems fundamentally rely on the NIC timestamping feature: as NIC puts a timestamp into the metadata of each packet, the enclave uses it to update the internal clock. In this case, the resolution of the timer is directly proportional to the rate of the incoming packets. We note that the solution in Lightbox is more robust, as it relies on the periodic heartbeat message exchange with a trusted host to synchronize the time, which also provides a lower bound on the timer resolution, which is missing in Safebricks.

5.3 Middlebox Challenges for Intel SGX

Intel SGX restrictions and SCONE architecture present several challenges for the construction of shielded middlebox instances.

First, as we have mentioned in §5.2, with Intel SGX, the middlebox code cannot be put into the kernel mode, and thus only userspace solutions can be used. This proves to be particularly challenging on Linux, which lacks an equivalent of the FreeBSD Netmap interface, thus making an application of kernel bypass a favorable option. While Netmap is available on Linux as a third-party patch, using it would restrict our ability to deploy ShieldBox in the cloud, where third-party Linux patches are typically not available.

Additionally, even with the kernel bypass solution in place, SCONE asynchronous system calls should not become a source of overhead, as system calls may still get invoked for service operations. In particular, access to the time source has proven to be a common operation in the middleboxes, as it is necessary to perform operations like rate limiting and periodically sending packets. This problem calls for a low-overhead time source.

The interaction of the userspace threading in SCONE with the DPDK expectation of having fully dedicated cores also presents a challenge for ShieldBox. For example, each system call in SCONE is a userspace thread preemption point, which in the case of DPDK adversely affects cache locality due to the necessity to enter the userspace scheduler, which may cause further overhead by scheduling a different thread on the same core.

Finally, for the kernel bypass, memory management provides a challenge: kernel bypass framework typically delivers packets via DMA to the unprotected physical memory. However, in case of Intel SGX, direct memory access to the enclave memory is impossible. Thus, ShieldBox has to ensure that the memory used for packet delivery is outside of the enclave. While handling this issue, it is necessary to make sure that the packet delivery mechanism cannot be used to perform Iago attacks.

In the next section, we will present the design decisions in ShieldBox that allowed it to overcome these challenges and achieve near-native throughput and latency.

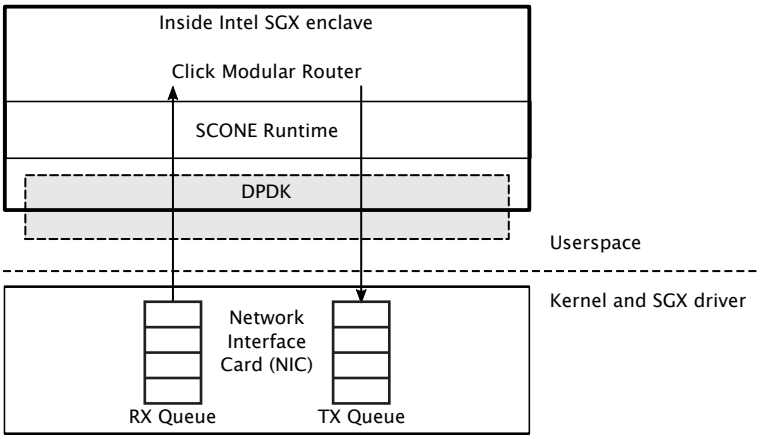

Figure 5.2: ShieldBox basic design.

5.4 Overview

Basic design. At a high-level, the core of our system consists of a simple integration of a DPDK-enabled Click [182] that is running inside the SGX enclave using SCONE (§3). Figure 5.2 shows the high-level architecture of ShieldBox.

While designing ShieldBox, we need to take into account the architectural limitations of Intel SGX. As described in §2.2, an enclave context switch (or exiting the enclave synchronously for issuing system calls) is quite expensive in the SGX architecture. The SCONE framework overcomes this limitation using an asynchronous system call mechanism [277]. While the asynchronous `syscall` mechanism is good enough for common Web services like HTTP servers or key-value stores, it is not sufficient to sustain the line rate as required by modern middleboxes. Especially, numerous modern middleboxes require a *fast path* bypassing kernel network stack to achieve the line rate [84]. Therefore, we design a high-performance I/O library for shielded execution based on the userspace DPDK library [22] as a better fit for the SGX enclaves. As we have noticed before, using Netmap on Linux requires patching the host kernel, which limits the possibility to easily install Netmap in the cloud.

Furthermore, we need to ensure that the memory footprint of ShieldBox code and data is minimal, due to several reasons: As described in §2.2, enclaves that use more than 94MB (198MB on the newer generation of Intel CPUs) of physical memory suffer grave performance penalties due to EPC paging (2× to 2000×). In fact, to process data packets at the line rate, an even stricter resource limit must be obeyed—the working set of the application must fit into the L3 cache. Therefore, our design diligently ensures that we incur minimum cache misses, and avoids EPC paging.

Besides performance reasons, minimizing the code size inside the enclave allows reducing the attack surface as it leads to a smaller Trusted Computing Base (TCB). The core of Click is already quite small (6MB for a statically linked binary section that is loaded in the memory). We decrease its size by removing the unnecessary Click elements at the build time. Importantly, we design ShieldBox with the packet-related DPDK data structures running outside of the enclave. More specifically, the TCB in our case comprises of the following components: the CPU and the microcode that implements the SGX functionality; code and data of SCONE's C library as well as its remote attestation mechanism, DPDK (except for the actual packet buffers), and Click. All other components are untrusted, and their compromise can

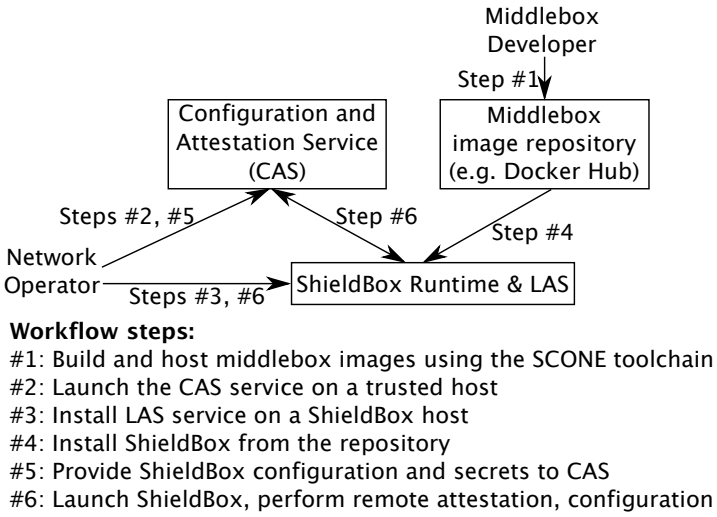

Figure 5.3: ShieldBox system workflow.

not lead to security failure.

Threat model. We target a scenario where the middleboxes that process confidential data are deployed in the untrusted cloud environment (or at the edge computing nodes) [267]. In this context [194, 268], attackers might try to learn the contents of encrypted data packets and system configuration such as cryptographic keys, filtering and classification rules, etc. Furthermore, attackers might try to compromise the integrity of the middlebox by subverting its execution.

To circumvent such attacks, we protect against a very powerful adversary even in the presence of complex layers of software in the virtualized cloud computing infrastructure. More specifically, we assume that the adversary can control the entire system software stack, including the OS or the hypervisor, and is able to launch physical attacks, such as performing memory probes.

We rely on Intel SGX to protect against direct memory-reading attacks by the privileged software. This guarantees confidentiality, integrity, and freshness of the SGX-protected memory pages. We also assume the attacker can launch memory safety attacks by forging pointers into trusted memory and pass them to ShieldBox [190, 231, 100]. For protection against this attack, see §5.5.6. Also, as noted in §5.5.5, all time sources available to ShieldBox are untrusted and can be potentially used as an attack vector.

However, we note that ShieldBox is not designed to protect against side-channel attacks [297], such as exploiting timing and page fault information, or microarchitectural attacks, such as Spectre and Foreshadow [180, 93]. Furthermore, since the underlying infrastructure is controlled by the cloud operator we cannot defend against denial-of-service attacks. We also assume that an attacker can arbitrarily reorder or drop packets—we take no particular actions against such attacks. Middlebox developers should protect against these attacks using appropriate cryptographic primitives, if necessary.

System workflow. Figure 5.3 shows the system workflow of ShieldBox. As a preparation for the deployment, developers build middlebox container images and upload them to an image repository (such as Docker Hub [219, 12]) using the SCONE toolchain. A network operator who wants to deploy a middlebox to the cloud should bootstrap a Configuration and Attestation Service (CAS) on a trusted host, and a Local Attestation Service (LAS) on

the host that will be running the middlebox (detailed in §5.5.1). After this, ShieldBox can be installed on the target machine in the cloud using the container technology—either manually or deployed as a container image from the image repository. Alternatively, it can be installed by transferring a single binary to the target machine.

The ShieldBox framework is bootstrapped using the Palaemon Configuration and Remote Attestation Service (CAS) (§5.5.1) [138]. The CAS service is launched either inside an SGX enclave of an (already bootstrapped) untrusted machine in the cloud or on a trusted machine under the control of the network operator outside the cloud. Middlebox developers implement the necessary NFs as Click configurations and send them to the CAS service together with all necessary secrets (cryptographic keys, proprietary IDS rules, etc.).

Once the operator launches ShieldBox, it connects to the CAS and carries out the remote attestation (§5.5.1). If the attestation is successful, the ShieldBox instance receives the configuration and necessary secrets. Thereafter, ShieldBox executes user-defined Click elements, which are responsible for reading packets in the userspace memory directly from NIC, performing network traffic processing, and sending them back to the network. All elements run inside an SGX enclave. Packets that must be processed under SGX protection are copied into the enclave explicitly. We efficiently execute the expensive network I/O operations (to and from the enclave memory) by using our high-performance I/O library for shielded execution based on DPDK. To summarize, ShieldBox provides the following benefits:

- **Security:** ShieldBox provides strong confidentiality and integrity for the middlebox execution by leveraging SGX enclaves.
- **Performance:** ShieldBox achieves near-native throughput and latency by building a high-performance network-I/O architecture for shielded execution by optimizing the combination of SCONE and DPDK.
- **Generality:** ShieldBox supports a wide range of NFs, as supported in the plain-text network processing, without restricting any functionalities by leveraging Click's simple and generic programming model.
- **Transparency:** ShieldBox provides network operators a portable, configurable, and verifiable architecture for seamless deployment of middleboxes. It builds on the container technology, and therefore, the changes to the software source code and deployment methods are kept at the minimum.

Limitation. We note that neither DPDK nor Click have built-in functionality for flow-based stateful traffic. More precisely, it has no functionality to reconstruct flows and process packets in flow context using Click or DPDK— this functionality must be added to the C/C++ core of these applications. This implies that ShieldBox currently supports NFs that work on L2 and L3; as only restricted processing of L4-L6 traffic is supported, which does not require flow reconstruction, for example checksum verification, but not direct reads and writes of data from a flow.

While this limitation is at odds with the stated goal of *Generality*, we argue that applications that are operating at L4-L6 typically use different programming models, which fit badly into the packet-oriented programming model of Click, designed for ease of L2-L3 processing. Additionally, the operating system interfaces for working with flow-based traffic are generally well-optimized and may not warrant the use of kernel bypass technologies. Also, maintaining a state per each flow inside enclave requires efficient state storage, otherwise EPC paging

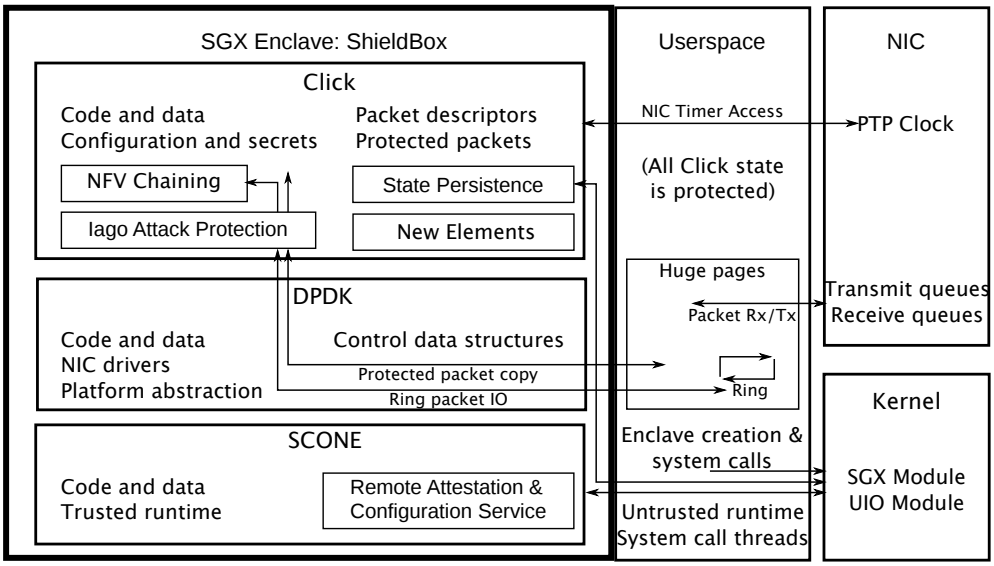

Figure 5.4: Detailed design of ShieldBox.

would cause an excessive slowdown [114]. We believe that to efficiently implement flow-based NFs, a foundation different from Click should be used.

5.5 Design Details

We next present the design details of ShieldBox. Figure 5.4 shows the detailed architecture of ShieldBox.

5.5.1 Configuration and Remote Attestation

To bootstrap a trusted middlebox in the cloud, one has to establish trust in the system components. While Intel SGX provides a remote attestation feature, a holistic system must be built for remote attestation and secure configuration of network appliances [260]. To achieve this goal, ShieldBox relies on an early version of Palaemon, a generic remote attestation and configuration framework. We present its protocol flow in Figure 5.5, as well as provide its telegraphical summary below.

To attest an enclave using Intel Remote Attestation, a verifier (operator of a ShieldBox instance) connects to the application and requests a quote. The enclave requests a report from SGX hardware and transmits it to the Intel Quoting Enclave (QE), which verifies, signs, and sends back the report. The enclave then forwards it to the verifier. This quote can be verified using the Intel verification service [23].

The Palaemon remote attestation system extends Intel's RA mechanism and is integrated with a configuration system, which provisions ShieldBox with its configuration in a secure way using a trusted channel established during attestation. This system consists of an enclave startup routine embedded in the SCONE library, Local Attestation Service (LAS), and Configuration and Attestation Service (CAS).

- The enclave startup routine takes control before ShieldBox's `main` function is called and interacts with LAS and CAS to carry out remote attestation, and allows securely setting

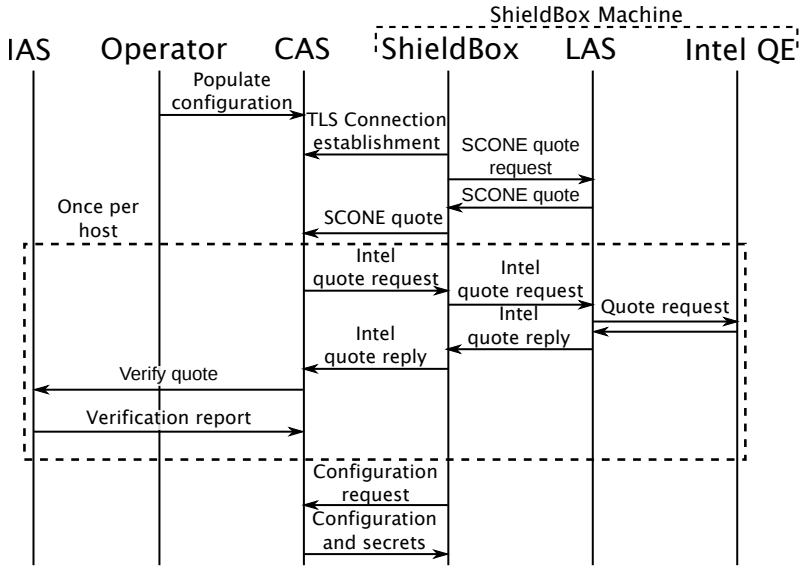

Figure 5.5: ShieldBox's remote configuration and attestation service.

environment variables, command-line arguments, and encryption keys for the SCONE shielding layer.

- Local Attestation Service is running on the same machine as ShieldBox middlebox. It, eventually, acts as the root of trust for remote attestation once CAS trusts LAS. On each host, LAS only has to attest itself one time using the Intel RA mechanism to CAS. This decouples our system from the Intel Attestation System after this initial attestation.
- Configuration Attestation Service is running on a logically single (possibly replicated) node and stores configuration and secrets of the services built with SCONE. It builds trust into unknown LAS using Intel Attestation Service (IAS), maintains information about attested LAS instances, and provisions configuration to applications using the startup routine.

To bootstrap the system, the operator launches CAS, either on the host under her control or the host in the cloud inside an SGX enclave. Then, the CAS service is populated with configurations and secrets using the REST API or a command-line configuration tool. LAS instances are launched on cloud hosts that will run ShieldBox instances either by the operator or the cloud provider. During startup, SCONE's startup routine in each ShieldBox instance establishes a TLS connection to the CAS. Simultaneously, it connects to the LAS to request a SCONE quote that is forwarded to CAS. In case the LAS instance is not yet trusted, CAS uses Intel's RA mechanism to attest it. After the trustworthiness of LAS is established, ShieldBox's SCONE quote is verified by CAS proving the binary's integrity and establishing whether it is running under SGX protection. After that, CAS ensures that the TLS connection is originating from the ShieldBox instance it received the quote of preventing man-in-the-middle attacks. Thereafter, ShieldBox obtains its configuration from the CAS service and transfers control to main ShieldBox code.

`ToEnclave`	Transfers a packet to enclave, frees the original packet
`Seal(Key, Security Association state)`	Encrypts the packet with AES-GCM
`Unseal(Key, Security Association state)`	Decrypts the packet with AES-GCM
`HyperScan(rule database)`	High-performance regular expression matching engine
`DPDKRing(Ring name)`	Transfers a packet to the DPDK ring structures
`StateFile(Key, path)`	Provides settings to the persistent state engine

Table 5.1: New specialized elements of ShieldBox.

5.5.2 Secure Elements

As described in §5.4, we design ShieldBox with the packet-related data structures of DPDK stored outside the enclave. Therefore, we need an efficient way to support the communication between DPDK and the enclave memory region. In particular, we have to consider the overheads of paging in the SGX-encrypted pages from the main memory and copying of the data between the protected and unprotected memory regions. When possible, the data packets with plain-text contents should not be needlessly copied into the enclave, as this copying will degrade the performance. Therefore, we design specialized secure Click elements (shown in Table 5.1) for copying the data packets into/outside the enclave to facilitate efficient communication.

By default, packets are read from NIC queues into the untrusted memory. This reduces the overhead of using SGX when processing packets that are not encrypted and can be safely treated with fewer security mechanisms involved. Such packets are immediately forwarded or dropped upon header inspection. On the other hand, we must move packets into the enclave memory with an explicit copy element. We have implemented such an element (`ToEnclave`), and use it to construct secure packet processing chains.

We have also added support for the commonly used AES-GCM cipher into ShieldBox (`Seal` and `Unseal` elements). This allows us to construct VPN systems that use modern cryptographic mechanisms. These elements were implemented using the Intel ISA-L crypto library. We use CAS to distribute the VPN traffic encryption keys.

To allow the creation of high-performance IDS systems based on ShieldBox, we have created an element based on the `HyperScan` regular expression library. It allows fast matching of multiple regular expressions for the incoming packets, simplifying the implementation of systems like Snort [43].

We have also added elements that implement more broad mechanisms: `DPDKRing` (§5.5.3) for NFV chaining (§5.5.3), and `StateFile` (§5.5.4) persistent state storage for ShieldBox elements.

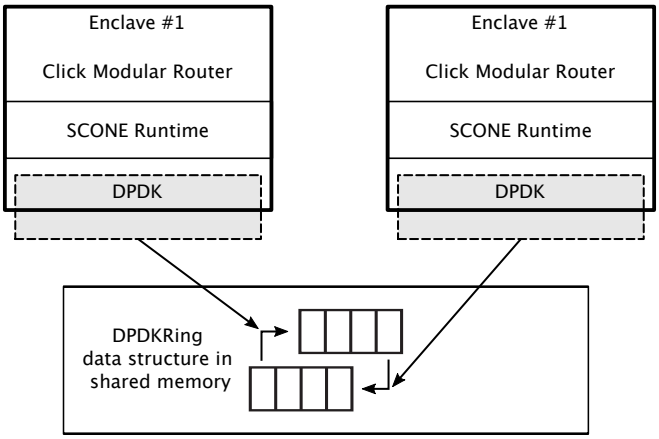

Figure 5.6: NFVs chaining in ShieldBox.

5.5.3 NFVs Chaining

Typically NFVs are chained together to build a dataflow processing graph with multiple Click elements, spanning across multiple cores, sockets, and machines [238, 173]. The communication between different cores and sockets happens through the shared memory, and communication across machines via NICs over RDMA/Infiniband. DPDK supports NUMA systems and always explicitly tries to allocate memory from the local socket RAM nodes.

However, unlike normal POSIX applications, SGX enclaves cannot be shared across different sockets[1]. As a result, in the current Intel SGX architecture, the users would need to run one ShieldBox instance per each CPU socket. Another important reason for cross-instance chaining is the collocation of middleboxes from different developers that do not necessarily trust each other. In this case, developers would want to leverage SGX to protect the secrets. Therefore, the ShieldBox framework must provide an efficient communication mechanism between enclaves to support high-performance NFVs chaining.

We built an efficient mechanism for communication between different ShieldBox instances by leveraging existing DPDK features. In particular, DPDK already provides a building block for high-throughput communication between different threads or processes with its ring API. This API contains highly-optimized implementations of concurrent, lockless FIFO queues which use huge page memory for storage. We have implemented the `DPDKRing` element (see Table 5.1) for ShieldBox to utilize it for chaining. As huge page memory is shared between multiple ShieldBox instances, the ring buffers are shared as well and can be used as an efficient way of communication between multiple ShieldBox processes. We do not use SCONE concurrent queue implementation for this task, as this would require additional changes to the DPDK, which we would like to avoid.

This solution requires assigning ownership of all shared data structures to a single process. For this, we rely on the DPDK distinction between primary and secondary processes. Primary processes, the default type, request huge page memory from the OS, allocate memory pools and initialize the hardware. Secondary processes skip device initialization and map the huge page memory already requested by the primary process into their own address space. To support network function chaining using multiple processes, we added support for starting ShieldBox instances as secondary DPDK processes.

[1] The future SGX-enabled servers might have support for the NUMA architecture

`Seal(StateFile)` Seals elements' state in the `StateFile`
`Unseal(StateFile)` Unseals elements' state from the `StateFile`
`Persist(timer, StateFile)` Periodically persists the state to `StateFile`

Table 5.2: ShieldBox APIs for state persistence. The code of each ShieldBox component must be extended to serialize the relevanta state.

Depending on the process type, the `DPDKRing` element either creates a new ring (primary process) or looks up an existing ring (secondary process). In ShieldBox, packets pushed towards a DPDKRing element are enqueued into the ring and can be dequeued from the ring in another process for further processing. Bidirectional communication between two processes can be established by using a pair of rings. Depending on the number of processes/threads enqueuing into and dequeuing from the ring, it can be configured as SPSC or MPMC ring.

One drawback of running multiple, cooperating DPDK applications is that the ASLR needs to be disabled in most cases so that huge page memory mappings are established at the same addresses in all participating processes, which might impose a security risk for the applications. This is required because data is passed through rings by pointers and those need to point to the same memory locations in all processes to reference the correct data. We note that this is not a significant problem for SGX enclaves, as they require additional in-enclave code for randomization [265].

5.5.4 Middlebox State Persistence

Middleboxes often maintain useful state (such as counter values, Ethernet switch mapping, activity logs, routing table, etc.) for fault-tolerance [266], migration [236], diagnostics [296], etc. To securely store this state in persistent memory, we extend ShieldBox with new APIs (shown in Table 5.2) for the state persistence. The `Seal` primitive is used to collect the state that must be persisted from the elements, and write it down in encrypted form to disk. `Unseal` reads this state from disk, decrypts it, and populates the elements with this state. In order to allow secure cryptographic key generation inside the enclave, we expose SCONE functions for getting SGX `Seal` keys to the ShieldBox internal APIs.

To configure this functionality, we have added a new configuration element to ShieldBox, called `StateFile` (see Table 5.1). Its parameters are the file to which state should be written, and the key that should be used for encryption. Note that this information is transmitted to ShieldBox instance in the configuration string via remote attestation, and is not accessible outside the enclave. We do not use SCONE file system shield but encrypt and decrypt the file as a single block instead. This ensures the confidentiality and integrity of stored data via the use of the AES-GCM cipher. While the current implementation does not protect against rollback attacks, we consider it to be an out-of-scope problem, which can be solved using a state-of-the-art approach such as ROTE [212] or by writing the encrypted data to a storage system like PESOS [185]

We do not attempt to extract the relevant state transparently. Instead, we rely on the programmer to provide necessary serialization routines that save only necessary parts of

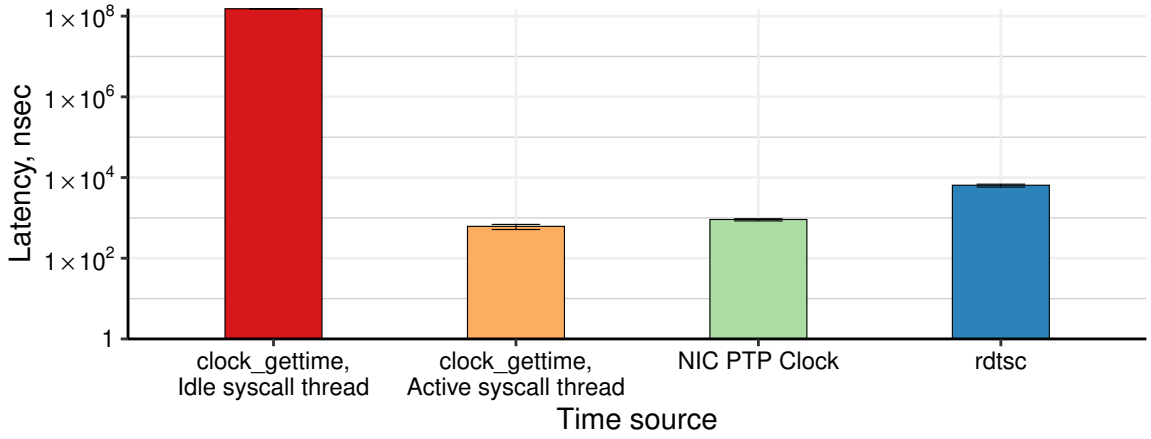

Figure 5.7: Access latency of different time source inside Intel SGX enclaves.

the element state. These routines are available in ShieldBox as the read and write handlers, and are triggered in the ShieldBox startup procedure after the configuration is loaded and parsed, and after the initialization of the basic components is finished, or manually via the `ControlSocket` interface of the `StateFile` element. It is also possible to trigger them periodically via a timer.

5.5.5 NIC Time Source

The timer is one of the commonly used functionalities in middleboxes [211, 238]. It is used for a variety of purposes such as measuring performance, scheduling NFs, rate limiting, and so on.

The time measurement can be fine-grained or coarse-grained based on the application requirements. For the fine-grained cycle-level measurements, developers use `rdtscp` or `rdtsc` instruction, which are extremely cheap and precise. Whereas for the coarse-grained measurements, applications invoke system calls like `gettimeofday` or `clock_gettime`, which on Linux is typically implemented as an extremely low overhead vDSO call.

However, in the context of SGX enclaves, both `rdtsc` and system calls have unacceptable latency to use in middleboxes for the line rate processing. More specifically, instructions that access x86 Timestamp Counter are forbidden inside the enclave, and therefore, it causes an enclave exit event. On the other hand, asynchronous system calls in SCONE are submitted through a system call queue that is optimized for the raw throughput, but not latency: as system call threads back off when the concurrent queue has no enqueued elements for prolonged periods of time, the latency of the system call increases dramatically.

