Keystone: An Open Framework for Architecting Trusted Execution Environments

Dayeol Lee
dayeol@berkeley.edu
UC Berkeley

David Kohlbrenner
dkohlbre@berkeley.edu
UC Berkeley

Shweta Shinde
shwetas@berkeley.edu
UC Berkeley

Krste Asanović
krste@berkeley.edu
UC Berkeley

Dawn Song
dawnsong@berkeley.edu
UC Berkeley

Abstract

Trusted execution environments (TEEs) see rising use in devices from embedded sensors to cloud servers and encompass a range of cost, power constraints, and security threat model choices. On the other hand, each of the current vendor-specific TEEs makes a fixed set of trade-offs with little room for customization. We present Keystone—the first open-source framework for building customized TEEs. Keystone uses simple abstractions provided by the hardware such as memory isolation and a programmable layer underneath untrusted components (e.g., OS). We build reusable TEE core primitives from these abstractions while allowing platform-specific modifications and flexible feature choices. We showcase how Keystone-based TEEs run on unmodified RISC-V hardware and demonstrate the strengths of our design in terms of security, TCB size, execution of a range of benchmarks, applications, kernels, and deployment models.

CCS Concepts: • **Security and privacy** → **Trusted computing**; *Hardware security implementation*; *Software and application security.*

Keywords: Trusted Execution Environment, Hardware Enclave, Secure Enclave, RISC-V, Memory Isolation, Side-Channel Attack, Hardware Root of Trust, Open Source

ACM Reference Format:
Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387532

1 Introduction

The last decade has seen the proliferation of trusted execution environments (TEEs) to protect sensitive code and data. All major CPU vendors have rolled out their TEEs (e.g., ARM TrustZone, Intel SGX, and AMD SEV) to provide a secure execution environment, commonly referred to as an *enclave* [1, 51, 64]. TEEs have use-cases in diverse deployment environments ranging from cloud servers, mobile phones, ISPs, IoT devices, sensors, and hardware tokens.

Unfortunately, each vendor TEE enables only a small portion of the possible design space across threat models, hardware requirements, resource management, porting effort, and feature compatibility. When a cloud provider or software developer chooses a target hardware platform they are locked into the respective TEE design limitations regardless of their actual application needs. Constraints breed creativity, giving rise to significant research effort in working around these limits. For example, Intel SGXv1 [64] requires statically sized enclaves, lacks secure I/O and syscall support, and is vulnerable to significant side-channels [35]. Thus, to execute arbitrary applications, the systems built on SGXv1 have inflated the Trusted Computing Base (TCB) and are forced to implement complex workarounds [18, 22, 31]. As only Intel can make changes to the inherent design trade-offs in SGX, users had to wait for changes like dynamic resizing of enclave virtual memory in SGXv2 [63]. Unsurprisingly, these and other similar restriction have led to a proliferation of new TEEs on other ISAs (e.g., OpenSPARC [30], RISC-V [36, 80]). However, each such redesign requires considerable effort and only provides another fixed design point.

We advocate that the hardware should provide security *primitives* instead of point-wise solutions and in this paper leverage RISC-V's primitives to construct highly customizable TEEs. We can draw an analogy with the move from traditional networking solutions to Software Defined Networking (SDN), where exposing the packet forwarding primitives to the software has led to far more novel designs and research. Such a paradigm shift in TEEs will pave the way for low-cost use-case customization. It will allow the features and the security model to be tuned for each hardware platform and

EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6882-7/20/04.
https://doi.org/10.1145/3342195.3387532

use-case from a set of common software components, drawing on ideas from modular kernel concepts [40, 60, 78]. This motivates the need for *Customizable TEEs*—an abstraction that allows entities that create the hardware, operate it, and develop applications to configure and deploy various TEE designs from the same base. Customizable TEEs promise independent exploration of gaps/trade-offs in existing designs, quick prototyping of new feature requirements, a shorter turn-around time for fixes, adaptation to threat models, and usage-specific deployment.

For realizing this vision, our first observation is the need for a highly programmable trusted layer below the *untrusted* OS. Second, we must decouple our isolation mechanisms from decisions of resource management, virtualization, and trust boundaries. We note that a hypervisor solution results in a trusted layer with a mix of security and virtualization responsibilities, thus complicating the most critical component. Similarly, firmware and micro-code are not programmable to a degree that satisfies our requirements. These two requirements ensure we avoid the mistake of using hardware with a separation mechanism encumbered with a static boundary between what is trusted and untrusted. Lastly, we draw inspiration from proliferation of commercial (c.f. Intel SGX, TrustZone) and non-commercial TEEs (c.f. Sanctum [36], Komodo [41]) which demonstrate the need for a common, portable software base adaptable to ever-changing hardware capabilities and use-case demands.

To this end, we propose Keystone—the first open-source framework for building customized TEEs. We built Keystone on unmodified RISC-V using its standard specifications [17] for *physical memory protection* (PMP)—a primitive which allows the programmable *machine mode* underneath the OS in RISC-V to specify arbitrary protections on physical memory regions. We use this machine mode to execute a trusted *security monitor (SM)* to provide security boundaries without needing to perform any resource management. Critically, each enclave operates in its own isolated physical memory region and has its own supervisor-mode *runtime* (RT) component to manage the virtual memory of the enclave and more. With this novel design, any enclave-specific functionality can be implemented cleanly by its RT while the SM manages hardware-enforced guarantees. An enclave's RT implements only the required functionality, communicates with the SM, mediates communication with the host via shared memory, and services the *enclave user-mode application (eapp).*

Our choice of RISC-V and the logical separation between SM and RT allows hardware manufacturers, cloud providers, and application developers to configure various design choices such as TCB, threat models, workloads, and TEE functionality. Specifically, Keystone's SM uses hardware primitives to provide in-built support for TEE guarantees such as secure boot, memory isolation, and attestation. The RT then provides functionality modules for system call interfaces, standard `libc` support, in-enclave virtual memory management, self-paging, and more inside the enclave. For strengthening the security, our SM leverages any available configurable hardware to compose additional security mechanisms. We demonstrate the potential of this with a highly configurable cache controller to, in concert with PMP, transparently defend against physical adversaries and cache side-channels.

We built Keystone, the SM, two RTs (our native RT—Eyrie—and an off-the-shelf microkernel seL4 [55]), and several modules which together allow enclave-bound user applications to selectively configure and use the above features (Figure 1). We extensively benchmark Keystone on 4 suites with varying workloads: RV8, IOZone, CoreMark, and Beebs. We showcase use-case studies where Keystone can be used for secure machine learning (Torch and FANN frameworks) and cryptographic tasks (`libsodium`) on embedded devices and cloud servers. Lastly, we test Keystone on different RISC-V systems: the HiFive Freedom Unleashed, 3 in-order cores and 1 out-of-order core via FPGA, and a QEMU emulation—all without modification. Keystone is fully open-source.