We perform access latency measurements with these time sources and present the results in Figure 5.7. One can see that the latency of `clock_gettime` system call when the SCONE system call thread is idle, that is it is in the back-off state due to lack of submitted system calls, access to the timer can take up to 150 msec, which is unacceptable overhead for a high-performance middlebox. On the other hand, when the system call thread is actively spinning, the system call can be processed in approximately 0.6 μsec. Thus, disabling back-off in system call threads should provide a timer of acceptable latency to ShieldBox. Unfortunately, disabling back-off has several drawbacks: increased CPU resource consumption due to collocation of system call threads and enclave threads on sibling hypercores, fewer

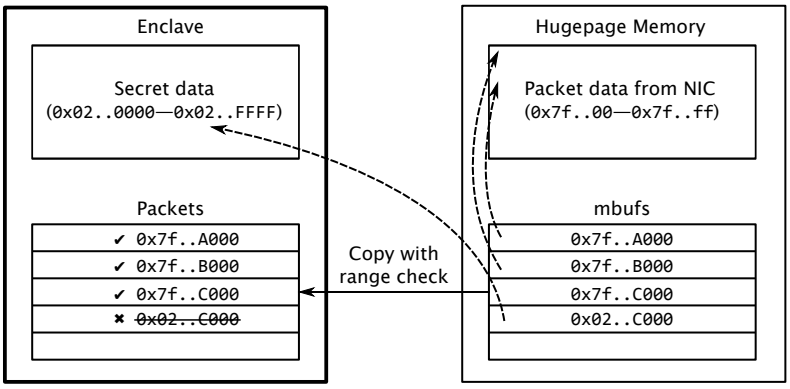

Figure 5.8: DPDK-specific Iago attack prevention in ShieldBox.

possibilities for frequency downscaling, and starvation of system tasks. Therefore, we do not configure SCONE to disable back-off. Also, as expected, `rdtsc` instruction, which has a nominal latency of approximately 20 cycles, has a latency of 6.5 μsec inside the enclave due to the asynchronous enclave exit that it causes. Therefore, a different time source is required.

To overcome these issues, we use the on-NIC PTP clock as the clock source for the enclave. This clock can be read inside the enclave reasonably fast (0.9 μsec, which is on the same scale as reading HPET). Moreover, it neither causes enclave exits nor requires submitting system calls. Furthermore, the on-NIC clock is extremely precise since it is intended to use for the PTP synchronization protocol.

We note that this time source is not secure, and can be used as a DoS attack vector by a malicious OS. However, we note that in the case of ShieldBox, time source is not used for security-critical tasks, and thus its manipulations can only lead to performance degradation or denial of service. Thus, providing a trusted, efficient and precise time source is out-of-scope of this chapter; we provide our solution in the context of leases in the Chapter §7.

Lastly, we note that the HPET timer can also be used for our purpose. We argue against its usage since it has to be enabled for the entire system in the kernel configuration to become available. Thus, it would negatively affect the performance of native `POSIX` applications running on the system, which typically use the TSC-based system time source.

5.5.6 Memory Safety for DPDK-Specific Iago Attacks

Iago attacks [100] are a serious class of security attacks that can be launched on shielded execution to compromise the confidentiality and integrity properties. In particular, an Iago attack originates through malicious inputs supplied by the untrusted code to the trusted code. In the classical setting, a malicious OS can subvert the execution of an SGX-protected application by exploiting the application's dependency on correct values of system call return values [83].

The decision (§5.4) to allocate huge pages for packet buffers and DPDK rings has security implications. The fact that packets are passed through rings by reference, and DPDK buffers contain pointers, opens a new attack surface. Attackers with access to this memory region could modify pointers to point into the SGX-protected regions and make the enclave inadvertently leak secrets over the network [190, 231].

The scenario for Iago attack on DPDK is depicted in the Figure 5.8: DPDK maintains a memory buffer associated with each received packet in the unprotected memory. The attacker

adds a maliciously crafted memory buffer descriptor (mbuf) with an offset or data address pointing to the enclave into the `rte_ring` structure. If NF sends all packets that, for example do not have an IP header to the output, this could leak memory content, and thus exfiltrate secrets like encryption keys or plaintext of the encrypted traffic. We consider this attack vector serious enough to require a defense.

To protect against DPDK-specific Iago attacks, we have implemented a pointer validation function. More specifically, the scheme uses an enclave parameter structure that is located inside the enclave memory and defines the enclave memory boundaries. Memory buffer descriptors are validated by checking if they do *not* overlap with the enclave memory range [*base*, *base* + *enclave_size*). The check happens after copying the descriptor into the enclave, which is necessary to prevent a time of check vs. time of use vulnerability. We note that ShieldBox is already protected against the classical syscall-specific Iago attacks through SCONE's *shielding* interfaces.

This ensures that no pointers possibly pointing to the secrets stored in EPC are accepted through the unprotected huge page memory. Pointers can still be modified by a malicious attacker, but they can only point to the unprotected memory. However, if they point to the unmapped virtual memory, the operating system will terminate the application. Furthermore, security measures such as `ASLR` also makes it harder for the attackers to find a valid attack vector [265].

As it is possible for an application to enqueue and dequeue arbitrary pointers into DPDK's `rte_ring` structures, it is not easily possible to integrate this pointer check directly into DPDK. Instead, we implement these pointer checks in the `DPDKRing` and `FromDPDKDevice` (§5.5.3) elements. If ShieldBox detects a malicious pointer, it assumes an attack, notifies the application operator, and drops the packet.

5.6 Implementation

5.6.1 Interaction with SCONE and Hardware

We build on SCONE to simplify porting of DPDK and Click. We needed to apply `musl-libc` compatibility patches only to DPDK to make the system run as an Intel SGX enclave — that is, we did not apply any SGX-specific patches. SCONE provides ShieldBox the memory management and remote attestation and configuration subsystems. Using Intel SGX SDK would require numerous wrappers for system calls, and thus would force substantial changes to Click and DPDK. We could have used Graphene-SGX [287] without significant drawbacks: its only drawback at the time this work was performed was the lack of built-in remote attestation features (added in version 1.1) and secure configuration service of SCONE. We next describe how we adapted SCONE for our system.

System startup. When ShieldBox starts, it performs remote attestation and obtains the configuration. ShieldBox initializes the DPDK subsystem, allocates huge page memory, and takes control over NICs that are available. Then, it starts running the Click element scheduler, which reads packets from the NIC and passes them through the processing graph until they leave the system or are dropped.

System calls. As one of the goals of the ShieldBox is high throughput and low latency, we minimize the rate of system calls in the fast path of the application, as this would make it impossible for us to sustain the line rate. On the other hand, systems calls are necessary for

the application startup, as it is necessary to do remote attestation, gain access to NIC, and so on. Thus, the asynchronous system call subsystem is mostly idle after the startup and causes no runtime overhead. On the other hand, it cannot be disabled completely during the operation of ShieldBox, as it is necessary, for example, to update the configuration or read statistics via the ShieldBox control socket.

Memory management. When the SCONE runtime starts the application, it automatically places the application code, statically allocated data, and heap (memory allocated via `malloc`, `mmap`) in the SGX-protected EPC memory. This mechanism is in contrast to the way DPDK operates—DPDK by default allocates memory using `x86_64` huge pages, which reduces the TLB miss rate and ensures continuous physical memory layout. Such pages are not supported inside the enclave; besides that, the NIC can only deliver packets to the unprotected memory, and network traffic entering or leaving the machine can be modified by an attacker. Therefore, we keep the huge pages enabled in DPDK outside the enclave, *and explicitly copy packets that must be processed with SGX protection into the enclave.* With this scheme, DPDK-created packet data structures are allocated outside the SGX enclave. We support an efficient data transfer between the DPDK and enclave and processing inside the enclave using the new secure Click elements (detailed in §5.5.2).

Accessing huge pages in DPDK does not require bypassing SCONE, because of the specific way DPDK allocates huge pages. In particular, instead of passing `MAP_HUGETLB` flag to `mmap` system call, it opens shared memory files in the `hugetlbfs` virtual filesystem and passes those file descriptors to the `mmap` call. SCONE does not apply shielding in this case, so no additional modifications are necessary (§ 3.3.4).

Partitioning ShieldBox. Another design aspect that is always present in designing software for Intel SGX is the question of partitioning. One of the components that we could have moved outside of the enclave is DPDK: in the end, NIC cannot deliver data into the enclave, as this would violate the SGX security mechanism, and thus a big part of DPDK data is located outside of the enclave. Therefore, DPDK can be easily moved out of the enclave. This would open two possibilities for interaction with enclave: via concurrent queue in shared memory or synchronously via enclave enters/exits. We argue that both approaches are suboptimal from the performance point of view.

If we use a synchronous interface, we would have to constantly execute enclave enters and exits, which have extremely high runtime cost. If we attempted to increase packet batch size to reduce this cost via batching, we would be processing batches higher than L3 size, and this would further reduce the performance. On the other hand, if we use a concurrent queue for communication, this leads to another problem: in such a partitioning scenario part of the cores would be wasted, because they only read packets from the network into the concurrent queue, reducing the number of cores available for useful work. Therefore, we conclude that having DPDK inside the enclave is the optimal solution for achieving high performance inside SGX enclaves. We do acknowledge that this approach increases the system TCB.

5.6.2 Toolchain

We build ShieldBox's toolchain using DPDK (version 16.11) and Click (master branch commit `0e860a93`). We further integrate it with the SCONE runtime to produce ShieldBox. We use `gcc` version 6.3.0 for the compilation process. We use Boost `C++` library (version 1.63) to build a static version of the Hyperscan high performance regular expression matching engine (master branch commit `7aff6f61`) and incorporate it into ShieldBox. We use `WolfSSL` li-

brary [51] to implement `StateFile` sealing and packet `Seal/Unseal` elements. The toolchain contains automated scripts for building and deploying middlebox images, and setting up ShieldBox and CAS services (as described in the system workflow in §5.4).

To make the compilation of ShieldBox work with SCONE, some changes to DPDK were necessary. In particular, we had to remove the helper functions for printing stack tracebacks and provide some `glibc`-specific structures, macros, and kernel header files. Click required no adaptions since it is implemented in `C++` mostly using high-level APIs.

The resulting ShieldBox binary is 8.2 MiB in size and requires approximately 16 MiB at runtime, including minimal runtime stack and heap allocation. This implies that we could run roughly up to six instances of ShieldBox in parallel on one processor without impacting the performance by EPC paging (94 MiB).

5.6.3 Optimizations

To further improve the performance, especially for the case of DPDK running inside the enclave, we optimized the data path inside Click. We use the Linux performance profiling tool `perf` [39] to find the bottlenecks. We further optimize the data path inside Click, especially for the case of DPDK running inside the enclave, by identifying the performance bottlenecks using the `perf` [39] tool.

Memory pre-allocation. The `FromDPDK` element allocates memory for packet descriptor storage on the stack each time the `run_task` function is called. We pre-allocate this memory once in a constructor instead.

Branching hints. We insert GCC-specific `unlikely` / `likely` attributes in several `if`-clauses. These attributes instruct the compiler to construct the branch in such a way that the CPU will assume it as not taken or taken correspondingly upon first execution.

Queue optimization. In the `ToDPDKDevice` Click element we replace the inefficient implementation of the queue, which used `std::vector` from the `C++` standard library, by the `rte_ring` structure provided in DPDK.

Timer event scheduler optimization. In the Click timer event scheduler, we optimize the code to reduce the number of `clock_gettime` system calls. This optimization allows us to reduce the latency in short element chains to the native level.

5.7 Evaluation

5.7.1 Experimental Setup

Testbed. We evaluate ShieldBox using two machines: (1) load generator, and (2) SGX-enabled machine. The load generator is a Broadwell Intel Xeon D-1540 (2.00GHz, 8 cores, 16 hyper-threads) machine with 32GB RAM. The SGX machine under test is Intel Xeon E3-1270 v5 (3.60GHz, 4 cores, 8 hyper-threads) with 32GB RAM running Linux kernel 4.4. Each core has private 32KB L1 and 256KB L2 caches, and all cores share an 8MB L3 cache. The load generator is connected to the test machine using a 40 GbE Intel XL-710 network card. We use `pktgen-dpdk` for throughput testing. The load generator saturates the link starting with 128-byte packets.

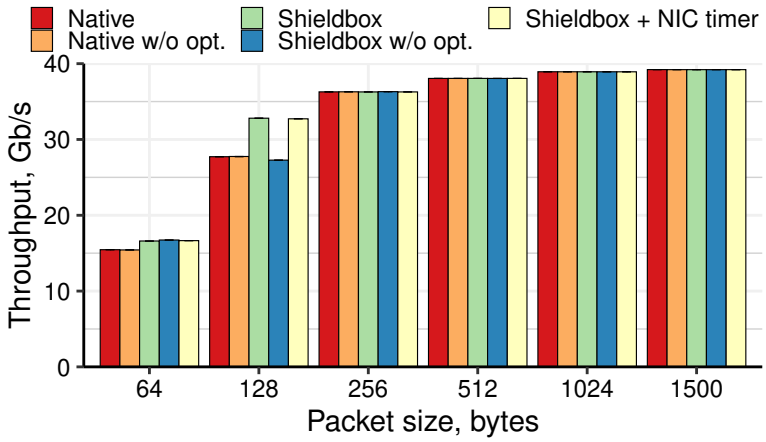

Figure 5.9: Throughput of a Wire function as a function of packet size.

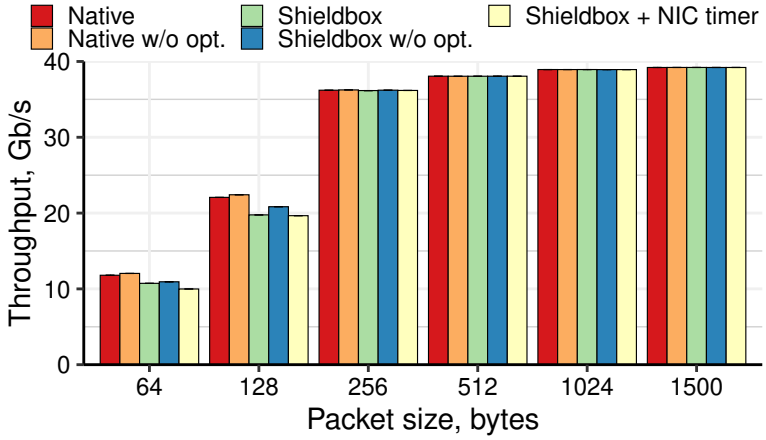

Figure 5.10: Throughput of an EtherMirror function as a function of packet size.

Applications. For the micro-benchmarks, we use three basic Click elements: (1) `Wire`, which sends the packet immediately after receiving; (2) `EtherMirror`, which sends the packet after swapping the source and destination addresses; and (3) `Firewall`, which does packet filtering based on PCAP-like rules.

For the case-studies, we evaluate ShieldBox using two applications: (1) a multiport `IPRouter`, and (2) an `IDS`.

Methodology. For the performance measurements, we consider several cases of our system:

- Native (Non-SGX) with and without generic optimizations.
- SGX-enabled ShieldBox with and without optimizations.
- SGX-enabled ShieldBox with the on-NIC timer.

We use native Click as the evaluation baseline since it is the worst-case scenario for us. Lastly, unless stated otherwise, `ToEnclave` element is not used in the benchmarks.

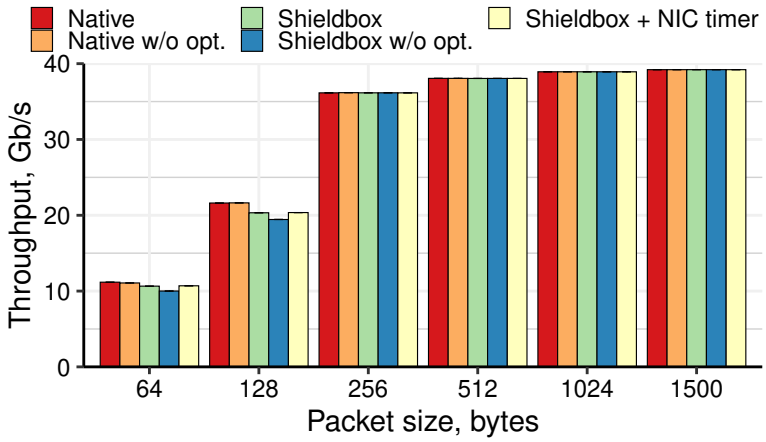

Figure 5.11: Throughput of a Firewall function as a function of packet size.

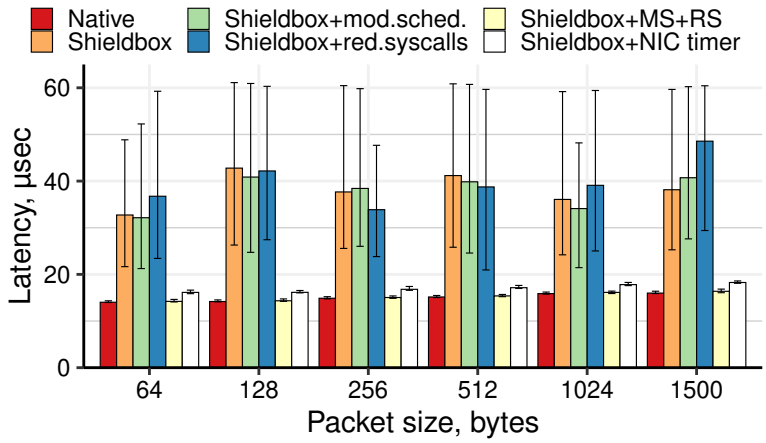

Figure 5.12: Latency of an EtherMirror function as a function of packet size.

5.7.2 Throughput

We first report ShieldBox's throughput with varying packet size running on four cores. Figure 5.9, Figure 5.10, and Figure 5.11 present the throughput for `Wire`, `Ethermirror`, and `Firewall`, respectively.

The results show that the performance of ShieldBox matches the performance of Click. In the case of Wire application with the packet sizes smaller than 256 bytes, ShieldBox is better than the native version. This is explained by the fact that Click timer event scheduler optimization is missing in the native Click, which removes some system call overhead from the `Wire` application. The impact is smaller with other applications because they contain elements that reduce the relative overhead of Click scheduler. We also see that ShieldBox achieves the line rate at 512 byte packets.

5.7.3 Latency

We also measure the packet processing latency using the following scheme: the load generator continuously generates a UDP packet and waits for its return from the enclave. We study packet round-trip time measured at the load generator. On the ShieldBox instance,

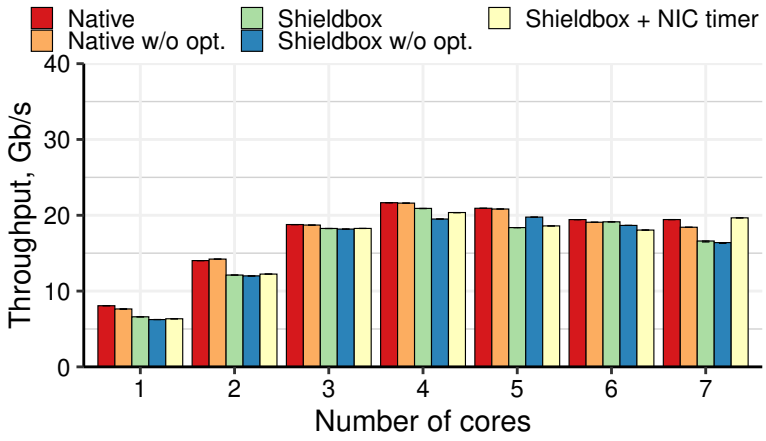

Figure 5.13: Throughput of a Firewall function with an increasing number of cores. System scales up to four cores in all configurations.

we are running the `EtherMirror` application. For these measurements, we do not perform any latency-specific tuning of the environment other than thread pinning, which is enabled by default in DPDK. We expect that a production system with stringent requirements for low latency will use `SCHED_FIFO` scheduler and have isolated cores.

Figure 5.12 presents the latency measurements for `EtherMirror` with varying packet size. The poor performance of ShieldBox without optimizations is explained by the fact that ShieldBox executes `clock_gettime` system calls in the timer event scheduling code. SCONE system calls are optimized for raw throughput with a large number of threads, but not for low latency; this makes the latency measurement result 3× worse than the native execution. We have considered the following latency optimizations:

- Reduced system call rate for immediately-scheduled timer events. It removes one system call round-trip from the packet latency.
- Modified scheduler that prioritizes immediately-scheduled events and allows to remove a system call from scheduler if there are no periodic timer events.

One of the surprising results that we have is that each of these optimizations does not have a statistically significant influence when applied individually, which can be explained by the fact that once the system call thread has left the back-off mode, it will execute system calls with low individual overhead. On the other hand, when applied simultaneously, they return the latency to almost-native levels—the influence of SGX and SCONE on the latency is extremely small.

We consider using a NIC timer as a separate optimization. One can see that reading a NIC timer is a costly operation; it happens twice per packet in our measurements, adding approximately $0.9 * 2 = 1.8\mu$sec to the total latency. On the other hand, it is much faster than executing clock-reading system calls, and can further improve system timeliness when combined with other optimizations.

5.7.4 Scalability

We next evaluate ShieldBox's scalability with an increasing number of cores. Figure 5.13 presents the throughput for `Firewall` with 128 byte packets. The scalability of both Shield-

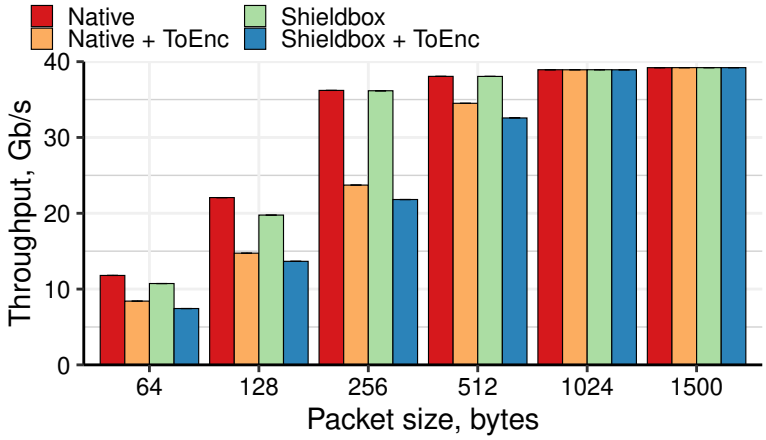

Figure 5.14: Throughput of an EtherMirror function with ToEnclave as a function of packet size.

Box and Click is limited. We can see that the performance for both native and ShieldBox peaks at four cores. This is due to the fact DPDK and ShieldBox work best with hyperthreading disabled. This is also confirmed by the poor scalability of native Click.

5.7.5 ToEnclave Overheads

Throughput. We next measure the throughput of the new secure `ToEnclave` element added in ShieldBox, which is used to copy the packet data inside SGX enclave protected memory. We evaluate the impact of this extra data copy by measuring the throughput scaling with varying packet size. Figure 5.14 shows the results for `EtherMirror`.

We can see that the overhead of the extra memory copy peaks with small packet sizes. This phenomenon is because for each received packet, operations with rather high overhead must be executed to allocate the packet. One way to reduce this cost would be to batch the memory allocation for all packets. Note that the overhead of ShieldBox compared to the native execution is relatively small: ShieldBox with `ToEnclave` is running within 88% of the native version with extra memory copy in the worst case of small packet sizes, and within 60% of the native Click without `ToEnclave` element.

Latency. The latency impact for the `ToEnclave` element is as follows: at 64 byte packets (median, 95th percentile) latency changes from (14.25, 15.04) to (14.51, 15.24) μsec, at 1500 byte packets it changes from (16.39, 17.39) to (17.49, 18.24) μsec.

5.7.6 Configuration and Attestation

We next evaluate the overheads of the configuration and attestation service in ShieldBox. The measurement results are presented in Table 5.3. The results show that remote attestation has a negligible effect on ShieldBox's startup time. Furthermore, even though TLS session establishment is a costly operation, it is performed once per instance start-up, allowing an operator to use a single CAS node for thousands of ShieldBox instances.

Phase	Average Duration, μsec
Attestation	19467
CAS communication	19301
LAS communication	1474
Configuration	825.6
Total time	26368

Table 5.3: Overheads of ShieldBox remote configuration and attestation.

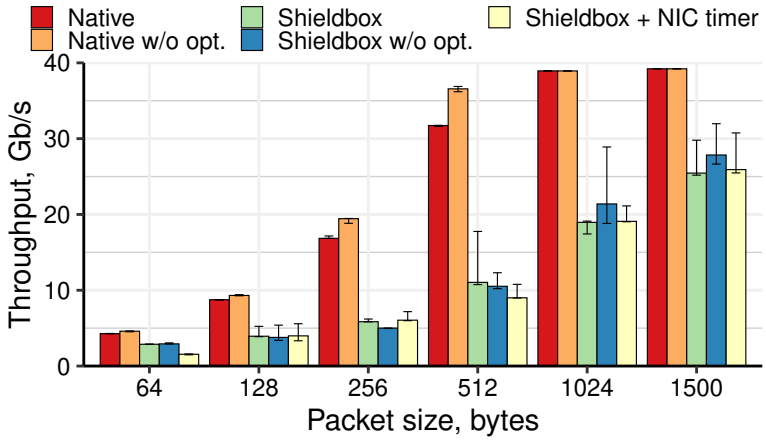

Figure 5.15: NFV chaining application throughput as a function of packet size. ShieldBox has lower scalability due to interference of system call and enclave threads.

5.7.7 NFVs Chaining

To measure the throughput of the NFV chaining scheme, we implement a chaining application. The chaining application implements packet communication between two ShieldBox instances running on the same machine through a DPDK packet ring. One instance contains an application that receives packets from the network and sends them to the other instance via the `DPDKRing` element. The second instance receives packets from the ring and sends them back through a different `DPDKRing` element. These packets are received by the first ShieldBox forming a circular ring. Thereafter, the packets are transmitted back to the load generating node. Note that the packets cross the rings twice. The chaining application showcases the worst-case scenario for us since the NF elements are not performing any computation.

Figure 5.15 presents the throughput with varying packet size for the NFV chaining application. The results show that using the ring communication causes a substantial performance drop for ShieldBox independent of the optimizations. This is mostly related to the way SCONE runs enclaves—it must allocate a constantly-running thread for the service threads created by ShieldBox and DPDK. Due to this, there is interference between the service threads and processing cores, which decreases the throughput and also increases the variance.

Importantly, note that our experiment for the NF chaining across multiple enclaves shows the scenario where two middleboxes are operated by different network operators, who may not necessarily trust each other. Whereas, the performance of NF chains within a single enclave would still be comparable to the native execution.

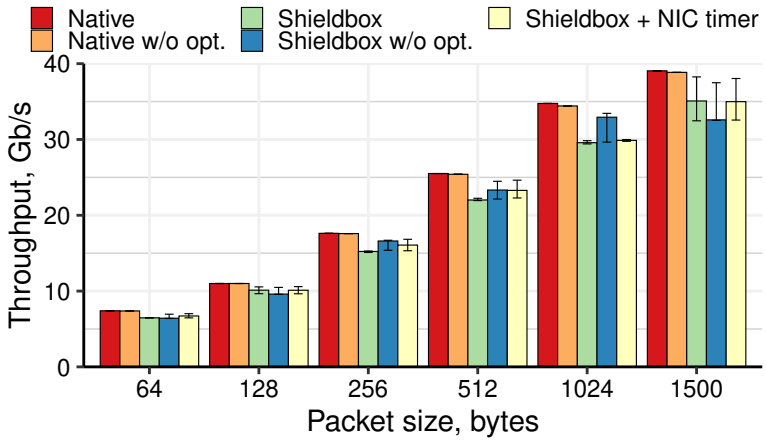

Figure 5.16: Throughput of a Seal function with varying packet sizes.

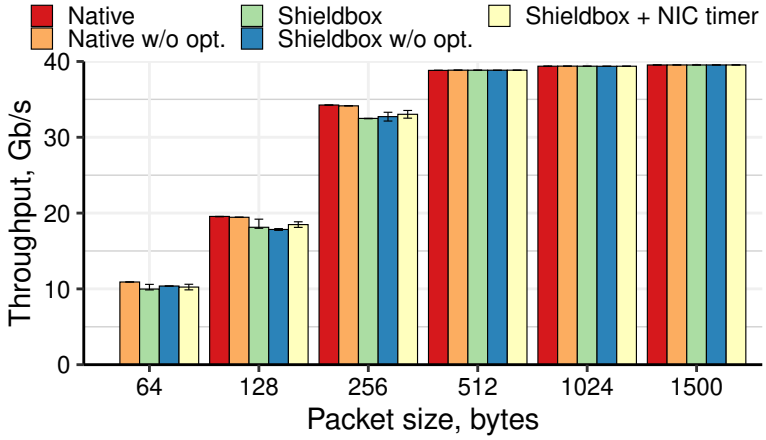

Figure 5.17: Throughput of an IPRouter application with varying packet sizes.