Contributions. We make the following contributions:

- *Customizable TEEs.* We define a new paradigm wherein the hardware manufacturer, hardware operator, and the enclave programmer can tailor the TEE design.
- *Keystone Framework.* We present the first framework to configure, build, and instantiate customized TEEs. Our principled way of ensuring modularity in Keystone allows us to customize the design dimensions of TEE instances as per the requirements.
- *Open-source Implementation.* We demonstrate advantages of different Keystone TEE configurations that are tailored for minimizing the TCB, adapting to threat models, using hardware features, handling workloads, or providing rich functionality without any microarchitectural changes. A typical Keystone instantiated TEE design adds a total TCB of 12-15 K lines of code (LoC) to an enclave-bound application, of which the SM consists of only 1.6 KLoC added by Keystone.
- *Benchmarking & Real-world Applications.* We evaluate Keystone on 4 benchmarks: CoreMark, Beebs, and RV8 (< 1% overhead), and IOZone (40%). We demonstrate real-world machine learning workloads with Torch in Eyrie (7.35%), FANN (0.36%) with seL4, and a Keystone-native secure remote computation application. Finally, we demonstrate defenses against physical adversaries with memory encryption and cache side-channels.

2 A Common Base for Diverse TEEs

2.1 Background: Commercial TEEs

Current widely-used TEE systems cater to specific and valuable use-cases but occupy only a small part of the wide design space (see Appendix A). Consider the case of a heavy server workload (databases, ML inference, etc.) running in an untrusted cloud environment. One option is an Intel SGX-based

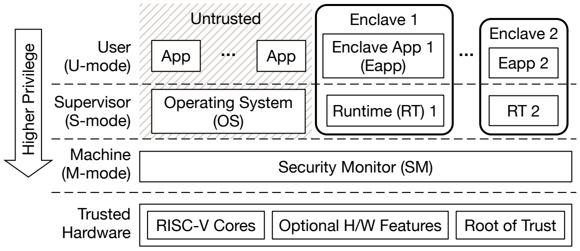


Figure 1. Keystone system with host processes, untrusted OS, security monitor, and multiple enclaves (each with runtime and eapp)

solution which has a large software stack [18, 22, 31] to extend the supported features. On the other hand, an AMD SEV-based solution isolates a full VM with a large TCB. If one wants additional defenses against side-channels it adds further user-space software mechanisms for both cases. If we consider edge-sensors or IoT applications, the available solutions are TrustZone based. While more flexible than SGX or SEV, TrustZone supports only a single hardware-enforced isolated domain called the Secure World. Any further isolation needs multiplexing between secure applications via software-based Secure World OS solutions [12]. Thus, irrespective of the TEE, developers often compromise their requirements (e.g., resort to a large TCB solution, one isolation domain) or build their custom design. One such emerging direction is to use a thin layer of trusted software, similar to a reference monitor in kernel designs. These designs protect against a strong adversary and provide better compatibility while maintaining a low TCB. Several proposals in this area have demonstrated the feasibility of this approach. Sanctum uses a series of modifications to hardware to construct user-space enclaves for RISC-V. Komodo takes this concept further and provides a verified monitor that executes on top of ARM's TrustZone. While these systems inherit the limitations of their underlying designs (e.g., hardware changes or only two security domains in TrustZone), monitor-based TEEs are a very promising direction.

2.2 Customizable TEEs

We call our model ***customizable TEEs***. It uses a common software framework to assemble a specialized TEE specific to the use-case with multiple stakeholders' inputs. The hardware manufacturer is only required to provide basic primitives. Realizing a specific TEE instance involves the platform provider's choice of the hardware interface, the trust model, and the enclave programmer's feature requirements. The entities offload their choices to a framework that composes the required modules to instantiate a specialized TEE.

A motivation for customizable TEEs is that the threat model may differ depending on the use case, the application or the hardware platform. Even on the same platform with the same SM, different applications may operate under differing threat models. For this reason, we allow each enclave to specifiy its configuration of security features. Consider a simple IoT sensor platform that signs measurements for authenticity guarantees and an adversary using a cache occupancy side-channel. In this case, the sensor driver must be protected and requires runtime memory integrity, but not memory confidentiality. The signing process requires both memory integrity and confidentiality. Thus, a possible configuration would be to have the cryptographic library operate with a private cache partition enclave while the driver may operate in a basic isolated enclave. An appropriate SM mechanism (e.g. mailboxes) can ensure authenticated communication between these two enclaves. An adversary using a cache occupancy side-channel against the driver learns only the public measurements, and cannot learn anything about the cryptographic library. By allowing each enclave to specify and deploy its own defenses, we can optimize our use of the available resources (in this case, limited private cache space) and expensive security mechanisms.

The existing commercial TEE systems offer inflexible threat models linked to the respective hardware platform. Notably, Intel's SGX [64] does not support any configuration of its memory protection systems as would be desirable for use cases not requiring expensive memory encryption. On the other hand, while offering some software and hardware customization, ARM's TrustZone provides an inferior substrate to build a modular TEE. Core to TrustZone's design is the concept of only two security domains. A TrustZone TEE implementing multiple enclaves must use the memory management unit (MMU) for further isolation. This fundamentally limits what operations enclaves can be allowed to perform and limits enclaves to user-mode. This limitation naturally extends to all TEE systems built using TrustZone as a base like Komodo. On the hardware side, TrustZone relies on system-wide bus-address filters (e.g., the TZC-400) to separate secure from insecure DRAM partitions, whereas RISC-V provides per-hardware-thread views of physical memory via machine-mode and PMP registers. Using RISC-V thus allows multiple concurrent and potentially multi-threaded enclaves to access disjoint memory partitions while also opening up supervisor-mode and the MMU for enclave use. This allows an enclave to contain either a lightweight or even a full supervisor-mode OS as we demonstrate.

Keystone requires no changes to CPU cores, memory controllers, etc. A secure hardware platform supporting Keystone requires: a device-specific secret key visible only to the trusted boot process, a hardware source of randomness, and a trusted boot process. Key provisioning [15] is an orthogonal problem. For this paper, we assume a simple manufacturer provisioned key.

2.3 Entities in TEE Lifecycle

We define five logical entities in customizable TEEs:

Hardware manufacturer designs and fabricates RISC-V hardware including relevant IP for trusted boot.

Keystone platform provider purchases manufactured hardware; operates the hardware; makes it available for use to its customers; configures the SM.

Keystone programmer develops Keystone software components including SM, RT, and eapps; we refer to the respective programmers who develop these specific components.

Keystone user chooses a Keystone configuration of RT and an eapp. They instantiate an enclave which can execute on hardware provisioned by the Keystone platform provider.

Eapp user interacts with the eapp executing in an enclave on the TEE instantiated using Keystone.

In real-world deployments, a single entity can perform multiple roles. For example, consider Acme Corp. hosts their website on an Apache webserver executing on Bar Corp. manufactured hardware in a Keystone-based enclave hosted on Cloud Corp. cloud service. In this scenario, Bar will be the Hardware manufacturer; Cloud will be a Keystone platform provider and can be an RT programmer and SM programmer; Apache developers will be eapp programmer; Acme Corp. will be Keystone user, and; the person who uses the website will be the eapp user.

3 Keystone Overview

We designed and built Keystone on RISC-V. RISC-V is an open ISA with multiple open-source core implementations [19, 29]. It currently supports up to four privilege modes: U-mode (user) for user-space processes, S-mode (supervisor) for the kernel, H-mode (hypervisor) for the hypervisor, and M-mode (machine) which directly accesses physical resources (e.g., interrupts, memory, devices). At the time of writing, H-mode (hypervisor) is not included in the standard specification. Keystone will also be able to support hypervisor-level isolation when H-mode becomes available.

3.1 Design Principles

We design customizable TEEs with maximum degrees of freedom and minimum effort using the following principles.

Leverage programmable layer and isolation primitives below the untrusted code. We design a reference monitor style *security monitor* (SM) to enforce TEE guarantees on the platform using four properties of M-mode: (a) it is programmable by platform providers, (b) it meets our needs for a minimal highest privilege mode, (c) it controls hardware delegation of the interrupts and exceptions in the system, and (d) M-mode's control of RISC-V's Physical Memory Protection (PMP) standard [17] enables isolation of memory-mapped control features at runtime.

Decouple the resource management and security checks. The SM enforces security policies with minimal code at the highest privilege. It has few non-security responsibilities. This keeps the TCB low and allows it to present clean abstractions. Our S-mode runtime (RT) and U-mode enclave application (eapp) both reside in enclave address space and are isolated from the untrusted OS or other user applications. The RT manages the lifecycle of the user code executing in the enclave, manages memory, services syscalls, etc. For communication with the SM, the RT uses a limited set of API functions via the RISC-V supervisor binary interface (SBI) to exit or pause the enclave (Table 1) as well as request SM operations on behalf of the eapp (e.g., attestation). Each enclave instance may choose its own RT which is never shared with any other enclaves.

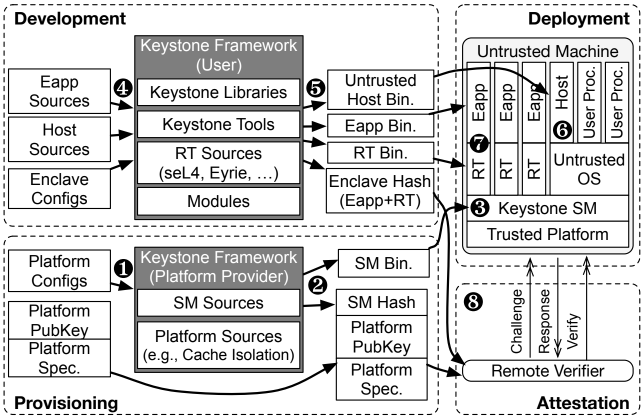


Figure 2. Keystone End-to-end Overview. ❶ Platform provider configures the SM. ❷ Keystone compiles and generates the SM boot image. ❸ Platform provider deploys the SM. ❹ Developer writes an eapp, configures the enclave. ❺ Keystone builds the binaries, computes measurements. ❻ Untrusted host binary is deployed to the machine. ❼ Host deploys the RT, the eapp, and initiates the enclave creation. ❽ Remote verifier can attest based on known platform specifications, keys, and SM/enclave measurements.

Design modular layers. Keystone uses modularity (SM, RT, eapp) to support a variety of workloads. It frees Keystone platform providers and Keystone programmers from retrofitting their requirements and legacy applications into an existing TEE design. Each layer is independent, provides a security-aware abstraction to the layers above it, enforces guarantees which can be easily checked by the lower layers, and is compatible with existing notions of privilege.

Allow fine-grained TCB configuration. Keystone can instantiate TEEs with the minimal TCB for given specific use-cases. The enclave programmer can further optimize the TCB via RT choice and eapp libraries using existing user/kernel privilege separation. For example, if the eapp does not need libc support or dynamic memory management, Keystone will not include them in the enclave.

3.2 Keystone Enclave Workflow

Figure 2 details the steps from Keystone provisioning to eapp deployment. The platform provider instantiates a SM with a proper hardware specification and security extenstions that bring additional isolation guarantees such as cache partitioning. Independently, the enclave developers use Keystone tools and libraries to write eapps and RT with rich features such as virtual memory management and system calls. The RT may use available SM SBI call, but they do not change the isolation guarantees that the SM enforces.

3.3 Writing eapps

Keystone supports 3 ways of writing enclave applications as: (a) standalone Keystone-native eapps, (b) un-modified RISC-V binaries with RT support, or (c) partitioned applications running selected parts in the enclave. Future work will allow Keystone to operate as a backend for cross-enclave SDKs (e.g., OpenEnclave [11], Asylo [67]) to allow for a wide variety of programming models. In sections 7.4, 7.3 we demonstrate un-modified RISC-V binaries and a manual partitioning.

3.4 Threat Model

The Keystone framework trusts the PMP specification as well as the PMP and RISC-V hardware implementation to be bug-free. The Keystone user trusts the SM only after verifying if the SM measurement is correct, signed by trusted hardware, and has the expected version. The SM only trusts the hardware, the host trusts the SM, the RT trusts the SM, the eapp trusts the SM and the RT.

Keystone can operate under diverse threat models, each requiring different defense mechanisms. For this reason, we outline all relevant attackers for Keystone. We allow the selection of a sub-set of these attackers based on the scenario. For example, if the user is deploying TEEs in their private data centers or home appliances, a physical attacker may not be a realistic threat and Keystone can be configured to operate without physical adversary protections.

Attacker Models. Keystone protects the confidentiality and integrity of all enclave code and data at all times after creation. We define four classes of attackers who aim to compromise our security guarantees:

A ***physical attacker*** A_{Phy} can intercept, modify, or replay signals that leave the chip package. We assume that the physical attacker does not affect the components inside the chip package. A_{Phy_C} is for confidentiality, A_{Phy_I} is for integrity.

A ***software attacker*** A_{SW} can control host applications, the untrusted OS, network communications, launch adversarial enclaves, arbitrarily modify any memory not protected by the TEE, and add/drop/replay enclave messages.

A ***side-channel attacker*** A_{SC} can glean information by observing interactions between the trusted and the untrusted components via the cache side channel (A_{Cache}), the timing side channel (A_{Time}) or the controlled channel (A_{Cntrl}).

A ***denial-of-service attacker*** A_{DoS} can take down the enclave or the host OS. Keystone allows the OS to DoS enclaves as the OS can refuse services to user applications at any time.