5.7.8 Packet Sealing Performance

We next evaluate throughput of the `Seal/Unseal` secure elements. In particular, we use our AES-GCM encryption code running inside the SGX enclave. Figure 5.16 presents the throughput of the `Seal` element with varying packet size. The result shows that the code inside SGX enclave runs within 88% of the native performance irrespective of the optimizations applied. This is explained by the fact that most of the application CPU time is spent doing the encryption. The difference between the native and SGX version can be explained by different thread scheduling strategies used by SCONE and native `POSIX`. In `POSIX`, threads are pinned to the real CPU cores, while in SCONE, the userspace threads inside enclave are pinned to the in-enclave kernel threads. This makes thread pinning non-deterministic—sometimes two threads that are to be pinned to different cores are pinned to sibling hyper-threads.

5.7.9 Case Studies

We next evaluate ShieldBox's performance with the following two case-studies: (1) `IPRouter`, and (2) `IDS`.

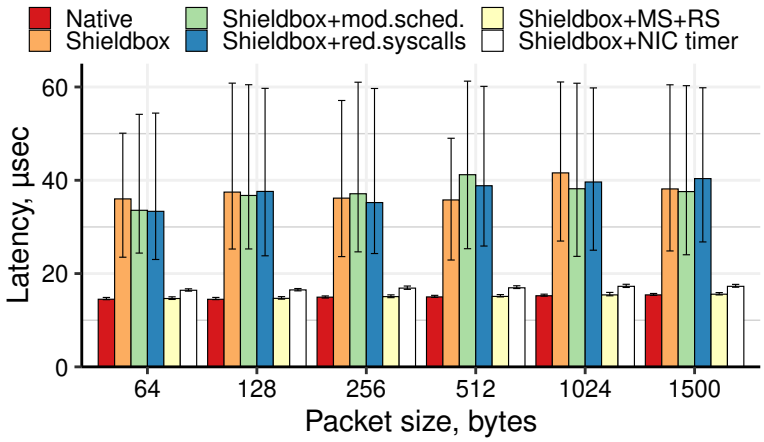

Figure 5.18: Latency of an IPRouter application with varying packet sizes.

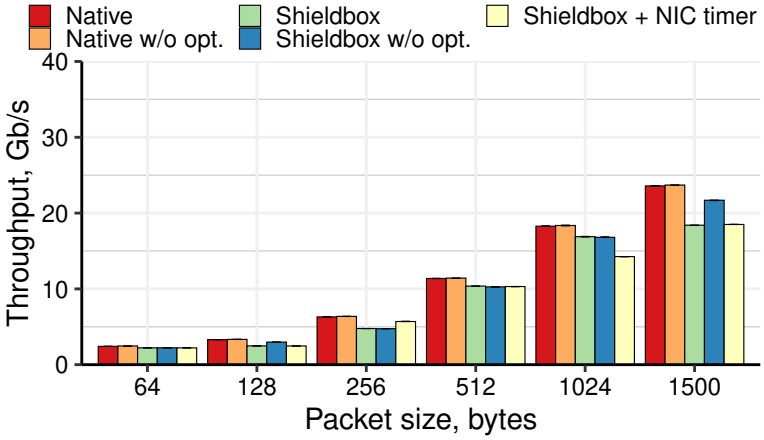

Figure 5.19: Throughput of an IDS application with varying packet sizes.

IPRouter. `IPRouter` application is an adaptation of a multi-port router Click example application to our evaluation hardware. This application first classifies all packets into three categories: ARP requests, ARP replies, and all other packets. ARP requests are answered. ARP replies are dropped. Other packets are passed to a routing table element that sends them to the NIC output port. Figure 5.17 shows the throughput of the `IPRouter` application with varying packet size. We can see that ShieldBox has the same performance as Click with packet sizes bigger than 256 bytes, and performs within 90% of Click with smaller packets.

We also measure the latency of the `IPRouter` application as presented in Figure 5.18. We can see that even if the number of elements in the application increases, the latency of the application remains the same as the native execution.

Intrusion Detection System (IDS). `IDS` application implements a network function that is commonly found in the enterprise network, where it is commonly implemented by software like Snort or Suricata. `IDS` pushes the traffic through the firewall, and then performs traffic scanning with the `HyperScan` element. Traffic that does not match any pattern is sent to the output while matching traffic passes through a counter and then dropped.

ShieldBox performs as close to the native Click execution with a slight performance drop. This drop comes from the general SGX overhead for memory accesses.

5.8 Discussion

Timestamp counter access and SGXv2. While with SGXv1 reading the x86 Timestamp Counter causes the AEX, this operation is permitted inside of the enclave with SGXv2. Unfortunately, SGXv2-based platforms were unavailable at the time this work was conducted, and even currently the available SGXv2 is available in low-power Atom CPUs, which lack SIMD features (AVX), have low core count and low CPU frequency, which all help achieve line-rate packet processing. Hence, the time access techniques presented in this chapter are still valid. However, if SGXv2 ISA extensions become commonly available, the alternative time access mechanisms will become superfluous.

Trusted peripherals. Whether DPDK is located outside of the enclave or inside, the transmission channel between the NIC and enclave is managed by the operating system, which controls the MMIO mappings and `procfs` entries necessary for acquiring the device from the OS networking stack. Therefore, the OS is always capable of performing MITM attacks against the enclave or even fully emulate the NIC. The question arises, whether it is possible to get a trusted channel to the NIC, or to other peripheral devices, for example SSDs?

One solution to this problem is to employ cryptography to secure the communication channel: PCI Express traffic and the DMA memory pages dedicated to the device. This solution requires the extension of hardware with high-performance and expensive FPGAs [131]. The CPU would either have to be extended with a generic cryptographic engine capable of decrypting the pages at line rate, or perform all cryptographic operations in software, increasing the system load.

An important and ever-present part of the trusted I/O problem is the attestation of remote devices, and secret distribution. Currently, it is an open problem in the context of existing PCI Express devices. In future devices, the authentication and measurement scheme inspired by the USB-C specification [48, 49] may be used; it is currently drafted by Intel [161].

We also note that it should be possible to remove the MMIO pages from the attack surface by changing the implementations of Intel's Data Direct I/O (DDIO)[2] [21] and Cache Allocation Technology (CAT):

- During the enclave creation, a range of L3 cache is statically and exclusively allocated to the enclave, and this range is included in the enclave measurement, using a modification of Intel CAT.
- After the establishment of a secure channel between the enclave and the device, only DDIO is used for the data transfer, without ever touching DMA pages.

Unikernels and kernel bypass. In this work, we have shown that it is possible to push the performance of the SGX enclave beyond the limitations of the system call interface, by dropping it altogether. This approach is typically utilized in high-performance storage systems that use kernel bypass for storage and network access [76, 184, 279], while most of the cloud software relies on the OS-provided POSIX interfaces. However, the importance of kernel bypass techniques may increase, in case the popularity of VM-based *unikernels* increases. As it is common to use `virtio`-based interfaces in hypervisors for efficiency reasons, unikernel libraries embed `virtio` drivers for network, disk, and PCI device access [295]. All kernel bypass techniques discussed in this chapter would be relevant if the developers decide to use SGX to secure their unikernel applications.

[2] DDIO allows direct transmission of data from a compatible PCI Express device into the L3 cache of the CPU.

5.9 Conclusion

In this chapter, we have shown how to extend Intel SGX-based enclaves to support data-intensive applications that require performance beyond what is available through the operating system call interface. We have shown that there are no significant obstacles to using SGX for securing high-performance networking applications. Most importantly, We outline and solve the problem of 2–3x latency increase in network functions caused by the slow time source.

To achieve the aforementioned goals, we have presented the design, implementation, and evaluation of ShieldBox—a secure middlebox framework for deploying high-performance network functions (NFs) on untrusted commodity servers. ShieldBox exposes a generic interface based on Click to design and implement a wide-range of NFs using its out-of-the-box elements and C++ extensions. To securely process data at line rate, ShieldBox integrates a high-performance I/O processing library (Intel DPDK) with a shielded execution framework (SCONE) based on Intel SGX. We have also added several new useful features and optimizations for secure end-to-end network processing. To improve the performance of the time source, ShieldBox relies on the PTP clock instead of the OS sources. Our evaluation using a wide-range of NFs and case-studies show that ShieldBox achieves near-native throughput and latency, which lets us conclude that kernel bypass technologies are a practical instrument of an enclave developer.

Finally, the work presented in this chapter has shown, how Intel SGX can be used to secure not only the standard POSIX software run by end-users in the cloud, but also the Network Functions and Middleboxes that the user may deploy to the cloud, thus bringing high confidentiality and integrity guarantees to one more element of the cloud stack.

6 Using Intel SGX Enclaves For Secure Remote Execution in FaaS

In the previous chapters, we have shown how to efficiently build and deploy TEEs with individual POSIX applications, including network middleboxes. However, when taking a wider look at the landscape of cloud infrastructure, it is clear that this kind of deployment is insufficient: most cloud services consist of multiple services, which are working in concert to provide service to the user, where the combinations of services can be static or dynamic. Static combinations are common in practice, for example, a machine learning system, such as Tensorflow [56], performing feature detection on the media files preprocessed by a separate service (using ImageMagick or FFMpeg). These services are typically deployed using Kubernetes or Docker Swarm; from the point of view of TEEs, they can rely on static configuration distributed by a TEE-aware configuration service, such as Palaemon [138]. On the other hand, dynamic combinations of services, exemplified by modern Function-as-a-Service systems, are more challenging to secure using Intel SGX.

Thus, in this chapter, we show how to apply TEE technologies to an existing serverless framework. We identify common use-cases and bottlenecks stemming from both serverless architecture and Intel SGX restrictions. Based on our analysis, We design and build Clemmys, a security-first serverless platform that ensures confidentiality and integrity of users' functions and data as they are processed on untrusted cloud premises while keeping the cost of protection low. We provide a generic design for hardening FaaS platforms with Intel SGX, and explain the communication protocol that our system uses to ensure confidentiality and integrity of data, and integrity of function chains. To overcome performance and latency issues that are inherent in SGX applications, we apply several SGX-specific optimizations to the runtime system: we use SGXv2 to speed up the enclave startup and perform batch EPC augmentation. To evaluate our approach, we implement our design over Apache OpenWhisk, a popular serverless platform. Lastly, we show that Clemmys achieves the similar throughput and latency to native Apache OpenWhisk in case hardware resource limits are not reached, while allowing it to withstand several new attack vectors.

6.1 Introduction

Serverless Computing. Serverless computing, or Function-as-a-Service (FaaS), is a cloud computing paradigm that has emerged to make the processing of bursty, irregular event-driven workloads cheaper, and deployment and development—simpler [59, 169]. To reap these benefits, application developers must decompose their software in terms of the core abstraction of FaaS—a *function*: a stateless, short-lived, single-purpose service that is spawned to process a single event. The stateful components, like databases and caches, need to be separated into external systems.

The serverless paradigm implies processing data with stateless functions, using a fresh runtime environment to serve each request or event. From the programmer's point of view, functions are written to an API and a set of libraries specified by the platform owner, and without any assumptions about the persistence function-local data or the underlying hardware. This concept is already implemented in multiple open frameworks [3, 36, 27, 14] and commercial platforms [5, 17, 20]).

Serverless computing runs with the promise of *automatic resource management*: the cloud operator is responsible for horizontal and vertical scaling of the client's code. The client is responsible only for uploading the code, while the cloud operators are responsible for selecting the number of instances and choosing their placement, providing runtime environment and the necessary CPU and memory resources for the computations. This shift of responsibilities, called *Backend as a Service* [169], greatly simplifies the job of application developers, who are freed from implementing virtual server management, load-balancing, and autoscaling solutions for their services, further reducing development cost. In particular, better load-balancing and scaling of user functions become possible, as the insights into resource availability, are solely available to the cloud operator.

Serverless computing is also distinguished from the classical cloud computing by its promise of *pay-per-use*: instead of billing clients based on the allocated resources, only the resources actually used for serving requests are billed. This type of billing allows performing CPU-intensive tasks in the cloud without exorbitant costs for the client, and without putting significant resource restrictions on the functions. This is a game-changer for tasks like machine learning or video processing, which were challenging to run in the cloud efficiently due to their bursty nature and high CPU and memory requirements.

Thus, the benefits that serverless computing presents to the users are twofold. First, the user is freed from making decisions about platform management, security, and software updates. These tasks are delegated to the cloud provider, who can handle them using the available infrastructure knowledge. Second, with short-lived services billed per invocation, it is possible to run in an economically efficient way even those services that are idle most of the time.

Trust issues. Despite the significant economic benefits that come with serverless computing, the trust issues could become a dealbreaker when it comes to processing sensitive data in the cloud. Specifically, two main aspects make cloud services extremely challenging to secure.

First, cloud applications run with a large set of system components, which must function correctly for the system to maintain its security properties, that is, with an excessive Trusted Computing Base (TCB). In the cloud, TCB includes the host (operating system, hypervisor), and the userspace stack running on the machine. Due to the large size and complexity of these components, they are likely to have flaws, and the attackers can probably find ex-

ploitable vulnerabilities and subvert the platform security properties.

Second, as trust in the cloud provider is unavoidable, outsourcing to the cloud implies giving the provider's employees access to the users' data. If the employees have malicious intentions, it could have grave consequences, especially in the case of medical and financial applications.

Although these issues are becoming a major obstacle to the adoption of serverless computing, no serverless platform currently tries to solve them in a principled way.

Clemmys. These challenges have motivated us to develop a system that allows users to benefit from serverless computing while preserving the security of their data.

Clemmys uses Intel SGX and SCONE to protect the integrity and confidentiality of the function code and data. To tackle this problem without incurring prohibitive overheads, only two components of Clemmys are running inside a TEE—the platform gateway and the user functions. Additionally, we develop a message format that preserves message confidentiality and integrity while the data is sent between the other, unprotected platform components.

Our contributions include:

- We design and implement a reencrypting proxy that terminates TLS connections and encodes function invocations into Clemmys message format.
- We develop a message format that preserves message confidentiality and integrity and can be used in a variety of other FaaS platforms.
- To prove the validity of our approach, we implement Clemmys as a modification of Apache OpenWhisk, a popular FaaS platform.
- To reduce the cost of protection, we additionally implement several important SGX-specific optimizations. These optimizations are orthogonal to the existing optimizations of FaaS platforms, and can be safely combined with them.

6.2 Background

Runtime environments for serverless functions. Each instance of a serverless function runs inside a platform owner-provided isolated environment, which comprises the set of platform owner-vetted libraries and management components. The choice of runtime has a big influence on the usability of a serverless platform, as it influences the platform efficiency, ease of auto-scaling, ease-of-use by the the function developer, and the achievable performance.

The main requirements to the serverless runtime are fast startup and high guarantees of performance and security isolation, which would make sharing of computational resources by mutually untrusting customers possible. These requirements are often in conflict, as the technologies that allow the strongest isolation also incur the highest overheads in the startup phase.

For example, virtualization-based runtimes rely on the mature and well-researched virtualization technology to isolate the functions of cloud tenants from each other. While security isolation of virtual machines is well-studied in the literature [174], the main problem of VMs in the context of serverless cloud computing is the large function startup time, which reaches up to one second. Virtual machines require the development of more lean hypervisors and paravirtualization solutions to be efficiently deployed in the serverless systems [60, 210].

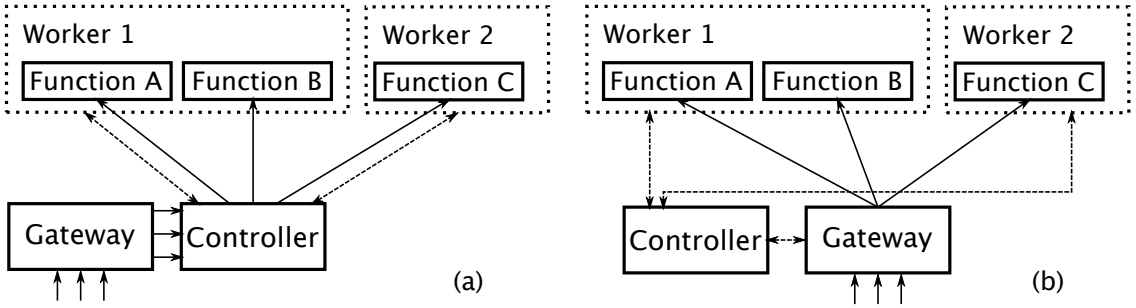

Figure 6.1: FaaS platform architectures: (a) with controller node as load balancer; (b) with gateway as load balancer.

The container-based approach provides an alternative to virtual machines, by building on the recent kernel namespacing features, and technologies like Kubernetes that automate scaling and configuration management. While containers exhibit the startup and runtime overheads smaller than the virtual machines, they also require additional isolation technologies to achieve a high level of security and performance isolation [191, 228]. Both VM-based and container-based approaches allow running arbitrary user binaries inside the functions, increasing the system flexibility.

The third approach completely dispenses with the OS-provided isolation technologies and instead relies upon the software fault isolation, sandboxing, or language-based isolation technologies like WebAssembly [142] or Javascript Isolates [50]. It is implemented in a number of academic and production systems [304, 10, 13, 26]. In this case, the user software may require additional porting efforts—either recompilation to WebAssembly or porting to the platform owner-provided libraries. In return, such functions achieve extremely high density and low startup speeds [269, 127]. On the other hand, this approach limits the selection of software that can be run as a serverless function, and the security of WebAssembly-based solutions requires further research [199].

As Apache Openwhisk, which Clemmys uses as a foundation, relies on container-based runtime images, Clemmys follows this approach. However, our contributions are generalizable to VM-based approaches, and partly to sandboxing-based serverless systems as well. Using TEEs with sandbox-based approaches are already well-studied in the literature [65, 90].

Serverless Computing Platforms. To design a usable and generic defense strategy for common serverless platforms, we have analyzed similarities among them. For this, we have studied architectures of several open serverless platforms: OpenLambda, Iron.io, OpenWhisk, Fission.io.

We can see that the architecture of these frameworks implements either of the two variants shown in Figure 6.1(a) and 6.1(b). The architectures contain a gateway node, a controller node, and multiple worker nodes. The architecture also contains service nodes such as a database system to store platform configuration, billing services, autoscaling services, message queues, and so on. We have omitted them from the figures.

There are three main node types in serverless systems: workers nodes, controller nodes, and gateway nodes.

The *worker* nodes execute functions with specified inputs and resource limits in response to network events. In a production deployment, there will be numerous worker nodes, necessary to provide the advertised level of scaling. They spawn the function in the runtime environment in response to the incoming request, process it, and return the results.

The *controller* node is the central management component of the platform. It manages the available functions images, resource limits, user accounts, billing, and permissions. Typically there is a single controller instance connected to a database for metadata storage. The controller node may be physically replicated even if it logically unique.

To secure the connection to the Internet, serverless platforms use a *gateway* service that terminates the TLS connections from the client and acts as a load balancer to the workers. It should be noted that some systems, like Apache OpenWhisk, use the controller as a load balancer.

A connection between the gateway and a function can be either direct or indirect. For example, a design may include a message bus for reliability, so that messages that were admitted to the system are guaranteed to produce a result.

A common pattern in serverless computing is *chaining*—composition of functions into sequences where data is passed from function to function without the involvement of the user. The functions in the chain can run either on different nodes or on a single node (for data locality). As chaining is one of the cornerstone features of FaaS, it is necessary to take it into account during the system design.

Bottlenecks of Intel SGX and serverless platforms. Intel SGX has several performance limitations that influence the design of the system. Some sources overheads, like enclave context switches and EPC paging, are generic and were explained in §2. Other limitations merit a more detailed consideration in the context of general bottlenecks of serverless systems.

Previous research has identified numerous issues that reduced the attractiveness of serverless computing to the users and operators: slow communication between the functions, high latency, and I/O operation throttling for the platform-provided persistent storage, programmability restrictions, and unpredictable performance variations [169]. Some of these issues depend only on the architecture of the serverless system and can be solved generically (e.g. support for function-to-function communication would speed up use-cases that are currently forced to communicate through object stores). Other restrictions, like programmability, would arguably require rethinking the interfaces that the platform operator exposes to the developers, as well as significantly modifying the corresponding libraries.

However, some restrictions, like limited I/O performance and the unpredictable performance, especially related to the slow function startup time, must be taken into account when designing an Intel SGX-based runtime for the functions. The reason for this is that Intel SGX exacerbates them by its performance overheads. For example, the throughput of communication with the local store would be 8 times higher with Intel SGX (see §2.5), thus rendering common optimizations like colocation of code and data close to useless. Clemmys solves this problem by building on SCONE, which was explicitly designed to provide efficient I/O for enclaves.

Another issue that the developer must solve is that the SGX enclave heap size adversely affects enclave startup time: multi-gigabyte heaps, such as required by the runtimes of dynamic programming languages, can take significant time to initialize: our measurements show that initializing a 4Gb heap can take up to 35 seconds. One of the main challenges of our work was to overcome these limitations to successfully apply SGX in the FaaS domain.

We make use of the recently released Intel CPUs that support the second generation of SGX (SGXv2) to solve speed up the function startup time. More specifically, SGXv2 extends SGX with the Enclave Dynamic Memory Management technology [217], which allows adding (*augmenting*) EPC pages to the enclave, modifying the EPC page metadata (for example protection flags), and removing (*trimming*) EPC physical page mappings from the enclave after

it started running. By adding support of SGXv2 to SCONE, we allow SGX-enabled functions implemented in, for example, Python, to start up with the latency which is on par with the non-protected Python functions.

Palaemon. Palaemon [138] is a key management service (KMS) implemented as a part of the SCONE remote attestation and configuration system, which uses an attestation scheme similar to that of ShieldBox [284, 164]. It supports standard KMS features, such as a flexible policy language for specifying secrets and entities that may access them and automatic generation of secrets. Most importantly, it supports provisioning secrets and shielding layer keys to SCONE-based applications.

Palaemon is implemented to run inside an Intel SGX enclave alongside with the applications it attests; Palaemon itself is attested using the Intel Attestation Service (IAS) [164]. It opens a possibility to operate Palaemon as a turn-key solution, that is using a single instance per data center.

Palaemon consists of two components: the Local Attestation Service (LAS) and the Configuration and Attestation Service (CAS). The LAS is running on the same node as the attested application, and issues SGX local attestation quotes to the CAS. The LAS itself is attested using IAS. The CAS is the service that securely issues the configuration to the correctly attested applications.

The CAS supports a wide range of functionality:

- generation of secrets without disclosing them to operators;
- policy board-based configuration management;
- enclave-based access control: only enclaves with correct measurement are configured;
- provides the configuration to the SCONE shielded filesystem;
- sets the environment variables and CLI arguments for the application;

In our work, this configuration comprises function chain configuration, cryptographic keys, and function-specific information.

6.3 Threat Model

We consider a typical scenario of a FaaS platform operation. A *user* acquires a function source from a *function provider*. The function is deployed on a cloud platform managed by an *operator*, who has access to the host OS on all nodes. A malicious *external attacker* may try to exploit a vulnerability anywhere in the function or in the cloud stack to gain access to the function source or data. This scenario is the foundation of our threat model:

- The operator should not be able to compromise the confidentiality and integrity of the function source and data.
- The function provider should not be able to compromise the confidentiality and integrity of the data processed by the function.
- Only the user should be able to define the functions which constitute a function chain. Other parties should not be able to insert, remove, or shuffle functions inside a chain specified by the user.

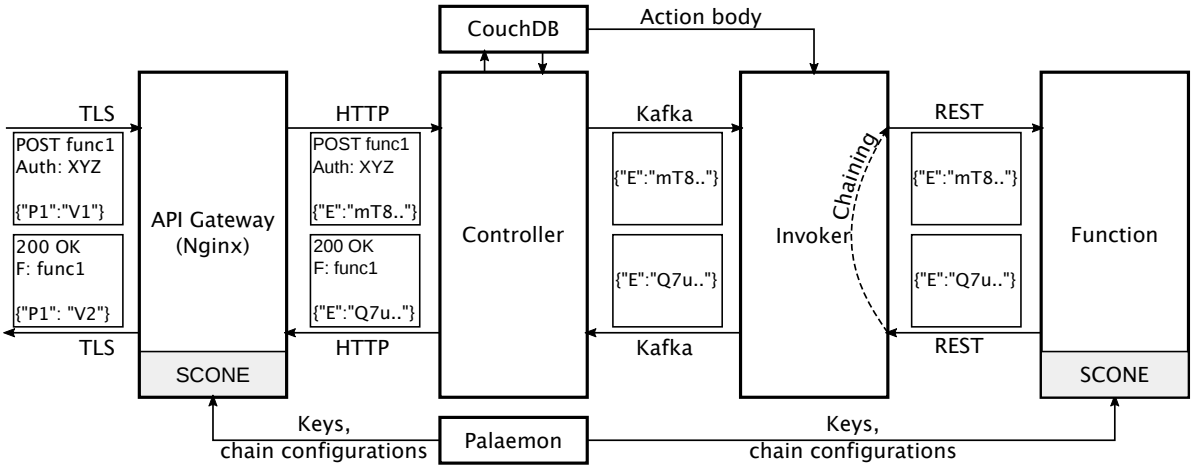

Figure 6.2: System architecture of Clemmys and transformations of a user request as it passes through the system.

Specifically, we target the following attack vectors:

AV1. The operator or the external attacker inspects the memory of the functions or the system components. This way, they can extract session keys and decrypt the traffic, or directly extract plaintext user data from the process memory.

AV2. The operator reads and modifies the traffic between the function and its users. This is possible because messages between the gateway and the functions are unencrypted. The external attacker could also intercept the traffic if she compromises a part of the cloud stack.

AV3. The operator modifies the execution order of functions in a chain to create an information disclosure. For example, the message content that is supposed to be sent to the user can be redirected to a logger that writes or sends plaintext messages over the network.

In our threat model, we do not consider microarchitectural attacks or memory safety [231] attacks. We assume that approaches like Cloak [139], Varys [233], SGXBounds [190], or other compiler-based approaches [270] can efficiently thwart them.

We also consider application vulnerabilities as orthogonal to our work. To prevent leaks of information due to neglect or malicious actions of the function provider, the user must manually inspect the function code. While theoretically, a form of static analysis or sandboxing can prevent such leaks, we do not tackle this problem.

As Intel SGX leaves system resource allocation to the control of privileged software, Clemmys does not guarantee availability: the operator may choose not to run any of the system components or stop them at arbitrary moments of time. However, denying service is against the operator's incentives, as it constitutes a violation of SLA.

6.4 Design

In this section, we discuss how we can secure common FaaS platforms from the attacks outlined in the threat model (§6.3). We defend against AV1 by running each function inside Intel

SGX enclaves. We prevent AV2 by encrypting traffic between the function and the gateway. To tackle AV3, we introduce a protocol that cryptographically ensures the correct order of functions execution in a function chain.

Combined, Clemmys comprises the following generic system architecture (see Figure 6.2). First, the user initiates a mutually authenticated TLS connection to the gateway. Then, every incoming HTTP request passes through the gateway which terminates the TLS connection, reencrypts the message body into an internal message format, and passes it to the FaaS controller. The controller inspects the request metadata (not modified by the gateway), and passes it to the appropriate function on the target machine, possibly via a message queue. On the target machine, the container is started to process the message. As an optimization, the platform could reuse a container from a previous request. The function starts inside a SCONE SGX enclave and performs remote attestation and configuration using Palaemon. Then, the function verifies that it is executing at the right stage of the chain using the information from Palaemon and from the message, decrypts, and processes it. When the result is ready, the function encrypts it and passes it either to the controller or the next function, depending on the current stage of the chain. When the final result is computed, the gateway decrypts it and writes it down to the client TLS connection.

In the following chapters, we will describe each of the design decisions in detail.

6.4.1 Preventing Memory Inspection

To understand how Clemmys prevents AV1, it is necessary to consider the components of OpenWhisk in more detail. As the user request is submitted over TLS to the *API Gateway*, the gateway forwards the request to the *Controller*, while terminating the TLS connection. The *Controller* inspects the request to determine which function or function chain must be spawned, and sends the corresponding commands along with the original user request to the worker node, where the *Invoker* is running. The Invoker uses the registry of Docker images with the runtimes and user functions, and spawns the containers as requested by the Controller. The Invoker also handles function chaining, by spawning the next function in the chain after the previous has stopped running. It also allows reusing the previous container from the same function if this is possible (e.g., if the function has been recently running on the same node).