Scope. Keystone currently has no meaningful mechanisms to protect against speculative execution attacks [27, 56]. Existing and future defenses against this class of attacks can be retrofitted into Keystone [24, 91]. Keystone does not natively protect enclaves or the SM against timing side-channel attacks. Programmers should use existing software solutions to mask timing channels [42] and hardware manufacturers can supply timing side-channel resistant hardware [54]. Side-channel attacks with off-chip components (e.g., memory bus [59]) are also out-of-scope of this paper and they can be orthogonally mitigated by oblivious memory. The SM exposes a limited API (i.e., SBI) to the host OS and the enclave. We do not provide non-interference guarantees for this API [41]. Similarly, the RT can optionally perform untrusted system calls into the host OS. We assume that the RT and the eapp have sufficient checks in place to detect Iago attacks via this untrusted interface [31, 73, 82]. We assume that the SM, RT, and eapp are bug-free. This is a strong assumption but can be partially achieved with formal verification [41, 68].

Caller	SM SBI	Description
OS	create	Validate, and measure the enclave
	run	Start enclave and boot RT
	resume	Resume enclave execution
	destroy	Clean & release enclave memory
RT	stop	Pause enclave execution
	exit	Terminate the enclave
	attest	Get a signed attestation report
	random	Get secure random values
OS & RT	extension*	Platform-specific functions

Table 1. The SBI functions the SM provides, *SM can provide additional functions (e.g., dynamic resizing) depending on the platform.

4 Keystone Security Monitor

The core of a Keystone TEE is the Security Monitor (SM). As the SM uses only standard RISC-V features, it is easily portable to the other RISC-V platforms. In addition, Keystone provides an easy way of configuring and compiling the SM depending on the underlying platform. With this design, we show how Keystone integrates with optional hardware to provide additional security guarantees such as cache side-channel defenses without any application changes. By design, the SM enforces isolation and provides security-critical features without the burden of high-level functionality like virtual memory management. This allows for a simple, and low attack surface, highest-privilege component.

4.1 Memory Isolation

Keystone only requires the RISC-V hardware to provide simple security primitives, assigns resource management logic either to the untrusted software or the enclave, and relies on trusted software executing at the highest privilege (e.g., bootloader, SM) to safely validate their decisions.

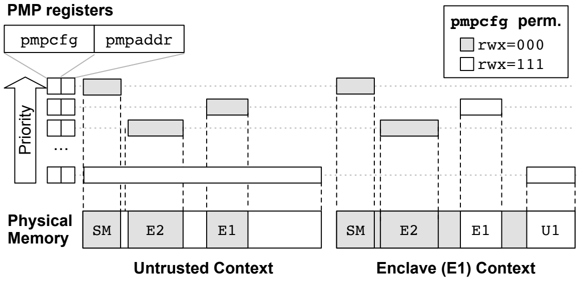


Figure 3. How Keystone uses RISC-V PMP for the flexible, dynamic memory isolation. pmpaddr and pmpcfg control and status registers (CSRs) are used to specify PMP entries. The SM uses a few PMP entries to guard its own memory (SM) and enclave memories (E1, E2). Upon enclave entry, the SM will reconfigure the PMP such that the enclave can only access its own memory (E1) and the untrusted buffer (U1).

Background: RISC-V Physical Memory Protection. Keystone uses physical memory protection (PMP), a feature provided by RISC-V. PMP restricts the physical memory access of S-mode and U-mode to certain regions defined via *PMP entries* (See Figure 3). Each PMP entry controls the U-mode and S-mode permissions to a customizable region of physical memory.[1] The PMP address registers encode the address of a contiguous physical region, configuration bits specify the r-w-x permissions for U/S-mode, and two addressing mode bits. PMP has three addressing modes to support various sizes of regions (arbitrary regions and power-of-two aligned regions). PMP entries are statically prioritized with the lower-numbered PMP entries taking priority over the higher-numbered entries. If U- or S-mode attempts to access a physical address and it does not match any PMP address range, the hardware does not grant any access permissions.

Enforcing Memory Isolation via the SM. PMP makes Keystone memory isolation enforcement flexible in three ways: (a) multiple discontiguous enclave memory regions can coexist instead of reserving one large memory region shared by all enclaves, (b) PMP entries can cover regions from 4 bytes to all of DRAM allowing for arbitrarily sized enclaves, (c) PMP entries can be reconfigured during execution to dynamically create new regions or release a region to the OS.

During the SM boot, Keystone configures the first PMP entry (highest priority) to cover its own memory region (code, stack, data such as enclave metadata and keys), disallowing access to it from U-mode and S-mode. It then configures the last PMP entry (lowest priority) to cover all memory and with all permissions enabled to allow the OS default access to memory not otherwise covered by a PMP entry.

When a host application requests the OS to create an enclave, the OS finds an appropriate contiguous physical region[2] and then calls into the SM. After validating the request, the SM protects the enclave memory by adding a PMP entry with all permissions disabled. Since the enclave's PMP entry has a higher priority than the OS PMP entry (the last in Figure 3), the OS and other user processes cannot access the enclave region. A valid request requires that enclave regions not overlap with each other or with the SM region.

During control-transfer to an enclave, the SM (for the current core only): (a) enables PMP permission bits of the relevant enclave memory region; and (b) removes all OS PMP entry permissions to protect all other memory from the enclave. This allows the enclave to access its own memory and no other regions. At a CPU context-switch to non-enclave, the SM disables all permissions for the enclave region and re-enables the OS PMP entry to allow default access from the OS. Enclave PMP entries are freed on enclave destruction.

PMP Enforcement Across Cores. Each core has its own complete set of PMP entries. During enclave creation, PMP changes must be propagated to all the cores via inter-processor interrupts (IPIs). The SM executing on each of the cores handles these IPIs by removing the access of other cores to the enclave. During the enclave execution, changes to the PMP entries (e.g., context switches between the enclave and the host) are local to the core executing it and need not be propagated to the other cores. PMP synchronization IPIs are only sent during enclave creation and destruction.

PMP Management. Each allocated enclave (executing or not) requires one PMP entry per isolated memory region it uses. We re-use the OS PMP entry during enclave execution for allowing access to shared memory. Additional PMP regions are available to enclaves via SM interfaces and are used for cases like self-paging as described in Section 5.1.

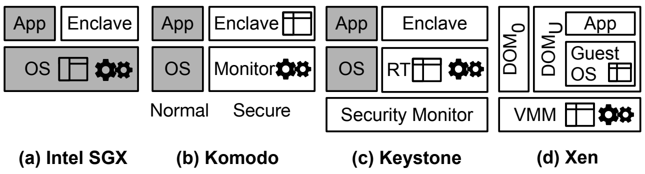
Naively, Keystone supports $N - 2$ simultaneously created enclaves, where N is the number of PMP entries available. Alternatively, with adjacent allocations by the OS, Keystone can virtualize the PMP entries at the cost of disallowing memory reclamation until all latter enclaves are destroyed. Future SM and RT features that support relocation may allow for complete virtualization of PMP entries via defragmentation. Similarly, the proposed RISC-V hypervisor mode (H-mode) would allow for an additional layer of address translation to transparently virtualize PMP entries [7].

4.2 Post-creation In-enclave Page Management

Keystone has a different memory management design from most TEEs (see Figure 4). It uses the OS-generated page tables for initialization and then delegates virtual-to-physical memory mapping entirely to the enclave during execution. Since RISC-V provides per-hardware-thread views of the

[1] Currently processors have up to 16 M-mode configurable PMP entries.

[2] Our kernel driver uses both the Buddy Allocator and the Contiguous Memory Allocator (CMA) to dynamically allocate enclave memory with various sizes.



(a) Intel SGX (b) Komodo (c) Keystone (d) Xen

Figure 4. Memory Management Designs (shaded area is untrusted). (a) Untrusted OS manages memory, translates virtual-to-physical address. (b) Page tables inside the enclave but monitor creates mappings. (c) Delegates page management to enclave with its own page table. (d) Hypervisor for page management, 2 page tables.

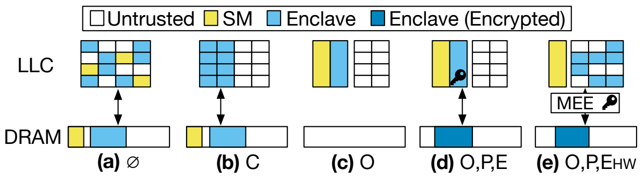
physical memory via the machine-mode and the PMP registers, it allows Keystone to have multiple concurrent and potentially multi-threaded enclaves to access the disjoint physical memory partitions. With an isolated S-mode inside the enclave, Keystone can execute its own virtual memory management which manipulates the enclave-specific page tables. Page tables are always inside the isolated enclave memory space. By leaving the memory management to the enclave we: (a) allow flexible virtual memory management with several RT modules (see Section 5.1); (b) remove controlled side-channel attacks as the host OS cannot modify or observe the enclave virtual-to-physical mapping.

4.3 Interrupts and Exceptions

During enclave execution, all machine interrupts trap directly to the SM. Exceptions (e.g. page faults, etc) may be safely delegated to the RT via the RISC-V exception delegation register. The RT then handles exceptions as needed to implement standard kernel abstractions and may forward other traps to the untrusted OS via the SM. To avoid the enclave holding a core to DoS the host the SM sets a machine timer before it enters the enclave. When the SM regains control after the timer interrupt triggers, it may return control to the host OS or request that the enclave cleanly exit.

4.4 Enclave Lifecycle

Keystone enclaves go through three distinct changes during their lifecycle. At **creation**, Keystone measures the enclave memory to ensure that the OS has loaded the enclave binaries correctly to the physical memory. Keystone uses the initial virtual memory layout for the measurement because the physical layout can legitimately vary (within limits) across different executions. For this, the SM expects the OS to initialize the enclave page tables and allocate physical memory for the enclave. The SM walks the OS-provided page table and checks if there are invalid mappings and ensures a unique virtual-to-physical mapping. The SM then hashes page contents along with the virtual addresses and the configuration data. At **execution**, the SM sets PMP entries and transfers control to the enclave entry point. On an OS initiated **destruction**, the SM clears the enclave memory region before returning the memory to the OS. SM cleans and frees all the enclave resources, PMP entries, and enclave metadata.



(a) ∅ (b) C (c) O (d) O,P,E (e) O,P,E_{HW}

Figure 5. Memory Model for Various TEE Scenarios. ∅: baseline, **C**: cache partitioning, **O**: on-chip scratchpad, **P**: enclave self-paging, **E**: software memory encryption $\mathbf{E}_{HW}$: hardware memory encryption.

4.5 TEE Primitives

Keystone supports the following standard primitives.

Secure Boot. A Keystone root-of-trust can be either a tamper-proof software (e.g., a zeroth-order bootloader) or hardware (e.g., crypto engine). At each CPU reset, the root-of-trust (a) measures the SM image, (b) generates a fresh attestation key from a secure source of randomness, (c) stores it to the SM private memory, and (d) signs the measurement and the public key with a hardware-visible secret. These standard operations can be implemented in many ways [53, 58]. Keystone does not rely on a specific implementation. For completeness, currently, Keystone simulates secure boot via a modified first-stage bootloader for all the above steps.

Secure Source of Randomness. Keystone provides a secure SM SBI call, `random`, which returns a 64-bit random value. Keystone uses a hardware source of randomness if available or can use other well-known options [66] if applicable.

Remote Attestation. The Keystone SM performs the measurement and the attestation based on the provisioned key. Enclaves may request a signed attestation from the SM during runtime. Keystone uses a standard scheme to bind the attestation with a secure channel construction [41, 58] by including limited arbitrary data (e.g., Diffie-Hellman key parameters) in the signed attestation report. Key distribution [15], revocation [46], attestation services [49], and anonymous attestation [26] are orthogonal challenges.

Other Primitives. Keystone can support other primitives, if required by the TEE: (a) it allows enclaves to access the read-only hardware-maintained timer registers via the standard `rdcycle` instruction; (b) it can provide monotonic counters by keeping a limited counter state in the SM memory. The SM can implement *trusted timers*, *rollback defense* [35], and *sealed storage* [15] with these features.

4.6 Platform-Specific Extensions

Keystone can leverage additional security and functionality features exposed by the hardware to provide stronger security guarantees and/or additional features to the enclave at the cost of various trade-offs. We demonstrate several examples of customizing the SM for a specific platform so that it can defend the enclave against a physical attacker or cache side-channel attacks. We use the HiFive Freedom Unleashed [9] RISC-V dev board containing a Rocket-based quad-core SoC chip (FU540) with a proprietary L2 controller.

Secure On-chip Memory. To protect the enclaves against a physical attacker who has access to the DRAM, we implemented an *on-chip memory* extension (Figure 5(c)). It allows the enclave to execute without the code or the data leaving the chip package. On the FU540, we dynamically instantiate a scratchpad memory of up to 2MB via the L2 memory controller to generate a usable on-chip memory region. The scratchpad is then allocated exclusively to the requesting enclave for it's entire lifetime. An enclave requesting to run in the on-chip memory loads nearly identically to the standard procedure with the following changes: (a) the host loads the enclave to the OS allocated memory region with modified initial page tables referencing the final scratchpad address; and (b) the SM copies the standard enclave memory region into the new scratchpad region before the measurement. Any context switch to the enclave now results in an execution in the scratchpad memory. This uses only our basic enclave life-cycle hooks for the platform-specific features and does not require further modification of the SM. The only other change required was a modification of the untrusted enclave loading process to make it aware of the physical address region that the scratchpad occupies. No modifications to the Eyrie RT or the eapps are required.