Clemmys targets AV1 by employing SGX enclaves that hide the memory contents from the adversaries and ensure its integrity and confidentiality. Of all the abovementioned system components, only the API Gateway and the functions have to run in SGX containers, as these components are the only ones that interact with raw user data: the rest of the system deals with benign metadata. This metadata contains the name of the function to spawn, user authentication to the system, and so on. Such metadata must be inspectable by the provider's software for scheduling the function execution, setting up resource limits for the runtime environment, billing, and so on. Most importantly, this inspection does not violate the confidentiality of the user request. To protect the message content, the user-provided arguments to the function chain are encrypted. We describe the encryption in more detail in the next chapters (§6.4.3).

6.4.2 Preventing Traffic Analysis and Modification

We prevent AV2 by introducing encryption between the gateway and the functions. Observing that a message queue may be used inside the system for reliability, we conclude that TLS should not be employed as the encryption mechanism.

Message queues are typically used in cloud settings to decouple services and to gain message persistence, and thus should be taken into account when designing the system. The strawman solution is to run all system components[1] inside Intel SGX enclaves and extend the components to use TLS for communication. However, this solution comes at a significant cost in case a serverless platform has memory-intensive components, for example Kafka. To secure these components, it would be necessary to use SGX enclaves, where this software would experience a slowdown due to EPC paging (see §2.5.

Another alternative is to encrypt the messages at the first client-facing node into a format that can be passed through the system transparently, while keeping the sensitive user data confidentiality- and integrity-protected. We explain this approach in §6.4.3, where we tackle AV2 and AV3 together.

6.4.3 Verifying Function Execution Order

We target AV3 by designing a protocol that cryptographically ensures that the functions are invoked in the order specified by the developer. Our key observation is that the user data in the messages are not read by the intermediate nodes in any way; instead, the nodes rely on metadata transmitted in, for example, HTTP headers to schedule the execution of functions. Thus, it is possible to encrypt the message data and add extra information to it without any effects on the system operation.

To facilitate secure function chaining, the protocol must allow functions to detect and to cryptographically verify the following violations of function chaining:

- A function is dropped from the chain.
- A function is inserted into the chain.
- The functions are executed in the wrong order.

We recognize that the message of the protocol should contain the plaintext metadata (user certificate fingerprint and function chain name) to query the decryption key from Palaemon. Additionally, the format must include the information necessary to identify if the processing happened in the right order. To achieve this goal, we store the chain as a list of functions inside Palaemon, and include the index of the functions in the chain in the protocol message.

Thus, given a user message M, the resulting protocol message has the following fields:

$$BASE_{64}(C, CN, N, IV, AESGCM(M, IV, \langle C, CN, N\rangle)_K) \tag{6.1}$$

where:

- C—the fingerprint of user certificate;
- CN—name of the function chain (carries no semantic information for the function);

[1] OpenWhisk-provided components were designed to run inside of the trusted network and thus lack provisions for message confidentiality after the API Gateway.

- N—index of the current function in the chain;
- $AESGCM(M, IV, B)_K$—AES-GCM encryption of plain-text message M using key K, initialization vector IV, and associated data B;

The function can use the certificate fingerprint C, chain name CN, and index N to detect the violations as follows. First, it performs remote attestation, and gets lists of functions in the chain and AES-GCM key for each C, CN pair from Palaemon. Then, it uses the fingerprint and function chain name to select the correct AES-GCM key, and verifies that the attacker did not modify the abovementioned fields. The action uses the function name and index fields to ensure that the processing in chains happens in the specified order. OpenWhisk action looks up a function with index N in the chain CN, and checks if it matches the identity of the action currently executing. If there is a match, the execution of the function chain is correct. The index field N is incremented as the function finishes execution and passes the message to the next function for processing.

The user request in OpenWhisk is submitted in JSON format, and the API gateway fully includes the request into the encrypted message (i.e., all service metadata is included as well). The output of the API Gateway and the functions is also in JSON format, where the output of the function or original user request is encrypted according to Equation 6.1. This cyphertext is added to the output message in a dictionary, as the value for the "`enc`" key.

Function Identity. To allow chain verification, an action running with SGX must be able to learn and verify its name (i.e. *identity*) CN. In the simple case when the identity corresponds to a single binary, we can ensure its validity using information from SCONE and Palaemon. When SCONE builds an enclave, it also calculates a cryptographic checksum of its initial image, called enclave measurement, verified during the remote attestation. Palaemon can send an attested enclave a secret that depends on the enclave measurement. In our case, Palaemon sends the enclave the intended identity for this measurement after the remote attestation.

However, actions implemented in interpreted programming languages require additional care. As the attestation verifies only the interpreter and the libraries, the interpreted application source is not included in the attestation report. Thus, for such functions, additional measures are necessary to bind the function identity to the executing memory image. For Python, we suggest using SCONE file system shield with integrity and confidentiality protection [71]. In this scenario, Palaemon performs remote attestation of the Python interpreter and sends the keys to the enclave for decrypting the shielded file system image. Then, Python can read the identity of the function from the file in the shielded file system. Support for this identity verification requires changes to the Python interpreter, to prevent it from loading executable scripts from unprotected file systems.

Key and Configuration Management

The protocol message outlined in §6.4.3 uses symmetric AES-GCM encryption. The keys used for the encryption are generated and stored in an external key management service—Palaemon—with a unique key for every user-chain pair. It allows Clemmys to isolate requests from different clients and prohibits the adversaries from moving a message from one chain to another.

Palaemon also stores the information necessary for the chain integrity protocol:

- Lists of functions inside each chain in the system.

- Bindings of function names to enclave measurements (for native functions, C/C++/Rust).
- Decryption keys to function sources stored in a protected file system (for interpreted programming languages, Python/Node.js).

The protected components (gateway and the functions) fetch the keys and other configuration data at the startup of the corresponding program, after the remote attestation. Thus, an extra round trip is required to start the application. Its effect, however, is moderate as all communication with Palaemon happens locally, unlike during the Intel attestation procedure.

Clemmys relies on client certificates for client authentication, instead of the authentication system available in the FaaS platform. Clients generate a private key and use it to receive a valid certificate signed by the Clemmys key. The gateway verifies the client certificates and rejects connections with those clients that do not present one. The fingerprint of the presented certificate is used in the protocol message to identify the client. We make this decision because the FaaS controller, which normally implement authentication in OpenWhisk, is run without SGX protection in our case, and therefore can be easily attacked by a privileged adversary. In our design, Palaemon acts as an authenticating authority instead.

6.5 Implementation

We rely on Apache Openwhisk to implement Clemmys. To keep the performance impact of the protection low, we strive to restrict the changes to as few components as possible.

API Gateway. In the original OpenWhisk design, API Gateway terminates TLS connections and manages functions triggered over REST. The original API Gateway is implemented using OpenResty (Nginx distribution with Luajit and numerous Lua extensions) [37]. Therefore, to extend the API Gateway with the message reencryption functionality (as explained in §6.4.3), we have implemented a dedicated Nginx plugin.

The core functionality of our plugin is:

- Maintaining key and chain information after startup and remote attestation.
- Performing message reencryption.
- Performing security checks on the messages and client connections.

The plugin implements an Nginx rewrite phase handler to encrypt the request before passing it to the OpenWhisk Controller, and uses the header and body filter to receive the reply, verify its correctness, decrypt it, and pass it to the client (1230 lines of C code in total). We use the Nginx configuration file to specify the REST endpoints for which the plugin must be active.

At the plugin initialization, it scans through the process environment variables supplied by Palaemon, and populates the configuration tables, which are later used to process user requests. So far, our Nginx plugin does not support dynamic updates to the Palaemon-provided configuration: to receive new chain configurations and keys, the API Gateway has to be restarted. Alternatively, operators can use systems like HAProxy [18] to perform a zero-downtime configuration update. We plan to alleviate this limitation of Palaemon and the Nginx plugin in the future.

Function image skeleton. We implement our skeleton images for native SGX functions and Python SGX functions. The native SGX image adds a configuration file for SCONE asynchronous system call interface and configures the environment variables to set default enclave heap size. The SGX Python image additionally replaces the stock Python from the Alpine Linux repository with SCONE-build Python and installs a set of predefined Python packages using pip utility and SCONE cross-compiler.

Controller. We modify the Controller to put the action name in the header of replies to the "activation get" command, which is used to retrieve the results of asynchronous function invocation. Before our change, neither client nor Controller passed the function name to the API Gateway for this command, and it was possible to retrieve the function name only by parsing the user response. This change allows us to avoid costly parsing functionality inside of the Nginx plugin, that we would have to otherwise implement to secure this REST endpoint.

Invoker. Invoker is a service running on the worker node that communicates with the Kafka queue and launches containers with functions to serve the user requests. The input from the user is provided to the function via standard input.

We also modify the Invoker to pass additional SGX and SCONE-specific parameters to the Docker. Our modified Invoker adds `/dev/isgx` device to the function container. Also, the Invoker configures additional container resource limits required by SCONE, most notably, allowing the spawning of the threads running under the realtime scheduling policy.

6.5.1 Function Startup Optimization

The requirement of not degrading the system latency clashes with the reality of running dynamic language runtimes inside SGX enclaves. These runtimes typically run with huge heaps, which increases the startup time of SGXv1 to the range of seconds, or even tens of seconds (Table 6.1).

The enclave loading procedure has several steps: creating control structures, adding and measuring EPC pages, and launching the enclave using the EINITTOKEN structure. Based on our observations, the main bottleneck is in adding pages: Even though heap pages are not measured, adding them to the enclave still takes significant amounts of time. To reduce the attack surface, the SCONE mmap implementation also zeroes the added pages, which slows down the application startup when large chunks of memory are allocated.

To mitigate the impact of this issue, we rely on SGXv2 EDMM [217] features to reduce the startup time. We use the SGXv2 EPC augmentation feature to skip adding most of the pages during enclave creation, which can take a significant amount of time during the enclave startup. Instead, we allocate only a small number of heap pages at the beginning of the heap region—20 MB by default—and around 40 pages (164 kB) at the end of the region, where the metadata (bitmap) for SCONE mmap allocator is located. SCONE loader skips adding all other pages. These changes sum up to significant savings: the SGX loader issues an `ioctl` to the driver for each page that needs to be added, which copies the full page contents into the kernel before adding them to the enclave. Avoiding this work brings a significant improvement to the startup time. Batching cannot reduce this cost, because it is mostly caused by EPC metadata updates, not `ioctl` calls.

On top of the SGXv2 support, we implement two additional optimizations: batch EPC augmentation and zeroing pages upon deallocations instead of allocations.

Because most of the accesses during startup cause augmenting enclave exits, we modify the driver and SCONE runtime to do batch augmentation of enclave memory. On the EPC augmentation event for page with address *Addr*, we additionally augment all pages in range $[B\lfloor\frac{Addr}{B}\rfloor, B\lceil\frac{Addr+1}{B}\rceil)$, where B is batch size. The runtime, in this case, would calculate the same range of pages, and accepts it using the EACCEPT instruction. We allow users to configure the amount of batching, as this optimization may not have a large effect on most applications. While this optimization moves the cost of adding pages to time after the application start, it distributes the faulting memory accesses over a long time, reducing the EPC paging rate.

We also modify SCONE to perform page zeroing on page deallocation, instead of page allocation. Deallocated pages are kept within EPC, and thus, the OS cannot modify their contents in-between allocations. The benefits of this design are twofold. First, the fresh pages added to the enclave via the augmentation mechanism are automatically zeroed by the hardware. Thus, enclaves can use these pages without zeroing them. Second, the application may optimize its run time by exiting without zeroing its memory.

In general, we expect a much lower effect from the latter two optimizations than from just switching to SGXv2. Our optimizations are orthogonal to those published in recent studies [62, 230, 113] and can be applied independently.

6.6 Evaluation

In this section, we answer the following questions:

- How does Clemmys perform compared to native OpenWhisk in terms of the best achievable throughput and latency?
- What is the performance impact of the function chain integrity protocol?
- What is the impact of our optimizations on the function startup latency?

Applications. We base our evaluation on the Fex [232] evaluation framework, with six computationally-intensive applications from PARSEC [87] and Phoenix [251] benchmark suites as workloads. We use the largest inputs that do not cause intensive EPC paging.

Methodology. The reported results are averaged over 10 runs and "mean" is a geomean across all the benchmarks.

Testbed. We ran all the experiments on two machines with different generations of the SGX technology. The machine with SGXv1 has a 4-core (8 hyperthreads) Intel Xeon CPU operating at 3.6 GHz (Skylake microarchitecture) with 32 KB L1 and 256 KB L2 private caches, an 8 MB L3 shared cache, and 64 GB of RAM. The machine with SGXv2 is a NUC with a 4-core (4 hyperthreads) Intel Pentium Silver CPU operating at 1.5 GHz (Apollo Lake microarchitecture) with 16 KB L1 and 128 KB L2 private caches, an 4 MB L3 shared cache, and 32 GB of RAM. We have used this machine because it was the only released SGXv2-enabled machine at the time this work was performed.

6.6.1 Security Evaluation

We begin with a security evaluation of the protocol outlined in the §6.4.3. To this end, we run a chain of two echo functions implemented in Python with different identities. When the message processing order corresponds to the specified processing order in the chain,

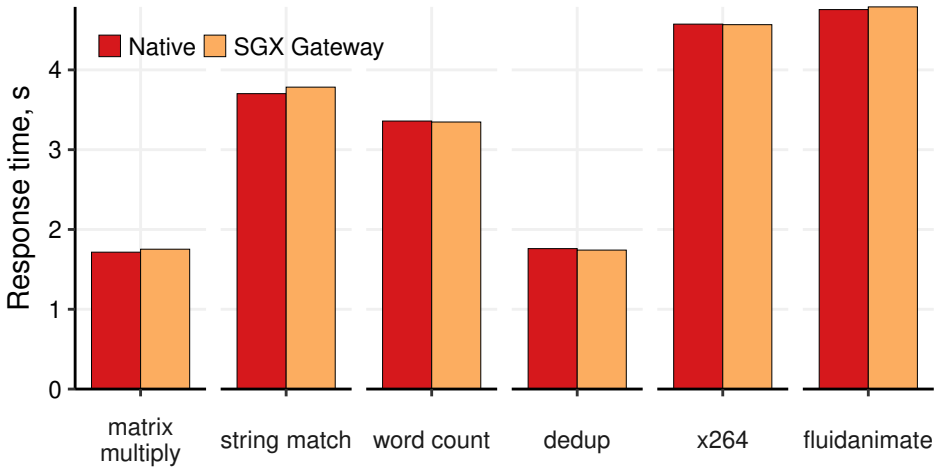

Figure 6.3: Response time with a protected API Gateway compared to native API Gateway, for different functions from a Parsec benchmarking suite. The functions are non-protected to highlight the impact of the Gateway.

the functions succeed. Then, we modify several components of the message and test if the modified message is accepted. The results were as follows.

- Changes to certificate fingerprint, chain name, and counter are detected by AES-GCM tag comparison.
- Sending the message to a wrong function in a chain causes function failure due to a mismatch between function identity and the intended function in the chain.

Finally, to simulate a rollback attack, we save the message after it has been processed by the first function, and separately invoke the second function with the same message. We see that the second function accepts and processes this message without signaling an error, even though this message has been already processed before. Thus, we can see that Clemmys offers no defense against rollback attacks. This attack vector targets only stateful functions, which must counter it at the application level.

6.6.2 Response time

Next, we measure the impact of Clemmys on the overall system response time. Because Clemmys hardens two components of OpenWhisk—API Gateway and the functions—we perform two experiments to evaluate their impacts separately.

In the first experiment, we evaluate the impact of the Gateway in isolation by running native functions with two configurations of the Gateway: native (as in OpenWhisk) and protected (with our Nginx plugin, see §6.5). In the second experiment, we isolate the impact of protecting the functions by running the native and SGX-protected functions separately, without a gateway. In both cases, we use SGXv2 machine as the function startup time on it is independent from the heap size (see §6.6.3 for details).

The response time in the second experiment is measured for different levels of oversubscription, from 1 to 16 instances of a function running in parallel, showing how many enclaves running in parallel the system can sustain on a single worker node. This information

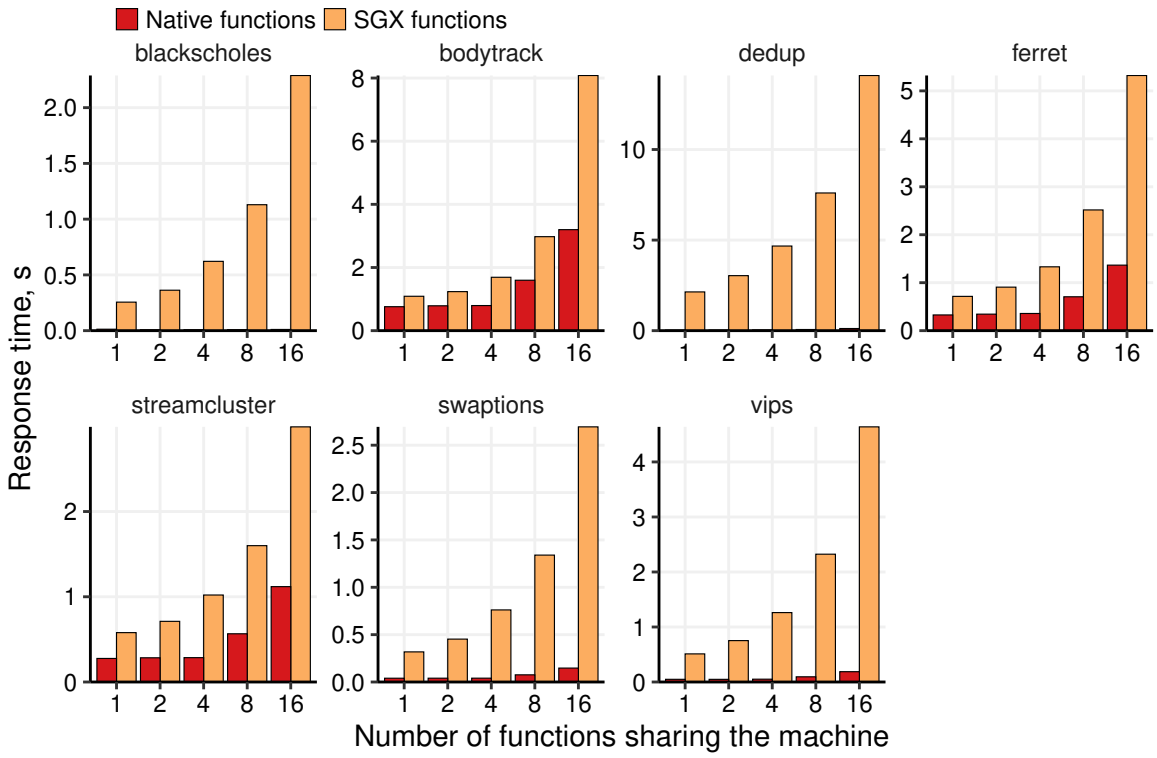

Figure 6.4: Response time with Clemmys-protected functions compared to native functions, for different numbers of functions sharing a machine. The API Gateway overheads are not included (worst-case overhead).

allows operators to assess the requirements for their hardware platform. As native functions achieve oversubscriptions at much higher concurrency levels (in the out-of-memory situations), we skip oversubscription in the first experiment.

Figure 6.3 represents the overheads of the API Gateway. It can be seen that the cost of protection at API gateway is minimal and is amortized by the overhead of the OpenWhisk itself.

However, the evaluation of function impacts on Figure 6.4 has shown bigger overheads, where all SCONE functions display an increase of latency for both short-running and long-running functions, which linearly increases with the contention level and is independent of function benchmarked. We perform additional performance analysis, which has shown that this slowdown happens due to contention on the EPC memory.

Our investigation has shown that this is caused by several reasons:

- There is a minimum amount of time necessary to create the enclave, even if only 20Mb of memory are initialized during the enclave creation. This extra initialization time drives the overheads at low oversubscription levels.
- At higher oversubscription levels, beyond 2 enclaves, there is extra overhead caused by oversubscription of *cores*: enclave threads and system call threads must execute in parallel to make forward progress, while the CPU on the benchmarking machine has only 4 cores.
- As the number of functions increases to 8 and 16, an extra factor of *EPC oversubscription*

Heap Size	SGXv1 startup time	SGXv2 startup time
4 Gb	35 s	0.37 s
2 Gb	15.6 s	0.37 s
128 Mb	0.84 s	0.37 s

Table 6.1: Startup time of a no-op enclave depending on the dedicated heap size for SGXv1 and SGXv2.

g

appears: during the enclave creation, the EPC is exhausted, and EPC paging starts to affect the enclave creation process.

To alleviate these issues, we propose the following measures:

- To reduce the amount of memory used during the enclave creation, it is possible to use profiling: record which pages are required by the enclave during runtime, and add only them to the enclave during the start-up. Alternatively, the minimum initialized heap size can be decreased to a few Mb.
- To alleviate the core oversubscription, a centralized server can be used during the enclave creation for core allocation, and the SCONE runtime can automatically switch to synchronous system call processing mode in case core oversubscription is detected.

However, we leave the implementation of the performance improvements to future work. We further discuss these results in §6.7. It should be further noted that Parsec functions are relatively short-running, and real-world applications can be expected to at least partly amortize the startup cost.

6.6.3 Function startup optimizations

To evaluate the impact of our SGXv2-based optimizations on the function startup time, we do measurements on the SGXv2 machine. The SCONE configuration uses a single enclave and a system call thread, with a 4Gb heap. We chose this particular heap size because it exemplifies a CPU-bounded application running with a dynamic programming language runtime.

We evaluate several versions of the runtime:

- SGXv1—no optimizations;
- SGXv2—SGXv2 version with EPC augmentation batch of 20 pages;
- SGXv2 (NB)—SGXv2 version with the batching disabled (No Batching);
- SGXv2 (NB, NO)—SGXv2 version without batching and without optimized mmap allocator (No Batching, No Optimizations).

Because of the greatly varying runtimes of experiments, we normalize the results to the runtime of SGXv1 version. We skip the raytrace benchmark in all cases as it required X11 libraries to build.

We show the results of these experiments in Figure 6.5. We can see that in all cases there is a significant speedup from using SGXv2. On average, applications have ~20× lower latency on Parsec benchmarks, and 10× lower latency on Phoenix benchmarks.

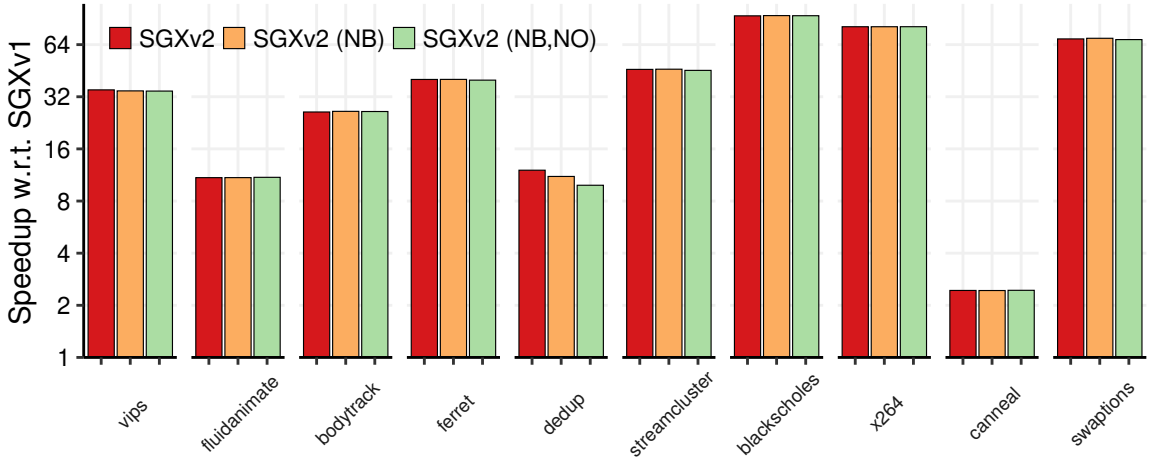

Figure 6.5: Parsec benchmarks results for our SGXv2-based optimizations.

To explain these results, we conduct an additional experiment (Table 6.1), where we measure the application startup time of a no-op C application while varying heap sizes with SGXv1 and SGXv2. We discover that enclave startup time depends linearly on the heap size in SGXv1 case. With larger heap sizes initialization time can reach 35 seconds (for 4GB heap), and it dominates the total application runtime. For comparison, the initialization time for 4 GB heap with SGXv2 is ~0.37 seconds. Thus, the results in Figure 6.5 do not mean that the applications were running faster, only that the cost of enclave initialization became greatly reduced.

The impact of additional optimizations is limited compared to just switching to SGXv2. On short-running Phoenix benchmarks (figure not shown), these optimizations do not have any influence. After investigation of PARSEC benchmarks, we have discovered that only *dedup* and *facesim* benchmarks had working sets larger than the EPC size. All other benchmarks dynamically allocate less than 30 Mb of memory, experience no EPC paging, and get no benefit from the proposed optimizations. The impact of the optimizations is smaller on the *dedup* benchmark: after the initial setup phase, *facesim* allocates memory without freeing it, leading to highly local memory use. On the other hand, *dedup* has approximately 1 free call per 2 allocations, reducing the memory allocation locality. Thus, we conclude that our optimizations are beneficial only for functions that constantly allocate memory, and have no influence on most functions.

6.6.4 Impact of API Gateway

We evaluate the API Gateway of our system to answer the following question: *At which point does the API Gateway become a bottleneck in our system compared to the native version?* To answer this question, we perform benchmarking using our modified API Gateway running inside SGX enclave, and a native version of API Gateway. The reencryption plugin is disabled in the native version, otherwise the configuration file used is the same in both versions. We do not perform any advanced Nginx configuration tuning. We evaluate Clemmys in two scenarios:

- With dummy OpenWhisk functions: We run stock OpenWhisk with a minimal function after the API Gateway. The function is implemented in C and returns a hardcoded correct message.

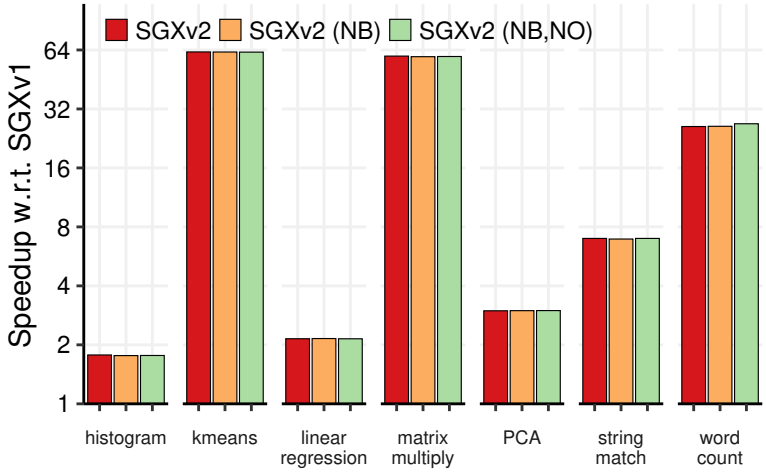

Figure 6.6: Phoenix benchmarks results for our SGXv2-based optimizations.

- Without functions: the Gateway passes the request to the Nginx upstream that replies to all requests with the same hardcoded message (OpenWhisk is not used at all).

We run the experiments on SGXv1 machine. We skip the second scenario with a native API Gateway, as it corresponds to normal file serving over HTTPS. On our machine, the saturation point for a single core is at 6000 requests/s.

The results are shown in Figure 6.7. We can see that without functions, the SGX-based API Gateways scales up to 300 requests/s. With the OpenWhisk functions, we see that both native and SGX versions saturate at around 225 requests/s, suggesting a bottleneck in the OpenWhisk components different from the API Gateway. The increased latency at 25 requests per second is caused by the increased rate of container spawning, which increases system load; but this rate is insufficient to always cause container reuse: 14% of all containers at this rate are freshly started, vs. less than 1% at 100 requests/s. On the other hand, at the higher request rates, starting at 200 requests/s, the system is running above its capacity and queuing excessive messages in the Kafka queue; the fresh container ratio at this rate is approximately 2%, and the (median, 95th percentile) latency at Invoker is (0,045s, 0.06s).