Cache Partitioning. Enclaves are vulnerable to cache side-channel attacks from the untrusted OS and other applications via a shared cache. To this end, we implement a cache partitioning scheme using two hardware capabilities: (a) the L2 cache controller's *waymasking* primitive similar to Intel's CAT [70]; (b) PMP to way-partition the L2 cache transparently to the OS and the enclaves. The resulting SM enforces effective *non-interference* between the partitioned domains (Figure 5(b)). Upon a context switch to the enclave, the cache lines in the partition are flushed. During the enclave execution, only the cache lines from the enclave physical memory are in the partition and are thus protected by PMP. The adversary cannot insert cache lines in this partition during the enclave execution due to the line replacement way-masking mechanism. As a net effect, adversary (A_{Cache}) gains no information about the evictions, the resident lines, or the residency size of the enclave's cache. Ways are partitioned at runtime and are available to the host whenever the enclave is not executing even if paused.

Dynamic Resizing. Statically pre-defined maximum enclave size and subsequent static physical or virtual memory pre-allocations: (a) prevent the enclave from scaling dynamically based on workload, (b) complicates porting applications to eapps. To this end, Keystone allows the SM to dynamically change the physical memory boundaries of the enclave. The Eyrie RT may request that the OS make an `extend` SBI call to add contiguous physical pages to the enclave memory region. If the OS succeeds in allocating, the SM increases the enclave's size by extending the relevant PMP entries and notifies the RT, which then uses the free memory module to manage the new physical pages (see Section 5.1).

5 Keystone Modular Runtime

As the SM physically isolates each of the enclaves, we can safely allow the enclave to run private S-mode code (i.e., the RT). This enables modular system-level abstraction for eapps (e.g., virtual memory management). Although the RT is similar in functionality to a kernel inside an enclave, it does not require most kernel functionality. We built a modular exemplar RT—**Eyrie**—to allow enclave developers the ability to include only necessary functionality and reduce the TCB.

Given the supervisor capability, we can cleanly implement selected kernel functionality without modifying user applications. The additional privilege layer allows for further defensive design, such as only allowing the RT access to the shared memory buffer. Moreover, it enables easy porting of a full-fledged off-the-shelf microkernel such as seL4 in an enclave. We introduce key Keystone RT modules and show how they support various workloads with small TCB.

5.1 Enclave Memory Management Modules

Each enclave can run both S-mode and U-mode code. Since they have the privileges to manage their own memory, they need not cross the host-enclave isolation boundary. By default, Keystone enclaves occupy a fixed contiguous physical memory allocated by the OS with a statically-mapped virtual address space at load time. While suitable for some embedded applications, it limits the memory usage of most legacy applications. To this end, we describe several optional modules to enable flexible enclave memory management.

Free memory. We built a module that allows the Eyrie RT to perform page table management, after the enclave reserves unmapped physical memory. Thus, the page mappings need not be pre-defined at creation time. The unmapped (hence, free) memory region is not included in the enclave measurement and is zeroed before beginning the eapp execution. The free-memory module is required for other more complex memory modules.

In-Enclave Self Paging. We implemented a generic in-enclave page swapping module for the Eyrie RT. It handles the enclave page-faults and uses a generic page backing-store that manages the evicted page storage and retrieval. Our module uses a simple random eapp-only page eviction policy. It works in conjunction with the free memory module for virtual memory management in the Eyrie RT. Put together, they help to alleviate the tight memory restrictions an enclave may have due to the limited DRAM or the on-chip memory size [71–73].

Protecting the Page Content Leaving the Enclave. When the enclave handles its own page fault, it may attempt to evict the pages out of the secure physical memory (either an on-chip memory or the protected portion of the DRAM). When these pages have to be copied out, their content needs to be protected. Thus, as part of the in-enclave page management, we implement a backing-store layer that can include page encryption and integrity protection to allow for the secure content to be paged out to the insecure storage (DRAM regions or disk). The protection can be done either in the software as a part of the Keystone RT (Figure 5(d)) or by a dedicated trusted hardware unit—a memory encryption engine (MEE) [44]—with the SM's on-chip memory capability (Figure 5). Admittedly, this incurs significant design challenges in efficiently storing the metadata and performance optimizations. The amount of available on-chip memory for integrity data storage will cap the total possible size of the enclave. Keystone design is agnostic to the specific integrity schemes and can reuse the existing mechanisms [65, 79].

5.2 Functionality Modules

Next, we demonstrate various functionality modules in Eyrie.

Edge Call Interface. The eapp cannot access the non-enclave memory in Keystone. If it needs to read/write the data outside the enclave, the Eyrie RT performs *edge calls* on its behalf. Our edge call, which is functionally similar to RPC, consists of an index to a function implemented in the untrusted host application and the parameters to be passed to the function. Eyrie tunnels such a call safely to the untrusted host, copies the return values of the function back to the enclave, and sends them to the eapp. The copying mechanism requires Eyrie to have access to a buffer shared with the host. To enable this: (a) the OS allocates a shared buffer in the host memory space and passes the address to the SM at enclave creation; (b) the SM passes the address to the enclave so the RT may access this memory; (c) the SM uses a separate PMP entry to enable OS access to this shared buffer. All the edge calls have to pass through the Eyrie RT as the eapp does not have access to the shared memory virtual mappings. This module can be used to add support for syscalls, IPC, enclave-enclave communication, and so on. As the current edge interface is a straight-forward shared memory region, it can easily to use alternative methods for dispatching calls such as mailboxes or HotCalls [89].

We allow the *proxying of syscalls* from the eapp to the host application by re-using the edge call interface. The user host application then invokes the syscall on an untrusted OS on behalf of the eapp, collects the return values, and forwards them to the eapp. Keystone can utilize existing defenses to prevent Iago attacks [32] via this interface [31, 73, 82]. Keystone resolves appropriate calls as *in-enclave syscalls* (e.g., mmap, brk, getrandom). Such calls are handled in Eyrie and invoke SM interfaces as needed (e.g. getrandom) before returning to the eapp.

Multi-threading. We run multi-threaded eapps by delegating the thread management to the runtime. We do not support parallel multi-core enclave execution yet, but this can be implemented by allowing the SM to invoke enclave execution multiple times in different cores.

6 Security Analysis

We argue the security of the enclave, the OS, and the SM based on the threat model outlined in Section 3.4.

6.1 Protection of the Enclave

Keystone attestation ensures that any modification of the SM, RT, and the eapp is visible while creating the enclave. During the enclave execution, any direct attempt by an A_{SW} to access the enclave memory (cached or uncached) is defeated by PMP. All enclave data structures can only be modified by the enclave or the SM, both are isolated from direct access. Subtle attacks such as controlled side channels (A_{Cntrl}) are not possible in Keystone as enclaves have dedicated page management and in-enclave page tables. This ensures that any enclave executing with any Keystone instantiated TEE is always protected against the above attacks.

Mapping Attacks. The RT is trusted by the eapp, does not intentionally create malicious virtual to physical address mappings [45] and ensures that the mappings are valid. The RT initializes the page tables either during the enclave creation or loads the pre-allocated (and SM validated) static mappings. During the enclave execution, the RT ensures that the layout is not corrupted while updating the mappings (e.g., via mmap). When the enclave gets new empty pages, say via the dynamic memory resizing, the RT checks if they are safe to use before mapping them to the enclave. Similarly, if the enclave is removing any pages, the RT scrubs their content before returning them to the OS.

Syscall Tampering Attacks. If the eapp and the RT invoke untrusted functions implemented in the host process and/or execute the OS syscalls, they are susceptible to Iago attacks and system call tampering attacks [32, 77]. Keystone can reuse the existing shielding systems [18, 31, 82] as RT modules to defend the enclave against these attacks.

Side-channel Attacks. Keystone thwarts cache side-channel attacks (Section 4.6). Enclaves do not share any state with the host OS or the user application and hence are not exposed to controlled channel attacks. The SM performs a clean context switch and flushes the enclave state (e.g., TLB). The enclave can defend itself against explicit or implicit information leakage via the SM or the edge call API with known defenses [83, 84]. Only the SM can see other enclave events (e.g., interrupts, faults), these are not visible to the host OS. Timing attacks against the eapp are out of scope.

6.2 Protecting the Host OS

Keystone RTs execute at the same privilege level as the host OS, so an A_{SW} in our case is stronger than in SGX. We ensure that the host OS is not susceptible to new attacks from the enclave because the enclave cannot: (a) reference any memory outside its allocated region due to the SM PMP-enforced isolation; (b) modify page tables belonging to the host user-level application or the host OS; (c) pollute the host state because the SM performs a complete context switch (TLB, registers, L1-cache, etc.) when switching between an enclave and the OS; (d) DoS a core as the enclave will be interrupted by the machine timer set by the SM such that the SM can return the control to the OS.

6.3 Protection of the SM

The SM naturally distrusts all the lower-privilege software components (eapps, RTs, host OS, etc.). It is protected from an A_{SW} because all the SM memory is isolated using PMP and is inaccessible to any enclave or the host OS. The SM SBI is another potential avenue of attack. Keystone's SM presents a narrow, well-defined SBI to the S-mode code. It does not do complex resource management and is small enough to be formally verified [41, 68]. The SM is only a reference monitor, it does not require scheduled execution time, so an A_{DoS} is not a concern. The SM can defend against an A_{Cache} and an A_{Time} with known techniques [42, 54].

6.4 Protection Against Physical Attackers

Keystone can protect against a physical adversary via platform features and a proposed modification to the bootloader. Similar to Chen et al. [33], the SM uses a scratchpad to store the decrypted code and data, while the supervisor mode component (Eyrie modules) handles paging enclave content to DRAM when the scratchpad becomes full.

The enclave itself is protected by the combination of the on-chip memory and the RT's paging module, with encryption and integrity protection on the pages leaving the on-chip memory. The page backing-store is a standard PMP protected physical memory region now containing only the encrypted pages, similar in concept to the SGX EPC. This fully guarantees the confidentiality and integrity of the enclave code and data from an attacker with control of DRAM.

The SM should be executed entirely from the on-chip memory. The SM is statically sized and has a relatively small in-memory footprint (< 150Kb). On the FU540, this would involve repurposing a portion of the L2 loosely-integrated memory (LIM) via a modified trusted bootloader.

With these techniques in place, content outside of the chip package is either untrusted (host, OS, etc.) or is encrypted and integrity protected (e.g., swapped enclave pages). Keystone accomplishes this with no application modifications.

Platform	**Core**		**Cache Size (KB)**		**Latency (cycles)**		**# of TLB Entries**	
	#	**Type**	**L1-I/D**	**L2**	**L1**	**L2**	**L1**	**L2**
Rocket-S	1	in-order	8/8	512	2	24	8	128
Rocket	1	in-order	16/16	512	2	24	32	1024
BOOM	1	OoO	32/32	2048	4	24	32	1024
FU540	4	in-order	32/32	2048	2	12-15*	32	128

Table 2. Hardware specification for each platform. L2 cache latency in FU540 (*) is based on estimation.

7 Evaluation

We aim to answer the following questions in our evaluation:

(***RQ1***) **Modularity.** Is the Keystone framework viable in different configurations for real applications?

(***RQ2***) **TCB.** What is the TCB of a Keystone-instantiated TEE in various deployment modes?

(***RQ3***) **Performance.** How much overhead do simple Keystone TEEs add to eapp execution time?

(***RQ4***) **Real-world Applications.** Does Keystone provide expressiveness with minimal developer efforts for eapps?

7.1 Implementation & Experimental Setup

We implemented our **SM** on top of the Berkeley Boot Loader (bbl) [13]. It supports M-mode trap handling, boot, and other features. We implemented the initialization of the SM at boot as well as the SBI specified in Table 1. Platform-specific extensions have been implemented with hooks in SBI functions. We simulated unavailable hardware primitives such as the random number generator and the root of trust. All modules in Sections 4 and 5 are available as compile-time options.

We implemented the **Eyrie RT** from scratch in C. Memory encryption is done via software AES-128 [2] and integrity protection is partially implemented. We ported the **seL4 microkernel** [55] to Keystone by modifying 290 LoC for boot, memory initialization, and interrupt handling. There is no inherent restriction to these two RTs, and we expect to add further options.

Our host user-land interface for interactions with the enclaves is provided via a **Linux kernel driver** that creates a device endpoint (`/dev/Keystone`). The untrusted host OS (i.e., Linux) launches and manages the enclaves via SBI on behalf of the user, and also manages the enclave ownership and enclave-related OS resources.

We provide several libraries (edge-calls, host-side syscall endpoints, attestation, etc.) in C and C++ for the host, the eapp, and interaction with the driver-provided Linux device. Our provided tools generate the enclave measurements (hashes) without requiring RISC-V hardware, customize the Eyrie RT, and package the host application, eapps, and RT into a single binary. We have a complete top-level build solution to generate a bootable Linux image (based on the

SM Component	LoC	Runtime Component	LoC
Base	1100	–	1800
Edge-call Handling	30	–	300
Dynamic Memory	70	–	100
Memory Isolation	500	libc Environment	50
Cache Partitioning	300	In-enclave Paging	300
Secure Boot	170	Syscalls	450
On-chip Memory	50	Free Memory	300
		IO Syscall Proxying	300

Table 3. TCB Breakdowns for the Eyrie RT and SM features in LoC.

tooling for the HiFive Freedom Unleashed) for QEMU, FPGA softcores, and the HiFive containing our SM, the driver, and the enclave binaries.