6.7 Discussion

As we mentioned in §6.1, typical FaaS applications are CPU-intensive, run for several seconds, and require a scalability guarantee. Applications that are I/O bound or have very short runtime would have huge overheads from running inside serverless platforms, as these platforms are designed for very different workloads. Current research shows that OpenWhisk can add up to a second of latency [65]; accordingly, as our evaluation of Clemmys shows, the OpenWhisk overhead completely masks the overheads of SGX. The results, however, are not representative of the workloads with very long runtimes, or if OpenWhisk system latency improves. In this case, the overhead of SGX would not be masked by the OpenWhisk overhead, as the latter does not depend on the workload. Previous research shows that SGX can cause applications to become up to 100 times slower during EPC paging; even without resource starvation, SGX application may see 8-25% SGX overhead due to memory encryption and

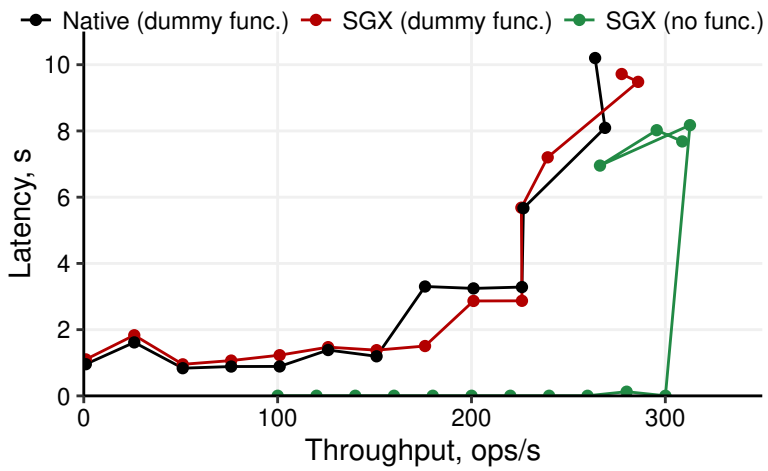

Figure 6.7: System latency (95th percentile) with SGX and native API Gateway with and without OpenWhisk functions.

interrupts (§3.4.2). These overheads can become even greater if side-channel or Spectre attack mitigations are deployed.

Currently, Clemmys has some limitations. We have already mentioned in §6.5 the inability to update the secure configuration after application startup. Also, OpenWhisk and Palaemon management systems are not integrated. Thus, when an operator creates a new function chain in the OpenWhisk, it is not automatically added to the Palaemon, and will not run until she manually updates the chain using the Palaemon API. The same is true for defining keys, signing user certificates, and so on. These inconsistencies can lead to denial of service when, for example, function chain configurations are separately changed in Palaemon and OpenWhisk.

OpenWhisk supports several action triggers, including periodic activations and activations based on, for example, Kafka streams. In our work, we have focused on the actions triggered via REST interfaces. In the future, Clemmys can be extended to support these triggers as well.

The function chaining protocol so far supports only linear processing chains, without branching and loops. In the case of more advanced chaining graphs [77], the format of the function chaining protocol would have to be redesigned. The format is not OpenWhisk-specific and can be reused with other FaaS platforms.

Clemmys does not depend on SCONE conceptually: the same functionality can be implemented with Graphene-SGX and Intel SGX SDK.

Security discussion. In this section, we argue why the security design of Clemmys is sound. First, we observe the communication in the system: at each hop between system nodes the communication is encrypted, either with TLS or with Clemmys function chaining protocol. Thus, neither eavesdropping nor message modification can cause a violation of security guarantees.

Likewise, each node that handles the plaintext data from the user is protected inside an Intel SGX enclave. All nodes that are running without SGX get access to the user data only in the cyphertext form. Thus, observation and modification of the data is either not possible (access denied by Intel SGX), or cannot affect the data confidentiality and integrity, as neither message plaintext nor keys are available. To guarantee integrity and confidentiality of the

data from third parties (e.g. Amazon S3), it must be accessed only via TLS.

With these defenses, the attacker is still able to perform two attacks: send the message to the wrong function in a chain, and send an outdated message to the same function chain (rollback attack).

We have shown in the evaluation (§6.6) that our protocol protects the system from the first attack: when the wrong function receives the cyphertext message, it will verify that the destination index in the chain of the message matches the identity of the current function. If this is not the case, the function will reject the message. Thus this attack should not be possible. As far as the rollback attack is concerned, it is still possible and should be countered at the application level.

Performance. In Section 6.6.2, we have shown three issues, that are limiting the performance of SGX-enabled functions on the worker node: CPU contention, EPC paging, and relatively high latency of function startup. We have concluded that the EPC paging is a major limiting factor for co-location of multiple independent enclaves on the same machine.

One of the approaches to alleviate this problem is switching to the language-based or sandbox-based function isolation model, which allows putting all functions inside a single enclave. This would improve the startup performance of functions even further, while sharing the trusted runtime memory, and of system call threads would use the limited EPC resources more sparingly, allowing to run more functions on a single node.

However, we would like to argue that switching the function isolation model eliminates only one of the issues (function startup latency), but does not solve the fundamental issue: the scarceness of the EPC memory, which would limit the performance of most FaaS framework even for the applications with moderate memory usage through EPC paging. Indeed, the RAM consumption of tasks like video processing or neural network processing is much bigger than the EPC even for a single function: around 190-220 Mb [243, 187].

Thus, the possibilities for concurrent execution of multiple enclaves on a single node will remain limited until the amount of EPC memory is increased. In this context, increased EPC sizes in recent Intel CPUs and switch to the MKTME-based encryption in Ice Lake-generation CPUs are key factors that may make Intel SGX a useful technology for protection of serverless function. Another solution comes from the features of serverless platforms themselves: as the cloud user has no control over the placement of functions, it is possible to distribute execution of functions over different nodes, reducing their contention over EPC memory.

Potential future extensions. One of the approaches for improving the startup latency of SGX-based Faas platforms is the use of zygotes—a pre-initialized "template" application, that is used for quickly starting its instances by forking. Zygotes have been used to improve the application startup times of Android applications, and more importantly, of the container- and VM-based serverless functions [230, 113, 88]. Adapting this approach to SGX can be done by using a `fork` system call after initialization.

Another important extension may be required by serverless users that want to keep the binary code of their application confidential. In this case, a functionality similar to that of the Protected Code Loader in the Intel SGX SDK [16] must be implemented.

6.8 Related Work

Secure FaaS platforms. Researchers and the industry have proposed several FaaS platforms. The first production system built with serverless architecture is AWS Lambda [5];

soon, several other open-source [3, 36, 27, 14] and commercial [17, 20, 7] platforms appeared. Yet, none of them protects against privileged attackers.

The work that most closely resembles ours is S-FaaS [65], a trustworthy accountable FaaS system. Like Clemmys, S-FaaS is built on Apache OpenWhisk, and uses transitive attestation, a scheme similar to that of Palaemon: a special Key Distribution Enclave (KDE) attests the worker enclaves and distributes the keys to them. These keys allow enclaves at worker nodes to establish a secure channel to clients, to produce the signed statement about the inputs and outputs to each function, and to report the function resource usage. As each function in the chain can perform the same attestation as the KDE, S-FaaS naturally supports function chaining. The client needs to attest the identities of the key distribution enclave and all worker enclaves. S-FaaS functions append function- and invocation-specific tags to the resource usage report to allow detection changes to the intended order of functions in a chain.

TFaaS is a system that aims to secure the execution of serverless workloads, which takes an approach different from Clemmys: instead of running user functions each in a different enclave and container, it executes them inside a single enclave for protection of user function from the untrusted host environment, and relies on the language-based isolation and sandboxing technologies to protect user functions from each other. It provides two execution modes, based on a minimal Javascript engine with low resource usage and lean TCB (Duktape) for most of the functions, and runs functions that require a high-performance JIT in Google's V8 engine. TFaaS provides a protocol for attesting both the enclave runtime and the in-enclave function when establishing the client connection. The limitations of TFaaS include lack of trusted resource accounting or explicit support of secure function chaining.

Similar to Clemmys, Se-Lambda relies on Intel SGX to protect the API Gateway and the user functions. For the API Gateway, it minimizes the system TCB by placing only the modules critical to the user data confidentiality into the SGX enclave, while keeping the rest of the gateway untrusted. Se-Lambda API Gateway contains code for remote attestation of user functions, and for establishing a secure channel to them. For the service runtime, Se-Lambda implements a two-way sandbox using Intel SGX, and WebAssembly as a software fault isolation technique, to strengthen the weaker isolation properties of containers commonly used in the serverless runtimes. As a performance optimization, Se-Lambda allows reusing the service runtime across invocations of different functions, improving the startup time by 20%. Se-Lambda also adds a shielding module for protection against the Iago attacks to the service runtime. Se-Lambda is built on top of OpenLambda, and uses Node.js as a service runtime. Unlike Clemmys, it does not implement any provisions for function chaining. Also, it does not rely on SGXv2, thus requiring costly initialization during the enclave startup.

Trustworthy resource accounting. In serverless platforms, reliable and trustworthy resource accounting is extremely important, as the cloud usage is billed per resource consumption. On the one hand, if the cloud operator has a bug in the resource accounting routine, it can lead to suboptimal system scaling or even allow dishonest users to get free executions. On the other hand, lack of transparency and independent verification of resource consumption allows dishonest cloud operators to increase the bills of the clients.

In the context of the former problem, Liang Wang et al. have studied and reverse engineered resource management policies of several commercial cloud platforms and discovered that they do not achieve the claimed levels of scaling [293]. Furthermore, the authors have discovered that there were several resource accounting issues, which could affect the host system operator.

In the context of the second problem, multiple research systems have been proposed:

VeriCount is an early system that explored resource accounting for SGX enclaves [282]. It relies on the compiler instrumentation to automatically insert the calls into the runtime accounting library, and the presence of the instrumentation code is checked by the static verifier. The runtime accounting library produces the resource accounting logs, which are processed by the post-execution analyzer and can be used to bill the client. VeriCount relies on SGX-provided trusted time source and on the monotonic counters to prevent the provider from slowing down the enclave by preempting or killing it, and to measure CPU time. For the network and disk I/O, VeriCount measures the number of transmitted bytes, but does not implement any mechanism for memory accounting. It has been since identified that SGX-provided time cannot be used for establishing the upper bound of CPU time inside enclave [134]; this topic is further explored in §7. VeriCount has also identified the problem of protection of the in-enclave accounting data from the enclave user code, and has proposed several solutions to this problem.

AccTEE is a two-way sandboxing framework for trustworthy resource accounting, built using WebAssembly for Intel SGX protection [134]. To allow the trustworthy accounting, it instruments the user code to count the number of executed WebAssembly instructions, memory allocations, and I/O operations. AccTEE is platform-independent and can be used with any programming language that supports the execution on WebAssembly virtual machine. It relies on the instrumentation enclave for modifying the user binary with the resource accounting instructions, which can be inspected by both cloud operator and user. For the CPU utilization, AccTEE efficiently maintains a weighted counter, which takes into account the complexity of the executed instructions, while minimizing the overhead of counting using static analysis. For memory, it tracks the usage of WebAssembly linear memory. For I/O operations, AccTEE tracks the executed operations of the underlying runtime (SGX-LKL).

S-FaaS also implements the mechanisms for trustworthy accounting, focusing on the three core metrics: memory, network utilization, and CPU time. Accounting for these metrics is implemented by extending the in-enclave runtime. For network utilization, S-FaaS measures the amount of data transmitted over socket interfaces. The CPU time inside the enclave is measured using an additional timer thread, which increments a counter in memory every N cycles, where the value of N is chosen to represent a commonly used real-world duration (e.g. 1 second). S-FaaS ensures that long preemptions cannot be used to slow down the counter by using a mechanism inspired by Varys [233], extended with the usage of Intel TSX to speed up the detection of interrupts. To estimate the memory usage, S-FaaS reads the CPU time on every operation that allocated or frees system memory. The CPU time is used to compute the memory-time integral, the amount of time at each memory allocation level in Mb$\times$sec. S-FaaS provides in-enclave isolation (interpreter- or sandbox-based) to forbid the user-provided code from tampering with the accounting data.

Optimizations of serverless performance. The performance issue of serverless computing has been the focus of research for quite some time already. In particular, function startup latency is the main performance bottleneck. Several approaches to alleviate this issue have been proposed, based on the reuse of function state, fast container or VM creation, or checkpoint-restore technologies.

SOCK [230] and Cntr [280] have suggested the use of lean containers to reduce the function startup cost. Additionally, SOCK uses the Zygote-based design to further speed up function startup. SAND [62] achieves low startup time using fine-grained function sandboxing where multiple instances of the same issue share the container, and by ensuring that message queuing and storage in the system have high locality. These optimizations do

not consider SGX-protected functions, but they are orthogonal to those proposed in our work. It should also be noted that using local queues or key-value storage to increase spatial and temporal locality of the processed data is a well-known approach in the serverless platforms [269, 97, 278].

Catalyzer [113] is a system for optimizing serverless system performance without compromising security. To this end, Catalyzer runs the functions in the virtual machines, however instead of booting them from scratch, they are restored from a known-good system image. As in Clemmys, the pages of the function image are added to the running instance on-demand, albeit using virtualization technologies, not Intel SGX. The creation of the function image allows optimizing both the boot time of the service runtime (VM), and of the language interpreters necessary to execute the function. To facilitate the creation of the images, a new system call, `sfork`, a sandbox fork, is proposed. It is possible to use Catalyzer with Intel SGX-protected enclaves as well, at the cost of modifying the enclave runtime. It should also be noted restoring system state from image typically removes the benefit of ASLR, and simplifies the exploitation of memory-based vulnerabilities.

Faasm [269] extends WebAssembly-based functions with two important extensions: shared memory support, and with a checkpoint-restore-based startup, achieving even lower function startup latency and per-function memory footprint. For resource isolation, on the other hand, Faasm relies on the operating system features, such as cgroups, virtual network interfaces, and firewall rules. Applications that need to interact with external systems can use a minimal systems interface, a message queue, and a read-global write-local file system. While Faasm assumes a trusted cloud provider, all of its optimizations can be applied to SGX enclaves.

Secure distributed computing frameworks. Several systems use Intel SGX to implement trusted distributed computing:

VC3 [262] is a framework for Map-Reduce computations in the cloud that relies on SGX to achieve its security properties. The authors also design a cryptographic protocol that enables verification of computations integrity. Unlike our work, the authors use an enclave partitioning scheme with minimal TCB for mapper and reducer code, and cannot run generic workloads.

SecureStreams [147] proposes a system for secure stream computations. In this system, ZeroMQ and symmetric encryption are used for secure network communication, while the stream processing functions are implemented in the Lua programming language. In our work, we also rely on custom protocol with symmetric encryption, while the function can be implemented in several programming languages.

Opaque [305] is a Spark-based data analytics platform with data-oblivious processing functions for untrusted cloud platforms. Unlike Clemmys, it ensures that no information about workload is leaked through metadata; in Clemmys, this property is considered to be out of scope, as Clemmys deals with much more generic workloads. Pesos [185] and Speicher [76] proposed secure storage solutions based on Intel SGX, which can be potentially used for the secure storage layer.

AirTNT combines blockchain-based smart contracts with Intel SGX-based TEEs and remote attestation to rent computing time on the machines of the remote users [63]. By repeatedly publishing the intermediate results of computations in exchange of payments, this systems ensures that the computation makes forward progress in presence of worker nodes that connect and disconnect from the distributed system.

In AirBox, Bhardwaj et al. design a system for efficiently outsourcing compute-intensive

computations to the edge nodes [86]. The authors reach a sweetspot between security, provisioning flexibility, and developer constraints for edge function by building on the container interfaces and Intel SGX TEEs. As a result, AirBox is performant and scalable in common edge computing benchmarks, and runs a wide variety of functions, including soft-state functions like caching. It should be noted that the performance evaluation of AirBox was done in a simulator, which may not faithfully represent Intel SGX overheads.

6.9 Conclusion

In this chapter, we have presented Clemmys, a secure platform for serverless computing, which allows users to ensure the confidentiality and integrity of functions' sources and data. Clemmys achieves these properties by building on Apache OpenWhisk, a popular serverless platform, Palaemon, a key management solution for Intel SGX, and SCONE, an Intel SGX TEE framework for unmodified POSIX applications. As part of Clemmys, we propose a message encryption scheme that ensures the integrity of the function chains. We have evaluated Clemmys in different scenarios and show that Clemmys achieves high throughput of function execution, while protecting against privileged adversaries.

Additionally, we show how the amount of EPC memory is likely to remain a limiting factor for real-world FaaS deployment until CPUs with alternative memory encryption technology become more widespread in the cloud.

7 A Trusted Lease Primitive for Distributed Systems

In the previous chapter, we have applied Intel SGX to serverless computing with Clemmys, allowing users to execute functions in the cloud with high scalability and low management overhead. However, an important challenge for Intel SGX was uncovered when stating the system model: are the users of the FaaS platforms function providers? This question is important, because the function provider may limit the number of concurrently running functions. This problem can be trivially solved by *leases*, but lacking a trustworthy time source, it is impossible to provide a lease mechanism that can operate in the threat models typical for Intel SGX.

A similar question was raised in the §5: while we have discovered a low-latency time source for SGX enclaves, this time source is not trustworthy, and can be manipulated by the OS. Even if this limitation affects only the availability of the system, which cannot be ensured with Intel SGX, the trustworthy time source and time-based protocols are a recurring theme when constructing systems based on TEEs. In this chapter, we will show that it is possible to build a trustworthy time-based protocol by constructing a lease protocol for untrusted environments using Intel SGX TEEs.

We focus on a lease as it is an important primitive for building distributed protocols, and it is ubiquitously employed in distributed systems. However, the scope of the classic lease abstraction is restricted to the *trusted* computing infrastructure. Unfortunately, this important primitive cannot be employed in the *untrusted* computing infrastructure because, as we have shown, the trusted execution environments (TEEs) do not provide *a trusted time source*. In an untrusted environment, an adversary can easily manipulate the system clock to violate the correctness properties of lease-based systems.

We will tackle this problem by introducing *trusted lease*—a lease that maintains its correctness properties even in the presence of a clock-manipulating attacker. To achieve these properties, we follow a "trust but verify" approach for an untrusted timer, and transform it into a trusted timing primitive by leveraging two hardware-assisted ISA extensions (Intel TSX and SGX) available in commodity CPUs. We provide a design and implementation of a trusted lease in a system called T-Lease—the first trusted lease system that achieves high security, performance, and precision. For the application developers, T-Lease exposes an easy-to-use generic APIs that facilitate its users to build a wide range of distributed protocols.

7.1 Introduction

Leases are one of the fundamental building blocks of distributed systems [137]. On an abstract level, a *lease* is a permission to access a shared resource for a certain period of time (the *lease term*). The lease is issued by an authoritative resource owner (the lease *granter*) to an entity that wants to access the resource (the lease *holder*). While the lease is active, the holder can freely access the resource without requiring any coordination with the granter.

Due to this coordination-less scheme, leases bring a significant benefit to building distributed systems for workloads with heavy read skew by eliminating the need for repeated resource locking. Therefore, leases are ubiquitously used in the design of distributed protocols and systems, such as two-phase commit [61], locking [299], consensus [193, 223], caching [300], leader election [123], failure detector [151], databases [229, 130, 102], storage [57], sharding [58], and file systems [226, 177, 213, 153, 130, 274]. Thanks to leases, such systems can achieve high performance and strong consistency with low overheads, while still allowing the writes to proceed. In this regard, leases are favorable compared to the traditional locking protocols for synchronization [152, 96]: with locks, the writes cannot proceed until all readers have unlocked their data.

Even though leases are widely used in distributed systems, their scope is mainly confined to the *trusted* computing infrastructure. However, this assumption is no longer valid with the prevalence of cloud computing: the potential risks of security violations in third-party cloud computing infrastructure have increased significantly. In an untrusted environment, an attacker can compromise the security properties of distributed systems. Many studies show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to distributed systems deployed on the untrusted computing infrastructure [141, 259]. In particular, lease violations can lead to both denial of service and correctness issues (see §7.2.1).

Intel SGX and SCONE can be used to protect the nodes which use lease-based protocols. However, even with Intel SGX, the design of a trusted lease is non-trivial: to enforce a lease term, TEE must have access to a trusted time source. Unfortunately, this important primitive is missing in the current versions of Intel SGX. In practice, an attacker has the capabilities to control the time sources (by setting value and frequency), the CPU frequency, by delivering interrupts and delaying messages, etc. Therefore, a strawman design for a trusted lease is bound to either be insecure (for a design where the lease duration is measured using the Timestamp Counter) or suffer from high performance overheads (for a TPM-based clock).

To overcome these limitations, we focus on the following question—*how can we design a trusted lease abstraction for distributed systems*? To answer this question, we present an abstraction of *trusted leases*, which we design and implement in a system called T-Lease. The trusted lease retains all properties of the classic lease [137], but it is designed to maintain its correctness properties even in the presence of a privileged attacker. More specifically, T-Lease is the first system for trusted leases with the following design properties:

- Security: It always maintains the lease correctness invariant; that is, the lease duration at the granter must be a superset of the lease duration at the holder.
- Performance: It imposes minimal performance overheads compared to classical leases.
- Usability *(time precision & APIs generality)*: It provides an easy-to-use generic APIs to support both short-termed (fine-grained time resolution) and long-termed leases for implementing a wide range of distributed protocols.

Timer	Type	OS mediated	OS controled	Cost
Software timer [263]	Software	No	Yes	Low
TSC [160]	Architectural	No (SGXv2)	Yes	Low
HPET [155]	Architectural	Yes: MMIO	Yes	Medium
PTP clock [284]	Hardware	Yes: MMIO	Yes	Medium
TPM [72]	Hardware	Yes: OS	No	High

Table 7.1: Time sources on the x86 architecture.

To achieve these design goals, we apply a "trust but verify" approach to a high-resolution low-overhead untrusted timer, improving its security without sacrificing performance and usability. More specifically, we transform the untrusted timer into a trusted time source by leveraging ISA extensions available in commodity CPUs. This transformation is based on a simple observation: *the untrusted timer can only be manipulated on interrupts*; thus, T-Lease needs to detect interrupts and verify the correctness of the timer after each interrupt detection. Thus, in addition to architectural features of Intel SGX, T-Lease relies on Intel TSX [160]. T-Lease relies on Intel SGX to detect interrupts by examining the memory used for the enclave state storage during the interrupt handling. Intel TSX provides us with a transaction primitive which we use to close a window of vulnerability after the lease check. A combination of these two features allows us to (a) detect interrupts before checking the lease, and (b) ensure that critical instruction sequences are executed without interrupts.

More specifically, our design builds on three core contributions: (1) *enclave-interval timer* allows secure and low-overhead measurement of the time intervals that the application spends inside the enclave; (2) *timer frequency verification* mechanism to prevent an attacker from manipulating the timer frequency, thus allowing verification of the timer correctness; (3) *transactional system call interface* using the hardware transactional memory to prevent the time-of-check to time-of-use (TOCTOU) vulnerability between a lease check and the corresponding system call submission.

We implement T-Lease as a static library, which provides an easy-to-use generic lease APIs for implementing a wide range of distributed protocols. In the evaluation, we study the performance, correctness, and precision properties of T-Lease using a set of microbenchmarks both in single-node and distributed system setups. We further employ T-Lease to design three real-world distributed case studies: (a) failure detector in FaRM [111], (b) Paxos Quorum Leases [223], and (c) strongly consistent caching [137]. The evaluation results show that T-Lease is effective at detecting timer tampering for TSC (x86 Timestamp Counter) and the overhead from the timer is minimal in a wide range of configurations (0-25%, up to 5% in most cases). Finally, the basic version of T-Lease protocol has been verified using TLA+ and TLC model checker.

7.2 Overview

A lease is a contract issued by a resource owner to give control to a holder over the protected resource for a certain time duration. This duration is defined using a *lease term* parameter. A lease term might have any length, from zero to infinity. In practice, however, the lease term is typically set to a limited amount of time. When the lease term expires, the holder has to either renew the lease or continue with reduced consistency (if allowed by the protocol).

Typically, classical distributed systems assume trusted environments in which they rely on

the system time sources, like `clock_gettime` to enforce the lease term. It provides resolution up to nanoseconds and has an extremely low overhead on modern Linux systems that use vDSO (virtual dynamic shared object) [55]. Compared to the classical systems, distributed systems built with TEEs assume a more privileged attacker who can affect the lease term by manipulating the system time resources. Hence, in this chapter, we introduce a novel concept of *trusted leases* to tackle this challenge.

7.2.1 A Case for Trusted Leases

The trusted lease abstraction is motivated by the necessity to secure distributed systems built using TEEs. TEEs provide strong confidentiality and integrity properties for the application memory but do not extend these security guarantees to the system time sources. Typically, the failure of the lease mechanism causes only denial of service. However, in the untrusted environment, a privileged attacker can manipulate the lease term as perceived by the holder or by the granter, leading to the violations of *correctness*, *e.g.*, system security properties that depend on the specific protocol. For example, leases may be used to limit the number of concurrently running enclaves (as a licensing mechanism for SGX runtime, or as a security measure, *e.g.* to limit brute-force throughput). In this case, the ability to continue running an enclaved application while the lease granter assumes that the enclave has stopped execution would constitute a violation of a safety property.

To support such use-cases, trusted leases retain all of the properties of classic leases, but extend them with a stronger threat model, where a privileged attacker tries to subvert its correctness by influencing the runtime environment. Therefore, in contrast to the leases for trusted environments, trusted leases cannot rely on the operating system time sources to enforce the lease term: these time sources are by definition under the control of the OS. Thus, only architectural time sources and TPMs could be used for the trusted lease implementation.

Hence, a trusted lease can be defined as a lease *designed to maintain its correctness properties even in the presence of a privileged attacker*. A timer manipulation, in the worst case, results in a performance loss.

Threat model. We assume a powerful attacker that has full control over the OS and can introduce arbitrary changes to the system configuration. We focus on the attacker that manipulates clocks: changes clock value and frequency, introduces delays into application execution and message delivery, manipulates CPU frequency. Therefore, standard timers provided by the platform or the OS are untrusted [69, 284].

Table 7.1 shows examples of the time sources available on the x86 architecture, their overheads, and the amount of control the OS (thus, the attacker) has over the time source. An attacker can directly affect the time readings using a variety of mechanisms: using MMIO control registers for HPET and PTP clocks, writing to model specific registers for TSC, and changing power management settings to modify the frequency of a software timer. She also has indirect ways to affect time reading by delaying the time reading requests and pre-empting the running application between the timer access and the use of the timing information (Time of Check vs. Time of Use vulnerability).

We assume a correctly implemented CPU and ISA extensions; that is, SGX protects the confidentiality and the integrity of enclave memory and TSX aborts transactions on interrupts. Other attacks, like buffer overflows [190, 231] and microachitectural side-channel attacks [233, 93, 294], are out-of-scope for T-Lease. Likewise, Denial of Service and perfor-

(H) `Init_Lease(Lease timeout)` Initializes a lease with configuration.
(G, H) `Init_Client(Local Addr, Remote Addr, AES key)` Initializes client communication endpoints.
(H) `Update_Renew_Lease(Lease, Client)` Updates and renews the lease.
(G) `Update_Lease(Lease, Current time)` Updates lease state without renewing it.
(G, H) `Lease_Protected_Syscall(Lease)` Enables TSX protection for system calls if lease is active.
(G, H) `T, AEX? = RDTSC_AEX()` Reports current `rdtsc` value and if enclave was interrupted since last call.

Table 7.2: T-Lease library APIs: G marks the functions used by the granter and H by the lease holder.

mance degradation attacks cannot be prevented using Intel SGX and are out-of-scope.

We consider attacks that delay the messages with *the results of lease-conditional computations* (in contrast to time read RPCs) out-of-scope: these attacks must be handled by the higher-level protocol, for example by including timestamps in those messages, as delays are common in distributed systems even without an attack.

7.2.2 T-Lease: A Trusted Lease Primitive

T-Lease overview. T-Lease allows building distributed lease-based applications that run inside Intel SGX enclaves and withstand attacks on the time sources by privileged adversaries.

T-Lease builds on three core abstractions:

- T-Lease presents an *enclave-interval timer* to securely estimate the duration of the time intervals using an untrusted, OS-controllable time source. This functionality is necessary to track the lease term on the granter and the holder sides. To achieve this, the enclave-interval timer builds on the SGX architectural features for interrupt detection and uses TSC as an underlying timer.
- T-Lease presents a *timer frequency verification* mechanism for thwarting attacks that manipulate timer tick rate. To achieve this, we design an empirical approach that measures the duration of a sequence of attacker-uncontrollable instructions (RDRAND) and aborts execution if the result is out of architecture-determined bounds.
- T-Lease presents a *transactional system call interface* to communicate the results of computations that depend on the lease being present. So, the holder can atomically check the lease state and communicate the computed results; thus, avoiding the TOCTOU vulnerability. We achieve this by starting an Intel TSX transaction before checking the lease and committing it on a successful system call submission.

T-Lease APIs. T-Lease implementation consists of a client library and a reference implementation of a lease granter, which communicate over a UDP socket. The granter runs on

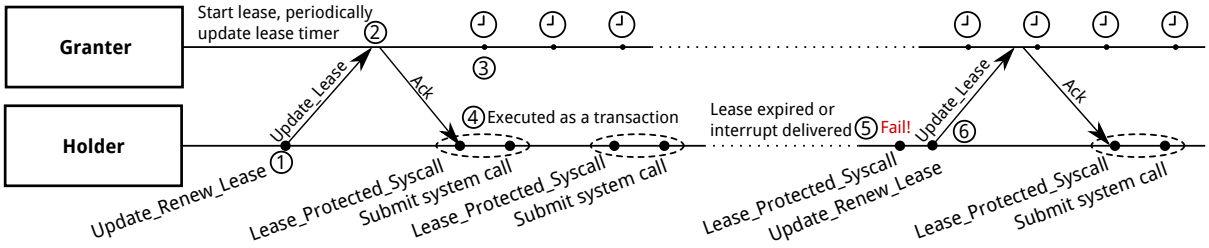

Figure 7.1: Basic workflow of the T-Lease protocol.

a dedicated machine and typically serves several clients. Table 7.2 lists the core APIs that are exported by the library for use by the granter and the holder (the service and auxiliary functions are not shown).

The functions can be divided into the following categories: Initially, two initialization functions are used for the lease and lease protocol client: `Init_Lease` and `Init_Client`. They are called by the granter and holders to initiate working with T-Lease. Then, two core functions are used to maintain a correct T-Lease protocol state: `Update_Renew_Lease`, called by the holder to request a lease and update the lease state, and `Update_Lease`, called by the granter to update the state of the lease at its side. `Lease_Protected_Syscall` is a function for secure results submission: it is used by the holder to atomically check the lease state and submit the computation results to the client. `RDTSC_AEX` is a low-level function, exported to facilitate building more complex distributed protocols than the default lease protocol used in T-Lease. Note that while T-Lease provides a simple, minimal lease protocol, it can be generalized to more complex protocols.

T-Lease basic workflow. Figure 7.1 describes the operation of T-Lease. First, since T-Lease uses TSC as a time source, which measures time in cycles, it is necessary to calculate nanoseconds-cycles conversion factors. Then, both the holder and the granter can initialize endpoints by using `Init_Client()`. This function opens the connection to the granter and configures the cryptographic key used to secure the communication. Next, the lease holder initializes the lease: sets the requested lease term and the lease identifier. Thereafter, the holder can request the lease from the granter ①.

The granter enters a work loop, where it first receives a message from the holder, and based on the holder command activates or disables the lease ②. After serving a message from a holder, it updates the state of all active leases, by using `Update_Lease` function ③. This function updates the enclave-interval timer for each lease, disabling all leases for which the accumulated timer value is larger than the lease term.

The holder, upon receiving a lease, enters a work loop: for example, a cache server may be handling user requests. It gets the user's request, processes it, and submits the results to the user. This operation is only valid if the lease is active, hence, it needs to call `Lease_Protected_Syscall()` to check the lease state and submit the system call in a transactional manner ④. If the return value indicates that the transaction is active, the system call can be submitted. If the transaction is inactive ⑤, the holder needs to renew its lease using `Update_Renew_Lease` and retry ⑥. The conditions under which transactions may become inactive are explained in §7.3.2.

7.3 Design

In this section, we first present two strawman designs and associated design challenges to realize the trusted lease abstraction. Thereafter, we present a detailed design of T-Lease.

7.3.1 Strawman Designs and Associated Challenges

As T-Lease is designed to operate with Intel SGX enclaves, it may access several timers, presented in Table 7.1. There is a set of trade-offs associated with each timer, that fall on the axis of the timer access cost and the control that the OS has over the timer. For example, some timers can be accessed only via the OS. For these timers, the OS can introduce arbitrary delays into message reads, so that these timers can be used only to establish the lower, but not the upper bound on the elapsed time. Other timers, like software timer and TSC with Intel SGXv2, can be accessed directly[1]. However, the OS can use the following capabilities to subvert the timer readings:

- *Power management:* Changing the CPU frequency influences the tick rate of a software timer.
- *Preempting the application or delaying messages:* This attack can be used on any OS-mediated timer.
- *Modifying timer value:* The OS can change TSC readings by writing to IA32_TSC_ADJUST or IA32_TIME_STAMP_COUNTER model-specific registers when the enclave is preempted, and spoof HPET or NIC PTP clock readings via MMIO writes.
- *Modifying timer frequency:* On virtualized platforms, writing to TSC Multiplier and TSC Offset fields in the VM control structure changes the TSC speed [160]. This attack also requires preemption of the VM.

All of these attacks must be thwarted by the T-Lease design, which is a non-trivial task. Consider, for example, the following two strawman solutions:

TPM-based design. Consider a design where the enclave checks the lease expiration using the time read from the TPM timer. Because the OS mediates in TPM communication, such a design cannot guarantee the *correctness* property. Specifically, during the lease check period, the OS can delay the TPM read beyond the lease expiration time. The holder gets the TPM read result after the lease expires, thus violating the lease invariant. Another vector for subverting system correctness is the delivery of an interrupt between the lease check and returning the results from the enclave. If the enclave execution resumes only after the lease expires, the lease invariant is also invalid. Besides, the TPM fails to meet the speed and accuracy goals. Reading a digitally signed TPM time takes from 50 ms to 600 ms depending on the selected cryptography system, i.e., hash-based or asymmetric cryptography.

We note that `sgx_get_trusted_time`[2] from Intel SGX SDK suffers from the same issue. It involves OS-mediated communication between several enclaves and the hardware: the time read RPCs are passed between the application enclave, the service enclave manager, and Platform Services Enclave, which communicates with the hardware via an OS driver.

[1] This is also true with common configurations of VMs, which *do not* cause `#VM` exception when the guest reads TSC.

[2] This function has been removed in the Intel SGX SDK v2.8.

These RPCs are cryptographically protected, but the OS can delay their delivery to any of the components.

TSC-based design. In a TSC-based design, the lease is initialized, its term in seconds is converted into `rdtsc` cycles using the CPU-specific multiplier, and the lease requested from the granter. After the granter acknowledges the lease, the expiration point (in `rdtsc` cycles) is calculated; as soon as this point of time elapses, the lease becomes invalid. The granter and holder both track the lease duration. With this design, the attacker has two prospects for subverting the security requirements. First, the OS can preempt the application, write to the Model Specific Registers to set the time inside the enclave back into the past, and then continue the application execution. Secondly, the attacker could launch the application in a VM, and use TSC Multiplier control to slow down the TSC. Next time when the enclave reads the time, it will not be able to detect the lease expiration.

Design challenges. To summarize, these attack vectors present the following design challenges for T-Lease:

1. How can the lease term be securely measured by the granter and the holder?
2. How can the timer frequency be verified?
3. How to atomically perform the timer check and return results?

We next explain our system design that addresses these challenges.

7.3.2 T-Lease Detailed Design

In this section, we describe the detailed design of T-Lease that addresses the aforementioned challenges. We explain how enclave-interval timer abstraction allows T-Lease ensure that the time measurement is untampered (§7.3.2) while the timer frequency is correct (§7.3.2). In §7.3.2, we show how T-Lease closes the window of vulnerability between lease check and returning computation results using the hardware transactional memory.

Enclave-Interval Timer

To help solve the first challenge, we use the following intuition: for the lease implementation, there is no need to measure the absolute time, only the relative—that is, time differences. To securely measure time intervals using the OS-controlled untrusted timer, which would retain the performance characteristics of the underlying clock, we need to: (a) ensure that the underlying timer was not manipulated or delayed for some period of time, and (b) precisely establish points when the manipulation could take place.

First, it is necessary to choose a time source. We note that all of the OS-mediated sources do not allow establishing whether the timer was not manipulated (i.e., each access is potentially manipulated), so, they cannot be used in the design of T-Lease. Thus, only the software timer and TSC with SGXv2 can be used since their value or frequency can be manipulated only when the enclave is preempted. We chose to use TSC in the implementation of T-Lease, because it incurs lower performance overhead: it does not require a dedication of a CPU core to a timer thread. Since in our case the resulting clock measures the duration of time intervals inside the enclave, we call this timer an *enclave-interval timer*.

Observing the capabilities of an attacker, we conclude that the attacker needs to deliver an interrupt to tamper with system clock configuration in all cases. T-Lease uses the corollary

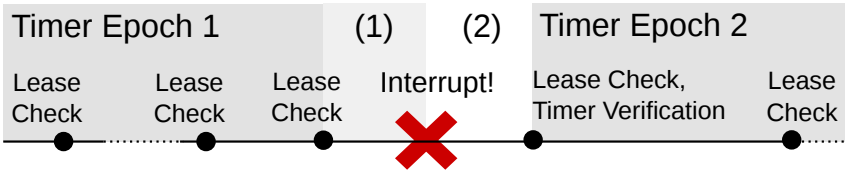

Figure 7.2: Enclave-interval timer operation. (1) Under-accounted time inside enclave; (2) Correctly unaccounted time outside enclave.

of this fact: *we can safely estimate a period of time as long as the entire period is spent inside the enclave, that is as long as no interrupts happen during that period*. We call such a non-interrupted period of time an *epoch*.

We detect interrupts by inspecting enclave State Save Area (SSA), a preconfigured memory region that saves the register state of enclave upon receiving an interrupt [103]. SSA has a predefined format, with fields for the registers and service data. We write 0 (zero) into the field of the IP register, which is an invalid value for that register. Later, we can check the value of that field, and if an interrupt happened, we will detect a non-zero value there.

To estimate the duration of an active lease, T-Lease periodically reads the TSC value and checks for the interrupts using the aforementioned mechanism. If no interrupt was delivered, it adds the duration of an interval from the previous such check to the current moment to the lease active time. In case there was an interrupt, the operations for the granter and holder are different. Because the lease term on granter must be a superset of the lease term on the holder, upon detecting an interrupt at the granter T-Lease can continue operation in a normal mode. This functionality is implemented in function `Update_Lease`.

The holder, however, cannot do the same, because it could have been preempted for an arbitrarily long period of time, and its lease on the granter could have expired in the meantime. Thus, upon each interrupt, the holder has to renew its lease from the granter. As before, the request-reply interaction should happen in the same epoch; otherwise, there is no guarantee that the packet has not been delayed. We have implemented the lease state update and communication in function `Update_Renew_Lease`.

Timer Frequency Verification

To solve the second challenge in which the attacker could change the TSC frequency in addition to the TSC value, T-Lease must verify the timer frequency after each interrupt.

A strawman design of the verification routine could consist of a sequence of instructions with deterministic execution time, *e.g.*; noops or in-register additions. However, these actions have a significant drawback: they open a privileged attacker a possibility to tamper with the execution speed of the CPU using the power management features.

The ability of an attacker to control the power management features has far-reaching implications for the verification routine: most modern Intel CPUs have constant TSCs, that is TSC speed is independent of the CPU frequency. On the other hand, the speed of other components of the CPU does depend on the CPU frequency: by manipulating CPU frequency and `rdtsc` speed simultaneously, an attacker can trick a simple verification routine into believing that the TSC rate is normal.

Therefore, the procedure that verifies the timer frequency must not depend on the CPU speed. By analyzing the literature [159, 143] and performing experiments on multiple SGX-enabled platforms, we have discovered that the RNG module embedded into the Intel CPUs

to implement RDRAND instructions is independent of the CPU frequency: entropy collection module is self-clocked at 3 GHz, and the post-processing module runs unconditionally at 800 MHz. Therefore, we use a sequence of six RDRAND instructions to measure the `rdtsc` rate. The number of RDTSC instructions to execute is a trade-off between accuracy and the verification cost.

Our measurements (§7.5.2) have shown that the latency variance of RDRAND is high: between 7000 and 10500 cycles. Due to an inherent variation of cost of this instruction, the attacker would still be able to modify the TSC frequency in some bounds. To increase the reliability of the rate estimation, the measurement can be repeated several times. While our verification routine depends on the microarchitectural details of the RNG, the RDRAND latency falls into these bounds on all SGX-enabled CPUs that were available to us, thus we argue that this technique is applicable in practice.

Transactional System Call Interface

Finally, T-Lease has to close a window of vulnerability between the lease check and the externally-observable actions that are conditional on the lease state. In our model, we use system calls (which may involve writing to disk or sending a message over the network). We argue that this model is adequate for most of the currently used distributed systems, as the number of TEE-based systems that use kernel bypass for the communication is comparably small.

We observe that with Intel SGX, the only way for an enclave to submit computation results is via the shared memory writes. Thus, the required atomicity of the lease check and the result submission can be achieved using the hardware transactional memory: if an interrupt is raised while the transaction is active, the underlying hardware will automatically rollback all changes made in the transaction. We consider a system call submitted when it becomes visible on the unprotected shared memory.

T-Lease uses Intel TSX to check the lease and submit computation results in a single atomic transaction [160, 189]. Intel TSX allows applications to perform arbitrary memory reads and writes in an atomic, transactional manner. TSX imposes some limitations on these transactions: the amount of writes that may happen in a transaction is limited by the L1 cache size, some instructions inside transactions are forbidden. In case these limitations are violated, a read-write or write-write conflict is detected, or an interrupt is delivered, the transaction is rolled back with an error flag set. To limit these effects, we commit the transaction immediately after a system call is submitted (§7.4.1).

The attacker can still delay the message or disk write after they are submitted, but this cannot violate the security properties: the messages/writes can be delayed in a distributed system even without an attack, and designing a system to tolerate these delays is out of scope of T-Lease. For synchronous and timed asynchronous systems, the maximum delay must be taken into account when checking if the lease is active.

7.4 Implementation

7.4.1 Implementation of the T-Lease Library

We implement T-Lease as a static library in 1037 lines of ANSI C, and 26 lines of inline assembly for the Intel TSX support and reading TSC.

Intel SGX framework. T-Lease relies on SCONE (§3) as an underlying SGX framework and to access the SSA region. Our work, however, is conceptually independent of SCONE and can be built on top of Graphene-SGX [287] and Intel SGX SDK [158]. Other than modifying the system call thread code for reducing the Intel TSX abort rate, we have added a transaction commit code in the SCONE system call handler to reduce the transaction length, and to ensure the pinning of threads to cores.

Communication. T-Lease uses UDP sockets for the communication, which is common for latency-sensitive services. All communication between the nodes is encrypted with AES-GCM-256. We use Intel IPSec Multibuffer Encryption library [54] for these cryptographic functions. T-Lease leases currently use a pre-shared AES key; in production use, we expect to use a full-fledged key management service, for example Palaemon [138], for the key distribution.

TSX-specific optimization. When designing the TSX protection, we need to take into account the architecture of SCONE. Since SCONE uses asynchronous communication via concurrent queues between the enclave and the untrusted world, the transaction abort rate due to the read-write conflicts between the system call thread and the in-enclave thread was reaching 79%. We have fixed this issue by adding six `pause` instructions into the back-off routine of the system call thread, as a trade-off between the instruction overhead and the abort rate. This has reduced the abort rate of transactions to 0.008% on a system call intensive benchmark without reducing its performance. This code can be further improved using the recent Intel's `TSXLDTRK` extension for suspending load tracking inside the transaction region, however we do not expect a significant performance effect from this optimization.

7.4.2 Implementation of the T-Lease Case Studies

To demonstrate how T-Lease can be used in practice, we apply it to three state-of-the-art distributed systems that rely on leases. In the following case studies, we implement a standalone implementation of granters (or nodes with equivalent features); for the holder part, support for each of the use-cases was added into the client library.

- FaRM [111]: Uses leases for failure detection;
- Paxos Quorum Leases [223]: Uses leases in its lease configuration activation protocol (after which it switches to leases with the unlimited term).
- Strongly Consistent Caching [137]: Uses leases as a part of a file system caching service.

Failure detector in FaRM [111]. FaRM is a high-performance distributed transactional storage with high availability and strong consistency [111]. FaRM uses leases as a failure detector: each worker node has to maintain a lease on a cluster manager node. When a lease expires, this signals to the lease granter that the lease holder has failed, and triggers the FaRM cluster reconfiguration. We implement the same failover protocol, recreating as many details of the original paper as possible (the lease renewal rate is set to 1/5 of lease duration, etc.). An attacker may choose to modify the time at the cluster manager node, preventing the detection of outdated leases. In this case, the cluster reconfiguration will not be updated, and the client would be directed to a node in a failed state.

Paxos Quorum Leases [223]. Paxos Quorum Leases is a modification of the Paxos protocol that splits objects and nodes of the system into lease groups according to the frequency of

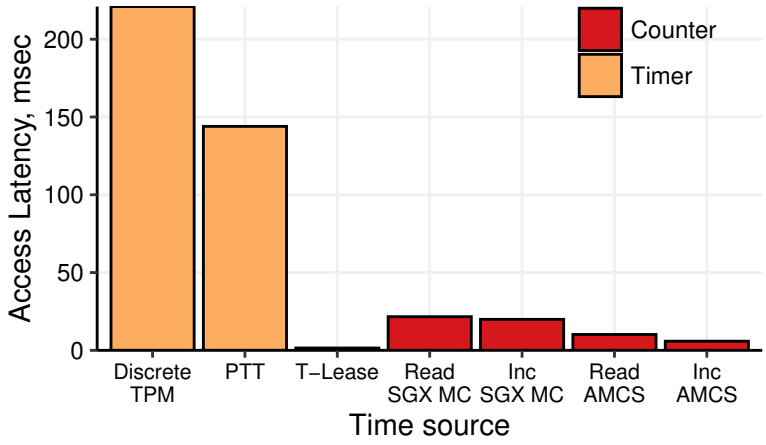

Figure 7.3: Access latency of trustworthy clocks and timers.

accesses to each of the objects on each node [223]. Inside lease groups, each node has an infinite term lease to objects belonging to the group, and it can serve read accesses to these objects without consulting the majority of the nodes; thus, it significantly improves the system throughput. While the lease itself has an infinite term, to activate a lease configuration, each node must exchange a non-infinite lease with a majority of the nodes in the lease group. An attacker that manipulates the time on one or multiple machines can cause the node to assume that it has successfully established the lease with the majority of the nodes, while in practice this would not be true. By using T-Lease inside the lease activation protocol, we can ensure that the attacker cannot violate the system correctness.

Consistent caching [137]. Strongly consistent caching is a use-case that is commonly used in distributed systems to improve throughput and latency [137]. It uses standard leases to grant caching node access to a set of objects (files on the file system, database rows) for a lease term, during which reads from the caching node can be done without consulting an authoritative data source, and the data source will notify the caching node about any write to objects under a lease. This system relies on the invariant that a lease duration at the lease holder is shorter than the lease duration at the granter. Violation of this requirement may cause stale reads or even conflicting, inconsistent results. Strongly consistent caching implemented with T-Lease detects the manipulations of system time and ensures the correctness of the system.

7.5 Evaluation

Our experimental evaluation answers the following questions:

1. **Single-node setup:** What are the performance, correctness, and precision properties of individual low-level components of T-Lease? (§7.5.2)
2. **Distributed setup:** What is the performance of T-Lease in a distributed multi-node setup? (§7.5.3)
3. **Case studies:** Can T-Lease be used for real-world distributed protocols? What are the associated overheads compared to classical untrusted leases? (§7.5.4)

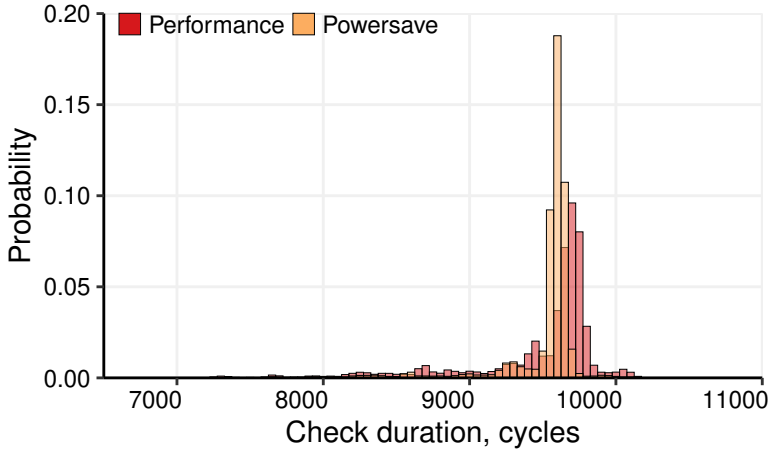

Figure 7.4: Latency of TSC timer check using 6 `rdrand` instructions.

To this end, we devise a set of microbenchmarks to answer the Question 1, and develop a set of distributed experiments to answer the Question 2. To answer the Question 3, we evaluate the systems outlined in §7.4.2.

7.5.1 Experimental Setup

Testbed. We use two types of machines. *SGXv2 NUC* is an SGXv2-capable Intel Pentium Silver NUC (Gemini Lake) operating at 1.5 GHz with 16 KB L1, 128 KB L2, and 4 MB L3 caches, and 32 GB of RAM. *SGXv1 server* is a Dell PowerEdge R330 server with an SGXv1-capable Intel Xeon E3-1270 v5 CPU (Skylake), with 32 KB L1, 256 KB L2, and 8 MB L3 caches, 64 GiB RAM, and a discrete Infineon 9665 TPM 2.0. We use the SGXv1 server in distributed experiments. Both machines are connected to a 1 Gb/s switched network. To measure Intel PTT access times, we use Intel NUC with an Intel Core i7-7567U CPU at 3.5 GHz (Kaby Lake microarchitecture, 2 cores, 4 hyperthreads) with 32 KB L1 and 256 KB L2 private caches, 4 MB L3 shared cache, 8 GB of RAM, and TPM 2.0 compliant Intel PTT provided by Intel ME under version 11.8.50.3425.

Methodology. As system interrupts have a major influence on the functioning of the T-Lease timer, we reconfigure the system to reduce their frequency. We change the kernel configuration to the lowest possible timer interrupt frequency (100 Hz), enable the dynamic ticks kernel mechanism, and steer all device interrupts to core 0. Because of the severe performance impact of interrupts on SGX enclaves, these changes are generally beneficial to SGX-based systems [103].

7.5.2 Single-node Setup

We first evaluate (a) performance, (b) correctness, and (c) precision properties of T-Lease in a single-node setup.

(a) Performance: Access latency. We begin by measuring the access latency of available secure clocks and timers (Figure 7.3). Without interrupts, the access latency for T-Lease is ~30 ns. When interrupts are delivered, the minimum cost is approximately 10k cycles (20.3 µs), and the full cost of using a timer will depend on the interrupt recovery actions. This

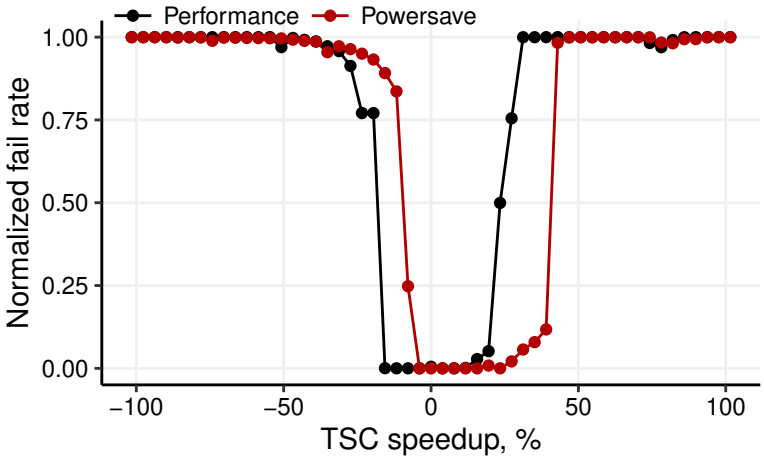

Figure 7.5: Probability of detecting the TSC rate manipulation.

latency is significantly lower than the latency of TPM-based timers (220 ms for discrete TPM, and 145 ms for Intel PTT).

To put the results into a perspective, we also show the access latencies of several *counter* implementations, which may also be used in a distributed system for message ordering and conflict resolution. SGX MC is the Intel SGX SDK monotonic counter implementation, which is built using Intel ME. AMCS is the network service that exposes the Intel SGX MC over the network, with on-disk caching for counter values (see §7.6). T-Lease performs favorably to these systems as well.

(a) Performance: Epoch duration. To apply T-Lease in practice, the epoch duration must be large enough for both communication with the granter and performing useful work. We evaluate the average duration of the lease on the system configured to reduce the interrupt frequency. We have discovered that an in-enclave thread running with normal priority has an average epoch duration of 15 ms; when a thread is running with a real-time priority, it achieved an average duration of 650 ms. Thus, in further experiments, we configure the enclave threads to run with real-time priority. In general, this change is beneficial for the SGX enclaves since frequent enclave exits significantly reduce enclave performance [103, 75].

(b) Correctness: TSC rate estimator. Next, we evaluate the operation of T-Lease `rdtsc` rate estimator (§7.3.2). Since an attacker can break the operation of a naive rate estimator by changing the CPU frequency, we perform this experiment at two extreme CPU frequency values (800MHz and 1.5GHz on SGXv2 NUC). If the estimator performs correctly for both of these values, T-Lease will operate correctly with any intermediate frequency value. We change the CPU frequency by setting the system frequency scaling governor to `performance` for 1.5GHz, and `powersave` for 800MHz.

Because the latency of `rdrand` and `rdtsc` is not fully deterministic, there is noise in the measurements. To illustrate it, we measure the latency distribution of our estimator (Figure 7.4). The check duration is largely independent of the CPU frequency, which matches the documentation [159]. Thus, we can use these operations to implement a TSC rate estimator. Execution of the estimator should take between 7.5k and 10.5k cycles.

However, a range of 3,000 cycles is still large enough to allow an attacker to manipulate the TSC rate. We evaluate the bounds in which the attacker can successfully manipulate the tick rate without being detected (Figure 7.5). To this end, we measure the probability

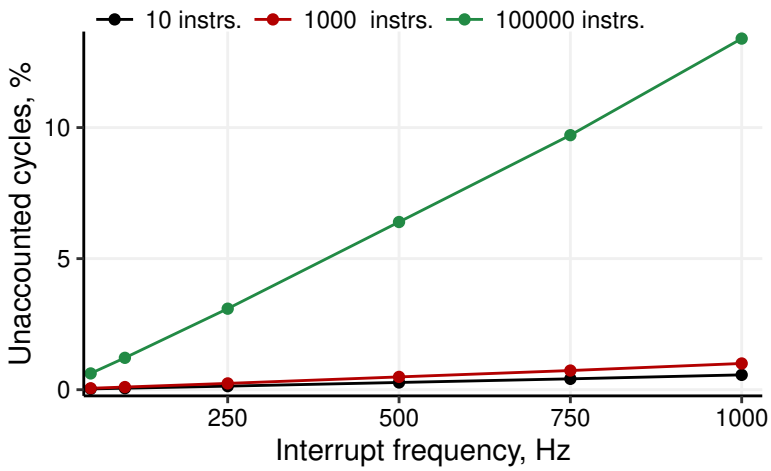

Figure 7.6: Underaccounted cycles as a function of an interrupt rate and an interval between lease checks. The microbenchmark executes a number of in-register instructions (10/1000/100000) between increments of the enclave-interval timer. The plot shows the relative difference between the ground truth timer and the enclave-interval timer.

with which T-Lease estimator will declare `rdtsc` manipulation depending on the change in the tick rate. We can see that the attacker can slow down the timer by approximately 40% or speed it up by 45% with a high success probability (low risk of detection). To protect against this manipulation, we need to either make the lease $\frac{1+0.45}{1-0.45} = 2.64$ times shorter at the lease holder, or increase its length at the granter by the same factor. We increased the lease duration at the granter by a factor of 2, which is a trade-off between attack detection probability and lease term extension, and which we use in all further experiments.

(c) Precision: Under-accounting due to interrupts. For leases to operate correctly, the granter's lease term must be longer than the holder's. However, this can cause a precision issue even if there is no attack: interrupt causes under-accounting of time by the interval starting from the last time update or lease check; this extends the lease by the lost interval (Region (1) in Figure 7.2). We measure the amount of time that is lost depending on the interrupt rate, and the number of instructions between the interval timer updates. The results are in Figure 7.6. We see that if the lease is checked in a tight loop, the loss of precision is minimal even at the extreme interrupt frequencies. If the granter performs significant work between the timer updates, it can incur up to 14% precision loss. Yet, we note that the normal interrupt frequency on Linux is 250Hz for desktop and 100Hz for the server configuration, so even in the case of significant work between the timer updates, the lease extension should stay within 5% of the nominal lease term.

7.5.3 Distributed Setup

Next, we evaluate trusted leases in a distributed setup with various system configurations: local (granter and holder on the same machine) and remote (different machines), to estimate the network latency impact.

Lease check frequency. First, we measure the frequency with which the lease holder can check the lease expiration (Figure 7.7). The measurement shows whether T-Lease can be-

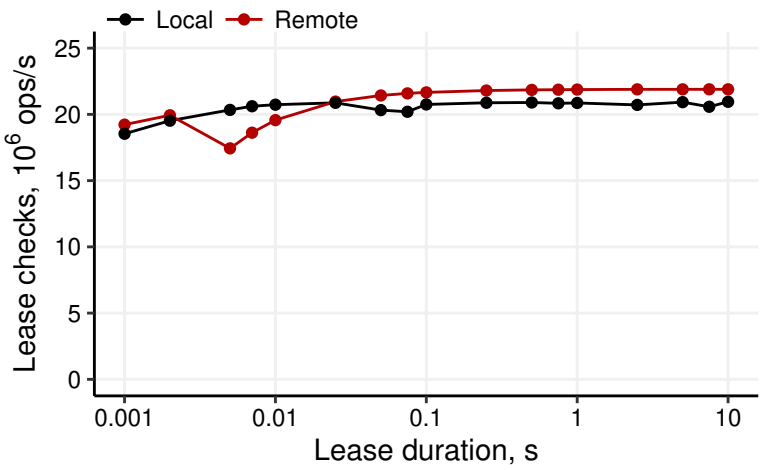

Figure 7.7: Client lease expiration check rate as a function of lease duration.

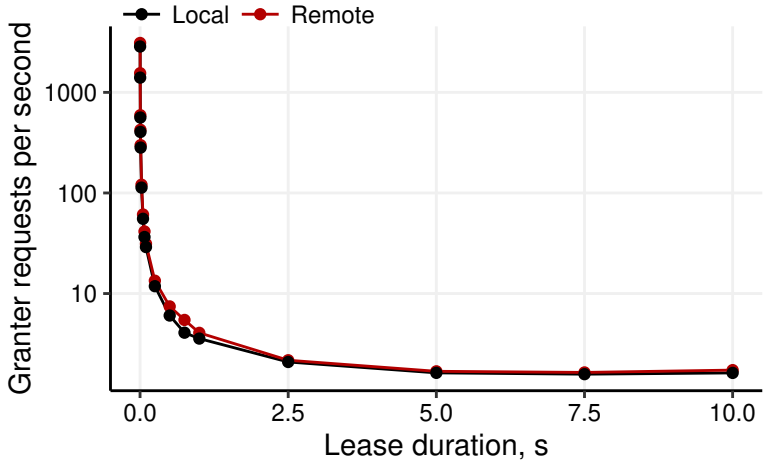

Figure 7.8: Frequency of network requests from holder to granter as a function of lease duration.

come a bottleneck if the checks are located on a hot path. As we can see, the frequency has a minor dependency on the lease duration: The longer the leases are, the less frequent lease requests become, and the less time is spent waiting on granter responses. In the remote setup, network delays also decrease the check rate, but only up to the lease duration of 12.5 ms, after which the effect becomes negligible. Notably, with longer leases, the remote setup has higher check rates compared to the local setup because in the local setup there are not enough free cores for system tasks, thus causing higher interrupt rates.

Frequency of remote requests. We also measure the rate of lease extension requests (Figure 7.8). They happen when either a lease expires at the holder, or an interrupt ends the holder's epoch. The message rate is driven by the lease duration for very short leases; with longer leases, interrupts maintain the minimum message rate. In this experiment, we have not discovered any difference between the local and remote setups.

Impact of interrupts. In T-Lease, the holder must request a new lease after every interrupt, causing an increased lease request rate if the system issues frequent interrupts. We eval-

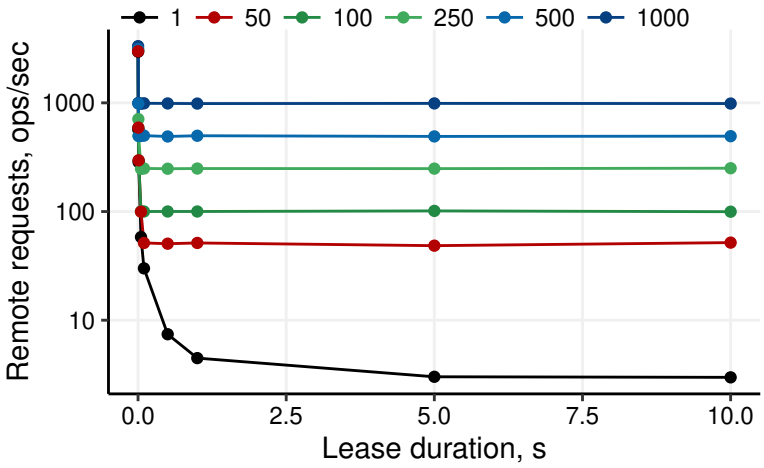

Figure 7.9: Request rate as a function of system interrupt rate (HZ).

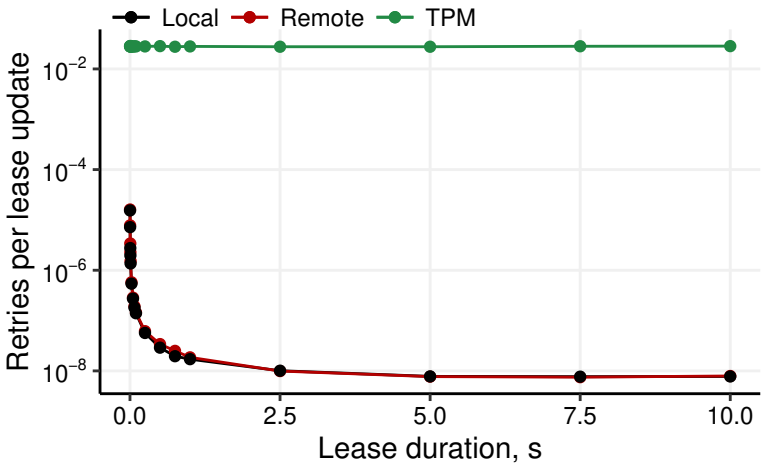

Figure 7.10: Frequency of retries due to interrupt delivery during lease renewal. TPM has much higher rate of retries due to its high access latency.

uate this property in Figure 7.9. While typical Linux systems are configured to have a timer interrupt rate between 100 and 250 Hz, devices such as disks can generate interrupts at a much higher rate. In this measurement, we estimate the message rate in a local setup that results from different interrupt rates, from 1 to 1000 Hz. They cover a wide range of usage scenarios, from a mostly idle server (1 Hz) to a server overloaded with interrupts (1000 Hz). For longer leases, the interrupt rate determines the communication rate in all of these cases; for the short leases, the communication rate is driven by the lease expiration. For high interrupt rates, the system may experience a high message load (2000 messages/s for response and reply).

Lease acquisition retries. If an interrupt is delivered before the granter response arrives, the holder sends one more request. We measure the average number of retries due to such interrupts, normalized by the total number of lease checks (Figure 7.10). With T-Lease timer, the number of retries is negligible (below 10^{-6} for leases longer than 100 ms). When instead a TPM is used as a trusted time source, the average retry rate is 0.28 per lease renewal. It

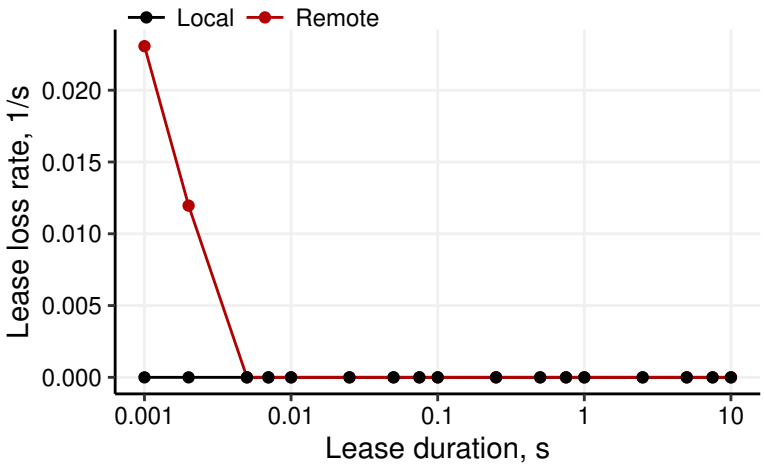

Figure 7.11: Number of lost leases per second for the local and remote T-Lease setups.

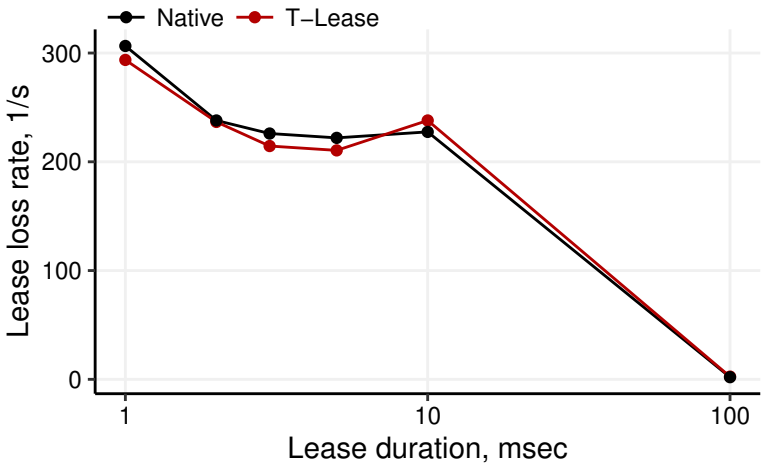

Figure 7.12: Number of lost leases for the FaRM failover protocol as a function of lease duration.

proves our previous claim that TPM cannot be used to efficiently implement the lease service as-is.

Lost leases. We evaluate T-Lease performance as a failure detector (Figure 7.11). To this end, we measure the rate of lease expiration despite the lease holder being active (false positives). The lease expires when the holder is descheduled for a long time, or if the packets with lease renewal messages are delayed or lost in the network. T-Lease performs without lost leases in the local communication case. However, in the case of network communication, leases with terms shorter than 5 ms exhibit a false positive rate of around 1 lost lease every 2 s. In practice, network delays are taken into account when choosing a lease duration [137], which allows minimizing the lease loss.

7.5.4 Case Studies

In our use-case study, we focus on the following questions:

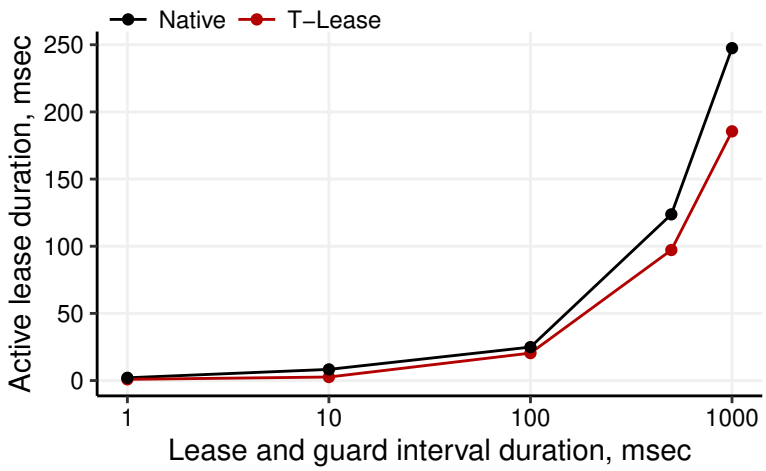

Figure 7.13: Timer interval duration with active lease for the PQL case study with varying active and guard intervals duration.

- Can T-Lease be used for implementing real-world distributed protocols?
- What overhead does T-Lease add compared to the standard leases?

To showcase T-Lease in realistic conditions, we applied it to time-critical components of several distributed systems as outlined in §7.4.2. In this section, we compare the performance of the use-cases as implemented with T-Lease to the implementations using standard leases.

Failure detector in FaRM [111]. Following the original FaRM paper, we measure the number of lease expirations (i.e. due to message delays or thread inactivity) over 10 minutes (Figure 7.12). FaRM uses unreliable datagrams over RDMA for transport, while T-Lease relies on UDP/IP and Ethernet, thus rendering the direct number comparison meaningless; nevertheless, the comparison of T-Lease with its version without interrupt detection allows us to determine the overhead of T-Lease. Unlike the original FaRM, which operates successfully with a 10 ms lease interval, T-Lease achieves operation without lost leases only at 100 ms lease duration: FaRM implements optimizations for eliminating false positives at shorter lease durations. On the other hand, the lease loss rate is the same in native and the T-Lease cases.

Paxos Quorum Leases (PQL) [223]. We implement the protocol in two variants: T-Lease and native. Because interrupt delivery may cause additional lease requests and affect performance, we measure the average duration of a lease depending on guard and lease interval, set to the same value (Figure 7.13). As the lease interval increases, the active lease duration also increases, albeit in smaller steps. The native variant has slightly longer durations of active intervals, as the interrupts cause lease invalidation. This makes active interval for long leases close to 247 msec, while it is 185 ms in the case of T-Lease. To put these numbers in a perspective, PQL authors used a 2-second lease duration with a 500 ms interval between renewal.

Strongly consistent caching service [137]. Our strongly consistent caching service implementation follows that of the original lease paper. We configure the system such that the granter is running on the SGXv1 machine outside of the enclave, and two cache nodes are running on the SGXv2 machine. One of the cache nodes is acquiring only read leases, while

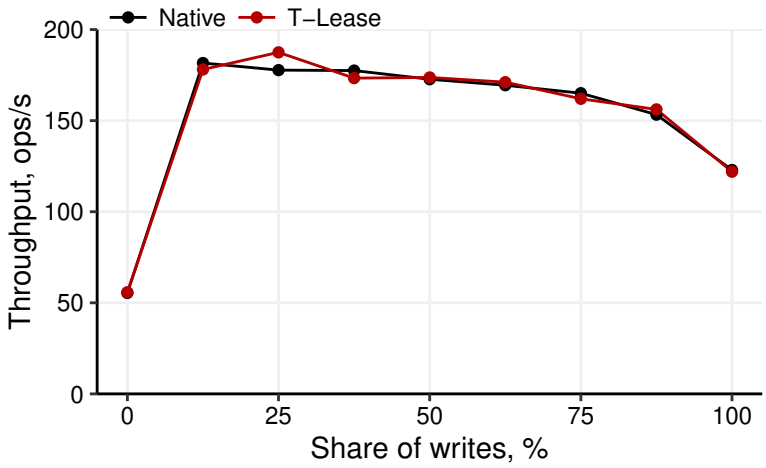

Figure 7.14: Message rate for the strongly consistent caching case study as a function of write ratio.

the second node acquiring read or write leases with a controlled probability. We measure the average number of messages per second over a 5 minute period (Figure 7.14). As the number of writes in the system becomes non-zero, the messaging rate in the system increases from 55 to approximately 175 msg/s. This is caused by frequent changes between reading and writing lease states, causing frequent lease invalidation and re-establishment. As the write share reaches 100%, the messaging rate decreases, as the writer submits requests faster than the reader reacquires the lease.

Summary. Overall, we can see that in most cases using T-Lease does not influence the system performance, with exception of the PQL use-case, where we have detected a reduction of active lease interval due to interrupts.

7.6 Related Work

In this chapter, we present the systems related to T-Lease in three categories: trusted hardware for distributed protocols, trustworthy monotonic counters, and trustworthy timers. For the low-overhead timers for Intel SGX enclave, we refer the reader to §5.2.

Trusted hardware for distributed systems. The pioneering systems that used commodity trusted hardware for securing distributed protocols were TrInc [200] and Assayer [239]. TrInc uses hardware-provided trusted counters to protect against equivocation attacks. Unlike T-Lease, it uses counters, not timers, and proposes non-standard, albeit minimal, hardware extensions. Assayer relies on the standard TPM hardware, which it leverages to convey end-host information to the network in a trustworthy and efficient manner. These both systems are not designed to secure lease-based protocols.

Pasture [183] provides secure offline data access, allowing a remote party to audit the data access log. As cryptographic keys are used to access the data, Pasture uses TPM for key and log management. Memoir [240] uses a conceptually similar state continuity technique, which relies on TPM in its operation.

Monotonic counters. Another important problem that shares design space with T-Lease is protecting storage systems from rollback attacks. This is typically accomplished using

monotonic counters or trusted disks [185]. Intel SGX SDK contained the implementation of monotonic counters using Intel Management Engine [158]. However, their performance was insufficient for applications. This has influenced the design of systems that use monotonic counters: these systems either cache the counter value on the disk, or exploit workload properties to update the counter asynchronously as proposed in Speicher [76].

ROTE [212] presents an alternative approach that overcomes the performance and security limitations of NVMEM-based monotonic counters using a distributed consensus-based trusted counters, relying on the modern low-latency high-throughput networks to avoid the bottlenecks in incrementing the counters.

Trustworthy timers. Intel has originally provided Trusted Time service through using the Platform Service Enclave, which communicated with Intel ME to read the system time. However, as communication happened in the operating system interfaces, the communication path turned out to be vulnerable to the attacks that employed delaying the messages to and from Intel ME. Due to this vulnerability, this functionality has been removed in the Intel SGX SDK v2.8.

TimeSeal [68] is a system that aims to provide to provide timing primitives to applications in a trustworthy manner on untrusted platforms for cyber-physical or real-time systems. To this end, TimeSeal is using Intel SGX enclaves, but does not rely fully on the vulnerable and coarse-grained SGX trusted time to provide these guarantees. Instead, it uses counting threads inside an SGX enclave that augment the SGX trusted time with fine-grained timing information in a secure manner, which is not vulnerable to malicious scheduling by the operating system, using a software timer thread and a set of threads that interpolate reading of SGX trusted time and the software timer. To prevent the scheduling attacks on the counting threads, TimeSeal uses a carefully-constructed counting policy that guarantees robust and controllable degradation under different scheduling scenarios, combined with time correction mechanisms. The high number of threads and mechanisms employed makes TimeSeal a resource-intensive application, which is justifiable given the guarantees it provides without any network back-up.

Aurora [202] addresses the trustworthiness requirement by using the System Management Interrupts to access the timer in a trustworthy environment (System Management Mode). However, it has high costs because an application is preempted when the SMM software is running. The operating system can also manipulate the time reads by delaying the return path from the SMI interrupt.

Several systems [101, 233, 263] require high-precision low-latency clock to measure cache access time inside enclave to mitigate side-channel attacks, motivated by the initial restriction on the execution of `rdtsc` instruction inside SGX enclaves. To that end, they employ a timer thread that increments a memory location in a tight loop. While OS interrupts can be detected by some systems [101] by timing the code runtime under Intel TSX protection, they are still prone to false positives as the CPU frequency changes due to the power saving features.

Lastly, S-FaaS [65] is a trustworthy serverless platform build using Intel SGX, similar to Clemmys (§6). For trusted CPU time accounting, it measures the duration of enclave time intervals, but uses a timer thread on the sibling hyperthread for this task, which has more overhead than T-Lease. S-Faas ensures that at no point of time the timer thread is running while the worker thread is not running. S-Faas timer thread executes tight loops inside the timer thread under Intel TSX transaction to measure the time intervals. S-Faas does not prevent frequency manipulations, as it is not required by its threat model.

7.7 Discussion

Formal verification. As the verification of the T-Lease protocol was performed without active involvement of the author of this thesis, we have omitted it detailed description in this chapter. Instead, we will provide a telegraphic summary of the verification effort here, referring the reader to the paper for the full details [283].

The verification was performed using TLA+ toolkit and the TLC model checker [301], modelling the main T-Lease scenario with one granter node and multiple holder nodes. The model assumes that each of nodes has a clock with the bounded drift, with relative bounds which we obtained empirically for our CPU frequency verification routine. This drift represents the attacker control over the node's clock when the probability of attack detection is low. The model also includes the exchange of messages between system nodes, and the delivery of intrerrupts which preempt the operation of nodes.

For this model, two system properties were verified: safety and liveness. Safety property states that the main system invariant (a lease at granter is a superset of a lease at holder) holds. Liveness states that in absense of attacker the lease protocol makes forward progress, assuming certain conditions hold, for example that the OS does not deny enclave the service using interrupts.

Verification of the model using TLC has shown that there are no violations of the system properties in absence of an attacker. With an attacker that can manipulate the timer frequency in the range of $\pm 50\%$ (see §7.5.2 (b)), the properties are not violated as long as the lease time at granter is $3\times$ larger than the lease duration at holder.

Compiler support. We currently implement T-Lease as a library. Therefore, the developer has to spend additional effort to instrument timer invocations in the application to use our library calls. This presents a barrier to adoption, especially when the application has to constantly maintain a lease from the granter: in this case, the developers must carefully select the locations where these library calls are necessary, and complex control flow graphs of networking applications make this task challenging.

However, we believe that it is possible to avoid this effort by automatically transforming the calls to `rdtsc` and system time into our library calls, and if necessary, automatically injecting the lease checks into the basic blocks of the application. We plan to implement this transformation as an LLVM compiler pass to transparently benefit existing applications.

Hardware extensions. The design of T-Lease could be simplified using a simple hardware extension to the TSC functionality. For example, if TSC would include a separate read-only register incremented every time the TSC value is modified by the platform operator, our system would be able to provide precise timing for much longer intervals. This extension would reduce the rate of messages in the lease-based systems. Also, if the hardware would expose the TSC frequency to the user-space applications, the timer verification mechanism would not be necessary. Finally, in the case of real-time and cyber-physical applications, the hardware vendor should provide the trusted, unmediated, and non-modifiable real-time clock to the TEE, as in this case the availability of the system becomes increasingly important, and the available network may not satisfy these requirements.

Multiple time sources. Anwar et al. has shown how multiple time sources can be used to maintain the real-world time in a secure fashion [68]. The same approach can be applied to T-Lease, which could potentially reduce the number of network messages. Given that most of the other available time sources have comparably high latency, this approach would work only for the lease terms with longer duration than that currently supported by the T-Lease.

7.8 Conclusion

In this chapter, we have introduced a concept of a trusted lease, a variant of the classical lease primitive that maintains its correctness properties in the presence of a privileged attacker. We have designed and implemented T-Lease—a trusted lease system for Intel SGX enclaves. T-Lease exposes an easy-to-use interface that allows system designers to implement a wide range of trusted distributed lease-based protocols for the untrusted computing infrastructure. To achieve our design goals, T-Lease relies on three core contributions: (a) enclave-interval timer for secure measurement of time intervals which are free from manipulations, (b) a timer frequency verification routine that detects manipulations of TSC speed, and (c) transactional syscall interface for atomic lease state check and resource access. T-Lease implements these abstractions using Intel SGX and Intel TSX ISA extensions. T-Lease protocol has been formally verified. Our evaluation with a wide range of state-of-the-art distributed protocols shows that in most cases T-Lease adds up to 5% overhead, allowing its practical utilization in building trusted distributed systems.

8 Conclusions

8.1 Summary of contributions

In this dissertation, we tackled several problems related to trusted execution environments. We aimed to make Intel SGX-based TEEs practical to use in the cloud, and we achieved our goal: the results of our work show how to protect most of the key components of cloud-based applications with low overhead. We provide a brief summary of the contributions of each of the proposed systems below:

SCONE, or *How to run unmodified POSIX applications inside Intel SGX enclaves?* (§3). SCONE succeeded with this task, allowing us to run a wide range of cloud software inside of the enclave. SCONE overcomes most severe restrictions of Intel SGX on performance and supported features: SCONE uses an asynchronous system call interface and M:N threading to avoid the high enclave entry cost, emulates the minimum necessary amount of system calls inside of the enclave, forwarding the rest to the operating system to provide a comprehensive and efficient interface to the operating system. SCONE uses musl-libc as a foundation, extending it with the necessary OS-like functionality, exhibiting a minimal TCB as a result.

FFQ, or *How to make the SCONE I/O interfaces performant?* (§4). To improve the performance of the SCONE system call interface, we designed a new concurrent queue, called Fast FIFO Queue. The key insight of FFQ is that by exploiting common domain-specific assumptions, it is possible to increase the performance of shared memory communication interfaces. With FFQ, SCONE achieves 5 times higher throughput in system call intensive microbenchmark: we reach this result by designing FFQ as an SPMC queue with wait-free enqueue and lock-free dequeue, and implicit flow control.

ShieldBox, or *How can enclaves interact with devices directly?* (§5). The NFV paradigm has allowed network operators to improve network flexibility and reduce operational costs by replacing the hardware middleboxes with the software ones. As a result, there is a necessity to protect the network functions with trusted execution environments. ShieldBox achieves this goal by using Intel SGX and SCONE. To process the network traffic at line rate, it uses the DPDK framework rather than the OS networking interfaces. To reduce the latency impact of the idle system call threads, ShieldBox uses an untrusted but low-latency on-NIC PTP clock instead of the OS-provided time source and SCONE's asynchronous system calls. These design decisions allow ShieldBox to avoid most SGX-related performance overheads.

Clemmys, or *How to make function startup fast?* (§6). Modern cloud services are moving to

the FaaS architecture to provide users with services that are easier and cheaper to use and to develop. Clemmys is the system that uses Intel SGX to bring confidentiality and integrity to the FaaS paradigm. We identify and solve the main issues that arise: we introduce an architecture that runs only the necessary components inside of the enclave, client and function attestation and secret distribution, and optimize the startup time for network functions using SGXv2 EDMM features. We also establish that the low EPC size is a hard limitation that must be alleviated by Intel before TEE-protected functions can truly be used in practice.

T-Lease, or *How to protect timing requirements in the practical, lease-based distributed systems?* (§7). Distributed systems commonly use leases to implement a wide range of protocols and use-cases. It is challenging to make leases trusted, as they depend on system timing which cannot be protected using cryptography. With T-Lease, we introduce a trusted lease primitive, revisiting one of the Intel SGX restrictions: lack of trusted time source. Unlike ShieldBox, we use the SGXv2-provided untrusted timestamp counter, but after identification of potential attacks, show how it can be used to measure in-enclave time durations in a trustworthy manner. This primitive allows us to construct a trusted lease protocol, which can be applied in a wide range of distributed systems with at most 15% overhead.

8.2 Challenges and future work

Trusted Execution Environments have a long history, which starts with IBM 4758 cryptographic accelerator and industry TPM efforts, and within twenty years have culminated with the general-purpose design adopted by the majority of CPU designs (AMD, ARM, IBM, Intel, RISC-V). However, general-purpose Trusted Execution Environments still have not reached their full potential, and offer rich possibilities for research and industrial development. Below are some of the challenges that merit further investigation.

We envision a future of the cloud where the trust could be established end-to-end, between each component of the system, both high-level and low-level. The research has shown that a clean-slate approach to such systems is necessary: the legacy functionality or oversight has undermined the security of enclaves in the past, and this should not repeat with the newer architectures. Finally, the open and compatible TEEs can be easily modified or extended without losing interoperability with existing trustworthy systems.

Trustworthy TEE design. There remains a question of how to design the TEE application and TEE technologies that are resilient to both microarchitectural and classic, memory- and synchronization-based attacks. Wider use of formal methods for the specification of hardware-software contracts, and for the design of both software and hardware promises to eradicate whole classes of bugs that plague the current TEE-based systems, while runtime mechanisms like Intel MPX or CHERI capabilities could significantly improve the security of code that is currently too costly to formally verify or rewrite in safe programming languages.

Uniform OS-enclave interfaces. Current TEE technologies do not provide a uniform interface to the operating system. While some of them, like AMD SEV, Intel TDX, and IBM SE, present a virtual machine interface to guests and to the host, Intel SGX and RISC-V Keystone enclaves are not integrated into the OS in any useful way. Thus, the user and the platform owner cannot perform management tasks, like listing, stopping, or debugging enclave performance issues using technology-independent tools. Furthermore, enclave startup and communication is exposed to the user code directly, without any OS-level interface that could hide the technology-specific details. As different vendors introduce their TEE technologies,

a common API or even ABI would increase the portability of enclave-enabled software.

Intra-enclave isolation and capabilities As we have seen in the context of FaaS, a promising approach to running functions is to run them in a single enclave, relying on the software sandboxing features for intra-enclave function isolation. However, in this case, it is necessary to protect the system against sandbox escape vulnerabilities. Two approaches to this problem could be hardware-assisted technologies for intra-process isolation, like Intel MPK [288], or trusted shared memory, which is possible with Keystone enclaves. A research direction worthy of attention in this context is the integration of TEE technologies with capability-based architectures like CHERI.

Heterogenous, distributed enclaves. Recent research has shown that on-core TEEs are vulnerable to a wide range of side-channel attacks, which motivates the research into *off-core* TEEs, which run on an isolated CPU with dedicated on-die memory. Additionally, modern SmartNICs have a hardware root of trust, that allows running TEEs on these peripheral devices as well. These developments raise the question the about construction of TEE-protected applications that span several different TEE technologies. The problems of mutual attestation, secure communication channels, and distribution of secrets between more trustworthy (off-core enclaves) and less-trustworthy components (on-core enclaves) all need to be solved.

Enclaves for storage systems: NDP and freshness. In the context of storage, off-core TEEs present additional opportunities for off-core Near-Data Processing: offloading the computations on user data to the storage engine. Another problem in the field of storage that calls for an efficient solution is rollback protection, which currently causes 35-times performance drop in state-of-the-art systems [76].

* * *

I prepared this thesis while working with one of the first commercial trusted execution technology, namely Intel SGX, and was confined to its limitations. For the majority of its lifetime, practical TEE design was limited to the labs of CPU vendors, but the situation has radically changed with the arrival RISC-V architecture, which, in my opinion, will further stimulate the research in this field. I hope that the new generation of researchers will use it to make great progress on all of these issues, bringing us more mature, performant, and secure trusted execution technologies.

Bibliography

[1] Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/. Accessed on 2020-06-15.

[2] Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/. Accessed on 2020-06-15.

[3] Apache OpenWhisk: a serverless, open source cloud platform. https://openwhisk.apache.org. Accessed on 2020-08-27.

[4] Asylo Enclave Framework. https://asylo.dev/. Accessed on 2020-06-19.

[5] AWS Lambda - Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/. Accessed on 2020-08-27.

[6] AWS Nitro Enclaves. https://aws.amazon.com/de/ec2/nitro/nitro-enclaves/. Accessed on 2020-06-19.

[7] Azure Functions Serverless Compute (Microsoft Azure). https://azure.microsoft.com/en-us/services/functions/. Accessed on 2020-08-27.

[8] BearSSL - Constant-Time Crypto. https://www.bearssl.org/constanttime.html. Accessed on 2021-03-08.

[9] Bionic: Android's C library, math library, and dynamic linker. https://android.googlesource.com/platform/bionic. Accessed on 2020-07-02.

[10] Cloudflare Workers. https://workers.cloudflare.com/. Accessed on 2020-10-27.

[11] Diet libc - a libc optimized for small size. https://www.fefe.de/dietlibc/. Accessed on 2020-07-02.

[12] Docker Hub. https://hub.docker.com/. Accessed on 2018-02-05.

[13] Fastly Terrarium Documentation. https://wasm.fastlylabs.com/docs. Accessed on 2020-10-27.

[14] Fission: Serverless Functions for Kubernetes. https://fission.io. Accessed on 2020-08-27.

[15] Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17% in 2020. https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020. Accessed on 2020-01-27.

[16] GitHub - intel/linux-sgx-pcl: Intel(R) Software Guard Extensions Protected Code Loader for Linux* OS. https://github.com/intel/linux-sgx-pcl. Accessed on 2020-11-18.

[17] Google Cloud: Cloud Functions. https://cloud.google.com/functions/. Accessed on 2020-08-27.

[18] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/. Accessed on 2020-08-27.

[19] IBM Cloud. https://www.ibm.com/cloud. Accessed on 2020-06-15.

[20] IBM Cloud Functions. https://www.ibm.com/cloud/functions. Accessed on 2020-08-27.

[21] Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. Accessed on 2020-08-24.

[22] Intel DPDK. http://dpdk.org/. Accessed on 2018-02-05.

[23] Intel Software Guard Extensions Remote Attestation End-to-End Example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example. Accessed on 2018-02-05.

[24] Intel Trust Domain Extensions. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html. Accessed on 2020-08-18.

[25] Intel Xeon E-2278G Processor (16M Cache, 3.40 GHz) Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/193745/intel-xeon-e-2278g-processor-16m-cache-3-40-ghz.html. Accessed on 2020-06-18.

[26] Krustlet: the WebAssembly Kubelet. https://deislabs.io/posts/introducing-krustlet/. Accessed on 2020-10-27.

[27] Kubeless: The Kubernetes Native Serverless Framework. https://kubeless.io. Accessed on 2020-08-27.

[28] Linux End of Year 2019 Statistics. https://phoronix.com/misc/linux-eoy2019/index.html. Accessed on 2020-01-28.

[29] Linux kernel documentation: overview of Amazon Nitro Enclaves. https://git.kernel.org/pub/scm/linux/kernel/git/gregkh/char-misc.git/tree/Documentation/virt/ne_overview.rst?h=e82ed736ad2d2dddf1384fc4c8a0f26021af04fe&id=bf15d79ce142fe1d01eb88bdad96367a3887648c. Accessed on 2020-09-23.

[30] LLVM libc. https://github.com/llvm/llvm-project/commits/master/libc. Accessed on 2020-07-02.

[31] Microsoft Azure Cloud Computing Services. azure.microsoft.com. Accessed on 2020-06-15.

[32] Microsoft Security Response Center: A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. Accessed on 2020-06-19.

[33] musl-cross-make: Simple makefile-based build for musl cross compiler. https://github.com/richfelker/musl-cross-make. Accessed on 2020-07-02.

[34] Ocalls interfaces of Graphene-SGX (source code). https://github.com/oscarlab/graphene/blob/master/Pal/src/host/Linux-SGX/enclave_ocalls.h. Accessed on 2021-02-02.

[35] Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/. Accessed on 2020-06-15.

[36] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com. Accessed on 2020-08-27.

[37] OpenResty - Official Site. https://openresty.org/en/. Accessed on 2021-04-10.

[38] Oracle Database Classic Cloud Service - Get Started. https://docs.oracle.com/en/cloud/paas/database-dbaas-cloud/index.html. Accessed on 2020-06-15.

[39] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2018-02-05.

[40] Samsung Knox. https://www.samsungknox.com/en. Accessed on 2020-06-18.

[41] SAP HANA Cloud Services. https://saphanacloudservices.com/. Accessed on 2020-06-15.

[42] SierraTEE. http://www.sierraware.com/open-source-ARM-TrustZone.html. Accessed on 2020-06-15.

[43] Snort. https://www.snort.org/. Accessed on 2018-02-05.

[44] The Cloud Market: EC2 Statistics. https://thecloudmarket.com/stats. Accessed on 2020-06-23.

[45] The GNU C Library. https://www.gnu.org/software/libc/. Accessed on 2020-07-02.

[46] Transport Layer Development Kit. https://wiki.fd.io/view/TLDK. Accessed on 2018-02-05.

[47] uClibc-ng - Embedded C library. https://uclibc-ng.org/. Accessed on 2020-07-02.

[48] USB Authentication Specification Rev. 1.0 with ECN and Errata through January 7, 2019. https://www.usb.org/document-library/usb-authentication-specification-rev-10-ecn-and-errata-through-january-7-2019. Accessed on 2020-08-24.

[49] USB Security Foundation Specification Rev. 1.0 with ECN and Errata through January 7, 2019. https://www.usb.org/document-library/usb-authentication-specification-rev-10-ecn-and-errata-through-january-7-2019. Accessed on 2020-08-24.

[50] V8 Isolates: Getting started with embedding V8. https://v8.dev/docs/embed. Accessed on 2020-10-11.

[51] Wolf SSL Library. https://www.wolfssl.com/. Accessed on 2018-02-05.

[52] *Oxford English Dictionary (3rd Edition)*. Oxford University Press, 2015. Trust.

[53] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. 2020.

[54] Intel Multi-Buffer Crypto for IPsec Library. https://github.com/intel/intel-ipsec-mb, accessed on 07/12/2018.

[55] vdso(7) - Linux manual page. https://man7.org/linux/man-pages/man7/vdso.7.html, accessed on 10/08/2020.

[56] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283, 2016.

[57] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[58] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 739–753, 2016.

[59] G. Adzic and R. Chatley. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 884–889, New York, NY, USA, 2017. ACM.

[60] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434, 2020.

[61] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, Nov. 2009.

[62] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[63] M. Al-Bassam, A. Sonnino, M. Król, and I. Psaras. Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations. *CoRR*, abs/1805.06411, 2018.

[64] H. Alayli. halayli/lthread: lthread, a multicore enabled coroutine library written in C. `github.com/halayli/lthread`. Accessed on 2021-03-13.

[65] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, pages 185–199, 2019.

[66] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, R. Mortier, M. Koleni, L. Oviedo, M. Migliavacca, and D. McAuley. FLICK: Developing and Running Application-Specific Network Services. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[67] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2012.

[68] F. M. Anwar, L. Garcia, X. Han, and M. B. Srivastava. Securing Time in Untrusted Operating Systems with TimeSeal. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 80–92, 2019.

[69] F. M. Anwar and M. B. Srivastava. Applications and Challenges in Securing Time. In *12th USENIX Workshop on Cyber Security Experimentation and Test, (CSET)*, 2019.

[70] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[71] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 689–703, 2016.

[72] W. Arthur and D. Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[73] R. Avanzi, S. Banik, O. Dunkelman, H. Montaner, P. Ramrakhyani, F. Regazzoni, and A. Sandberg. Protecting Memory Contents on ARM Cores. In *Proceedings of the Real World Crypto Symposium 2020, New York, USA, January 8, 2020*, 2020.

[74] J. Axboe. Efficient IO with io_uring. Accessed on 2021-01-10.

[75] M. Bailleu, D. Dragoti, P. Bhatotia, and C. Fetzer. TEE-Perf: A Profiler for Trusted Execution Environments. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 414–421, 2019.

[76] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 173–190, 2019.

[77] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, 2017. ACM.

[78] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015*, pages 5–16, 2015.

[79] E. Barker and A. Roginsky. Recommendation for Cryptographic Key Generation. Dec 2012.

[80] E. B. Barker, M. Smid, and D. Branstad. A Profile for U. S. Federal Cryptographic Key Management Systems. Oct 2015.

[81] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. High-assurance cryptography software in the spectre era. *IACR Cryptol. ePrint Arch.*, 2020:1104, 2020.

[82] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 14–22, 2019.

[83] A. Baumann, M. Peinado, and G. C. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, 2015.

[84] A. Beaupré. New approaches to network fast paths. `https://lwn.net/Articles/719850/`. Accessed on 2018-02-05.

[85] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[86] K. Bhardwaj, M. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, Scalable and Secure Onloading of Edge Functions Using AirBox. In *IEEE/ACM Symposium on Edge Computing, SEC 2016, Washington, DC, USA, October 27-28, 2016*, pages 14–27, 2016.

[87] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.

[88] D. Bornstein. Dalvik VM Internals. In *Google I/O developer conference*, volume 23, pages 17–30, 2008.

[89] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[90] S. Brenner and R. Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 33–43, 2019.

[91] M. Budiu and C. Dodd. The P416 Programming Language. *Operating Systems Review*, 51(1):5–14, 2017.

[92] R. Buhren, C. Werling, and J. Seifert. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1087–1099, 2019.

[93] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008, 2018.

[94] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 54–72, 2020.

[95] J. V. Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1741–1758, 2019.

[96] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[97] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. H. Katz. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 13–24, 2019.

[98] R. Chandramouli. Security Strategies for Microservices-based Application Systems. Technical report, NIST, 2019.

[99] O. Chang, A. Arya, K. Serebryany, and J. Armour. OSS-Fuzz: Five months later, and rewarding projects. `https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`. Accessed on 2020-06-19.

[100] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS*, 2013.

[101] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 7–18, New York, NY, USA, 2017. ACM.

[102] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[103] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[104] V. Costan, I. A. Lebedev, and S. Devadas. Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture. *Foundations and Trends in Electronic Design Automation*, 11(1-2):1–248, 2017.

[105] M. Coughlin, E. Keller, and E. Wustrow. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks Network Function Virtualization (SDN-NFVSec)*, 2017.

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

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

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

[109] B. Danev, R. J. Masti, G. Karame, and S. Capkun. Enabling secure VM-vTPM migration in private clouds. In *ACSAC*, 2011.

[110] M. David. A single-enqueuer wait-free queue implementation. In *Proceedings of the 18th International Conference on Distributed Computing*, pages 132–143, Berlin, Heidelberg, 2004.

[111] A. Dragojevic, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP'15)*. ACM – Association for Computing Machinery, October 2015.

[112] D. Du, Z. Hua, Y. Xia, B. Zang, and H. Chen. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 671–684, 2019.

[113] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481, 2020.

[114] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed. In *Proceedings of the 2019 ACM SIGSAC*

Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019, pages 2351–2367, 2019.

[115] A. Dunkels. Full TCP/IP for 8-Bit Architectures. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services, MobiSys 2003, San Francisco, CA, USA, May 5-8, 2003*, 2003.

[116] V. Duta, E. van der Kouwe, H. Bos, and C. Giuffrida. PIBE: Practical Kernel Control-flow Hardening with Profile-guided Indirect Branch Elimination. In *ASPLOS*, Apr. 2021.

[117] Eta Labs. Comparison of C/POSIX standard library implementations for Linux. `http://www.etalabs.net/compare_libcs.html`. Accessed on 2020-07-02.

[118] P. Fatourou and N. D. Kallimanis. A Highly-efficient Wait-free Universal Construction. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'11, pages 325–334, New York, NY, USA, 2011.

[119] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, pages 257–266, New York, NY, USA, 2012.

[120] R. Felker. Musl Libc. `https://www.musl-libc.org`, 2016.

[121] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305, 2017.

[122] C. Fetzer. Building Critical Applications using Microservices. *CoRR*, abs/1908.08744, 2019.

[123] C. Fetzer and F. Cristian. A Highly Available Local Leader Election Service. *IEEE Trans. Softw. Eng.*, 25(5):603–618, Sept. 1999.

[124] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures, 2000.

[125] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, Aug. 2004.

[126] FSF. *GCC 6.1 Manual*. `https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/`.

[127] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*, pages 261–266, 2019.

[128] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric Computing: Vision and Challenges. *SIGCOMM CCR*, 2015.

[129] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 193–206, 2003.

[130] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[131] S. Ghosh, L. S. Kida, S. J. Desai, and R. Lal. A >100 Gbps Inline AES-GCM Hardware Engine and Protected DMA Transfers between SGX Enclave and FPGA Accelerator Device. *IACR Cryptol. ePrint Arch.*, 2020:178, 2020.

[132] J. Giacomoni. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *In PPoPP'08: Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[133] GlobalPlatform Technology. TEE System Architecture Version 1.1.0.10 (Target v1.2), 2018.

[134] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*, pages 123–135, 2019.

[135] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P. Aublin, P. Costa, C. Fetzer, P. Felber, P. R. Pietzuch, and R. Kapitza. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 386–397, 2018.

[136] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[137] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, Nov. 1989.

[138] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. L. Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 502–514, 2020.

[139] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 217–233, 2017.

[140] R. Guerzoni. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. Issue 1. Oct. 2012.

[141] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[142] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200, 2017.

[143] M. Hamburg, P. Kocher, and M. E. Marson. Analysis of Intel's Ivy Bridge digital random number generator. Technical report, Cryptography Research, Inc., San Francisco, CA 94105, 2012.

[144] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network Function Virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[145] J. Han, S. Kim, J. Ha, and D. Han. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet)*, 2017.

[146] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 135–148, 2012.

[147] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 124–133, New York, NY, USA, 2017. ACM.

[148] M. Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, Jan. 1991.

[149] M. Herlihy. The Art of Multiprocessor Programming. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC'06, pages 1–2, New York, NY, USA, 2006.

[150] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[151] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and Enhancing in Situ System Observability for Failure Detection. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 1–16, Berkeley, CA, USA, 2018. USENIX Association.

[152] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[153] F. Hupfeld, B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Martí, and J. Malo. FaTLease: scalable fault-tolerant lease negotiation with Paxos. *Cluster Computing*, 12(2):175–188, 2009.

[154] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual; Chapter 16: Programming with Intel Transacactional Synchronization Extensions*.

[155] Intel Corporation. *IA-PC HPET (High Precision Event Timers)*, October 2004.

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

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

[158] Intel Corporation. *Intel® Software Guard Extensions SDK for Linux OS*, November 2017.

[159] Intel Corporation. *Intel Digital Random Number Generator (DRNG) Software Implementation Guide, Revision 2.1*, October 2018.

[160] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, May 2018.

[161] Intel Corporation. *PCI Express Device Security Enhancements, version 0.71*, September 2018.

[162] Intel Corporation. *Intel Architecture Memory Encryption Technologies Specification*, 2019.

[163] Intel Corporation. White Paper: Intel Trust Domain Extensions. `https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf`, August 2020.

[164] Intel Corporation. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. `https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf`, accessed on 2020-08-27.

[165] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[166] W. Jansen and T. Grance. Guidelines on Security and Privacy in Public Cloud Computing. Technical report, NIST, 2011.

[167] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[168] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. 2016.

[169] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR*, abs/1902.03383, 2019.

[170] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2015.

[171] A. Kantee. Rump File Systems: Kernel Code Reborn. In *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*, 2009.

[172] D. Kaplan. Protecting VM Register State With SEV-ES. 2020.

[173] G. P. Katsikas, G. Q. Maguire Jr., and D. Kostic. Profiling and Accelerating Commodity NFV Service Chains with SCC. *Journal of Systems and Software*, 2017.

[174] B. Kauer, P. Veríssimo, and A. N. Bessani. Recursive virtual machines for advanced security mechanisms. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011), Hong Kong, China, June 27-30, 2011*, pages 117–122, 2011.

[175] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[176] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[177] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 213–225, New York, NY, USA, 1991. ACM.

[178] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*, pages 238–247, 1986.

[179] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. *CoRR*, abs/1801.05863, 2018.

[180] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19, 2019.

[181] E. Kohler. Click Modular Router Delay Example. `https://raw.githubusercontent.com/kohler/click/master/conf/delay.click`. Accessed on 2020-08-24.

[182] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[183] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure Offline Data Access Using Commodity Trusted Hardware. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 321–334, 2012.

[184] K. Kourtis, N. Ioannou, and I. Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 1–15, 2019.

[185] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 25:1–25:17, 2018.

[186] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[187] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. *CoRR*, abs/1902.04413, 2019.

[188] D. Kuvaiskii, S. Chakrabarti, and M. Vij. Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX). *CoRR*, abs/1802.00508, 2018.

[189] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer. HAFT: hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 25:1–25:17, 2016.

[190] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 205–221, 2017.

[191] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung. DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 183–191, 2020.

[192] L. Lamport. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983.

[193] B. W. Lampson. How to Build a Highly Available System Using Consensus. In *Distributed Algorithms, 10th International Workshop, WDAG '96, Bologna, Italy, October 9-11, 1996, Proceedings*, pages 1–17, 1996.

[194] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[195] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi. CliMB: Enabling Network Function Composition with Click Middleboxes. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox)*, 2016.

[196] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015.

[197] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 38:1–38:16, 2020.

[198] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[199] D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 217–234, 2020.

[200] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 1–14, 2009.

[201] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[202] H. Liang, M. Li, Q. Zhang, Y. Yu, L. Jiang, and Y. Chen. Aurora: Providing Trusted System Services for Enclaves On an Untrusted System. *CoRR*, abs/1802.03530, 2018.

[203] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 285–298, 2017.

[204] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990, 2018.

[205] H. Lu, M. Matz, M. Girkar, J. Hubička, A. Jaeger, and M. Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*, June 2017.

[206] V. Maffione, G. Lettieri, and L. Rizzo. Cache-aware design of general-purpose Single-Producer-Single-Consumer queues. *Softw. Pract. Exp.*, 49(5):748–779, 2019.

[207] V. Maffione, L. Rizzo, and G. Lettieri. Flexible virtual machine networking using netmap passthrough. In *IEEE International Symposium on Local and Metropolitan Area Networks, LANMAN 2016, Rome, Italy, June 13-15, 2016*, pages 1–6, 2016.

[208] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[209] M. Majkowski. Cloudflare blog: Kernel Bypass. `https://blog.cloudflare.com/kernel-bypass/`. Accessed on 2020-08-19.

[210] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the*

26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 218–233, 2017.

[211] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[212] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. M. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1289–1306, 2017.

[213] D. Mazières. A Toolkit for User-Level File Systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 261–274, Berkeley, CA, USA, 2001. USENIX Association.

[214] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 259–270, 1993.

[215] S. McConnell. *Code complete - a practical handbook of software construction, 2nd Edition*. Microsoft Press, 2004.

[216] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 143–158, 2010.

[217] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 10:1–10:9, New York, NY, USA, 2016. ACM.

[218] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[219] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, Mar. 2014.

[220] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, pages 267–275, New York, NY, USA, 1996.

[221] K. Mitropoulou, V. Porpodas, X. Zhang, and T. M. Jones. Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS'16, pages 18:1–18:12, New York, NY, USA, 2016.

[222] A. Mogage, R. Pires, V. C. Craciun, E. Onica, and P. Felber. Supply chain malware targets SGX: Take care of what you sign. *CoRR*, abs/1907.05096, 2019.

[223] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 22:1–22:13, New York, NY, USA, 2014. ACM.

[224] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, pages 1:1–1:6, 2018.

[225] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'13, pages 103–112, New York, NY, USA, 2013.

[226] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[227] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[228] J. Nider, M. Rapoport, and J. Bottomley. Address space isolation in the Linux kernel. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, page 194, 2019.

[229] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398, 2013.

[230] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[231] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *POMACS*, 2(2):28:1–28:30, 2018.

[232] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, and C. Fetzer. Fex: A Software Systems Evaluator. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 543–550, 2017.

[233] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 227–240, 2018.

[234] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *CoRR*, abs/1805.08506, 2018.

[235] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1481–1498, 2020.

[236] V. A. Olteanu and C. Raiciu. Efficiently Migrating Stateful Middleboxes. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[237] OMTP. Advanced Trusted Environment, 2009.

[238] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[239] B. Parno. *Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers (dissertation, updated version)*, volume 2 of *ACM Books*. ACM / Morgan & Claypool, 2014.

[240] B. Parno, J. R. Lorch, J. R. Douceur, J. W. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 379–394, 2011.

[241] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[242] R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Arxiv*, 2017.

[243] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1039–1056, 2020.

[244] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 201–216, 2018.

[245] A. Pop and A. Cohen. A Stream-computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC'11, pages 5–14, New York, NY, USA, 2011.

[246] T. Preud'Homme, J. Sopena, G. Thomas, and B. Folliot. An Improvement of OpenMP Pipeline Parallelism with the BatchQueue Algorithm. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 348–355, Dec 2012.

[247] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *CoRR*, abs/1908.11143, 2019.

[248] R. Rajesh, K. B. Ramia, and M. Kulkarni. Integration of LwIP Stack over Intel(R) DPDK for High Throughput Packet Delivery to Applications. In *2014 Fifth International Symposium on Electronic System Design, Surathkal, Mangalore, India, December 15-17, 2014*, pages 130–134, 2014.

[249] A. Randal. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *CoRR*, abs/1904.12226, 2019.

[250] A. Randazzo and I. Tinnirello. Kata containers: An emerging architecture for enabling MEC services in fast and secure way. In *Sixth International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019, Granada, Spain, October 22-25, 2019*, pages 209–214, 2019.

[251] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 13–24, 2007.

[252] Redis. `http://redis.io`, 2016.

[253] W. Reese. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal*, Sept. 2008.

[254] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 101–112, 2012.

[255] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[256] J. Rutkowska. Intel x86 considered harmful. `https://blog.invisiblethings.org/2015/10/27/x86_harmful.html`, 2015. Accessed on 2019-10-09.

[257] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 57–64, 2015.

[258] J. Sakkinen. LKML: [PATCH v35 23/24] docs: x86/sgx: Document SGX micro architecture and kernel internals. `https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg2224088.html`. Accessed on 2020-08-17.

[259] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[260] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[261] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. 2018.

[262] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[263] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity*, 3(1):2, 2020.

[264] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *In the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[265] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[266] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[267] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[268] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[269] S. Shillaker and P. R. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 419–433, 2020.

[270] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 317–328, 2016.

[271] S. Shinde, J. Cui, S. Sen, P. Yuan, and P. Saxena. Binary Compatibility For SGX Enclaves. *CoRR*, abs/2009.01144, 2020.

[272] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[273] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena. BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 523–540, 2020.

[274] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10, 2010.

[275] R. Sinha, S. Rajamani, S. A. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *The ACM Conference on Computer and Communications Security (CCS)*. ACM Association for Computing Machinery, October 2015.

[276] S. W. Smith and S. H. Weingart. Building a high-performance, programmable secure coprocessor. *Comput. Networks*, 31(8):831–860, 1999.

[277] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *OSDI*, 2010.

[278] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[279] N. Suneja. ScyllaDB optimizes database architecture to maximize hardware performance. *IEEE Softw.*, 36(4):96–100, 2019.

[280] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 199–212, 2018.

[281] H. Tian, Y. Zhang, C. Xing, and S. Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*, pages 35–44. ACM, 2017.

[282] S. Tople, S. Park, M. S. Kang, and P. Saxena. VeriCount: Verifiable Resource Accounting Using Hardware and Software Isolation. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, pages 657–677, 2018.

[283] B. Trach, R. Faqeh, O. Oleksenko, W. Ozga, P. Bhatotia, and C. Fetzer. T-lease: A trusted lease primitive for distributed systems. *CoRR*, abs/2101.06485, 2021.

[284] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 2:1–2:14, New York, NY, USA, 2018. ACM.

[285] C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 16:1–16:16, 2016.

[286] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *EuroSys*, 2014.

[287] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658, Berkeley, CA, USA, 2017. USENIX Association.

[288] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1221–1238, 2019.

[289] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune. Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me? In *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*, pages 159–178, 2012.

[290] M. A. M. Vieira, M. S. Castanho, R. Pacifico, E. R. da Silva Santos, E. P. M. C. Júnior, and L. F. M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.*, 53(1):16:1–16:36, 2020.

[291] D. Vyukov. Bounded MPMC queue - 1024cores. `http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue`. Accessed on 2020-07-03.

[292] J. Wang, K. Zhang, X. Tang, and B. Hua. B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication. *International Journal of Parallel Programming*, 41(1):137–159, 2013.

[293] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146, 2018.

[294] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report*, 2018.

[295] D. Williams, R. Koller, M. Lucina, and N. Prakash. Unikernels as Processes. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 199–211, 2018.

[296] W. Wu, K. He, and A. Akella. PerfSight: Performance Diagnosis for Software Dataplanes. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*, 2015.

[297] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[298] C. Yang and J. Mellor-Crummey. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'16, pages 16:1–16:13, New York, NY, USA, 2016.

[299] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed Lock Management with RDMA: Decentralization Without Starvation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1571–1586, New York, NY, USA, 2018. ACM.

[300] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 163–174, New York, NY, USA, 1999. ACM.

[301] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA$^+$ Specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 54–66, 1999.

[302] Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena. Elasticlave: An Efficient Memory Model for Enclaves. *CoRR*, abs/2010.08440, 2020.

[303] K. Zetter. NSA Hacker Chief Explains How to Keep Him Out of Your System. *Wired*, Jan. 2016.

[304] T. Zhang, D. Xie, F. Li, and R. Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 1–12, 2019.

[305] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.