We used four different platforms for our experiments; the HiFive Freedom Unleashed [9] with a closed-source FU540 (at 1GHz), and three open-source RISC-V processors: small Rocket (Rocket-S), default Rocket [19], and Berkeley Out-of-order Machine (BOOM) [29] (See Table 2). We instantiate the open-source processors on cloud FPGAs using FireSim [52] which simulates the cores at 1GHz. The host OS is buildroot Linux (kernel 4.15). All performance evaluation was performed on the HiFive and the data is averaged over 10 runs unless otherwise specified.

7.2 Modularity & Support

We outline the qualitative measurement of Keystone flexibility in extending features, reducing TCB, and using the platform features. Table 3 shows the TCB breakdown of various components (required and optional) for the SM and Eyrie RT. Most of the modifications (e.g., additional edge-call features) require no changes to the SM, and the eapp programmer may enable them as needed. Future additions (e.g., ports of interface shields) may be implemented exclusively in the RT. We also add support for a new RT by porting seL4 to Keystone and use it to execute various eapps (See Section 7.4). Keystone passes all the tests in seL4 suite and incurs less than 1% overhead on average over all test cases. The advantage of an easily modifiable SM layer is noticeable when features require interaction with the core TEE primitives like memory isolation. The SM features were able to take advantage of the L2 cache controller on the FU540 to offer additional security protections (cache-partitioning and on-chip isolation) without changes to the RT or eapp.

TCB Breakdown. Keystone comprises of the M-mode components (bbl and SM), the RT, the untrusted host application, the eapp, and the helper libraries, of which only a fraction is in the TCB. The M-mode component is 10.7 KLoC: a cryptographic library (4 KLoC), pre-existing trap handling, boot, and utilities (4.7 KLoC), the baseline SM (1.6 KLoC), and platform-specific code for FU540 (400 LoC). A minimum Eyrie RT is 1.8 KLoC, with modules adding further code as shown in Table 3 up to a maximum Eyrie RT TCB of 3.6 KLoC. The current maximum TCB for an eapp running on our SM and Eyrie RT is thus a total of 15 KLoC. TCB calculations were made using cloc [8] and unifdef [14].

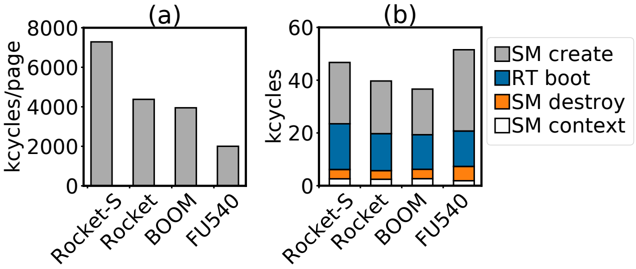


Figure 6. Breakdown of operations during the enclave life-cycle. (a) shows enclave validation and hashing duration, and (b) shows the breakdown of other operations. (b) does not include duration of size-dependent operations such as measurement in create (Shown in (a)) and memory cleaning in destroy (4K-11K cycles/page).

7.3 Benchmarks

We use 4 standard benchmark suites with a mix of CPU, memory, and file I/O for system-wide analysis: **Beebs**, **CoreMark**, **RV8**, and **IOZone**. We report the overheads of the cache partitioning and physical attacker protection with RV8 as an example of Keystone trade-offs. In all the graphs, 'other' refers to the lifecycle costs for enclave creation, destruction, etc. All benchmarks are run as unmodified RISC-V binaries using an Eyrie runtime with relevant modules as needed.

Common Operations. Figure 6 shows the breakdown of various enclave operations. Initial validation and measurement dominate the startup with 2M and 7M cycles/page for FU540 and Rocket-S due to an unoptimized software implementation of SHA-3 [4]. The remaining enclave creation time totals 20k-30k cycles. Similarly, the attestation is dominated by the ed25519 [6] signing software implementation (not shown in the graph, 0.7M-1.6M cycles). These are both one-time costs per-enclave and can be substantially optimized in software or hardware. The most common SM operation, context switches, currently take between 1.8K(FU540)-2.6K(Rocket-S) cycles depending on the platform. Notably, creation and destruction of enclaves takes long on the FU540 (4-core) due to the multi-core PMP synchronization.

Standard Benchmarks as Unmodified eapp Binaries.
Beebs, CoreMark, and RV8. As expected, Keystone incurs no meaningful overheads (±0.7%, excluding enclave creation and destruction) for pure CPU and memory benchmarks.
IOZone. All the target files are located on the untrusted host and we tunnel the I/O syscalls to the host application. Figure 7 shows the throughput plots of common file-content access patterns. Keystone experiences expected high throughput loss for both write (avg. 36.2%) and read (avg. 40.9.%). Three factors contribute to the overhead: (a) all the data crossing the privilege boundary is copied an additional time

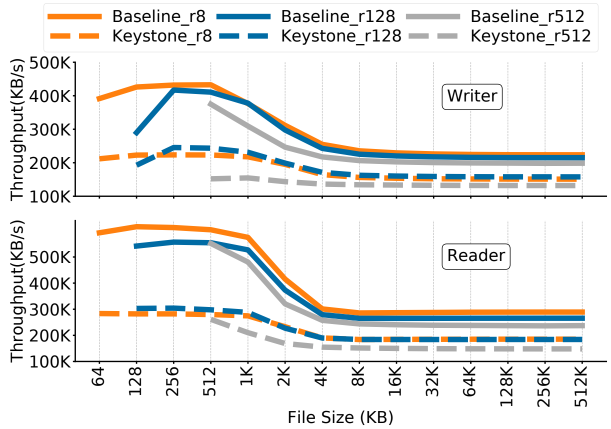


Figure 7. IOZone throughput in Keystone for various file and record sizes (e.g., r8 represents 8KB record). We only show write and read results due to limited space.

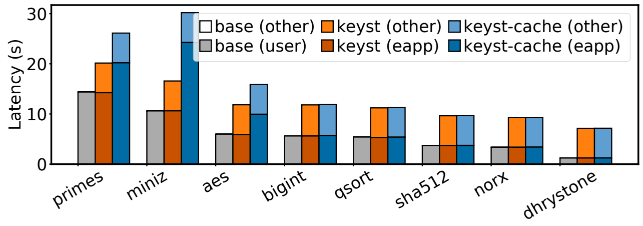


Figure 8. Full-execution time comparison for RV8. Each bar shows the duration of the application (user or eapp), and the other overheads (other). Keystone (keyst) and Keystone with cache partitioning (keyst-cache) compared to native execution (base).

via the untrusted buffer, (b) each call requires the RT to go through the edge call interface, incurring a constant overhead, and (c) the untrusted buffer contends in the cache with the file buffers, incurring an additional throughput loss on re-write (avg. 38.0%), re-read (avg. 41.3%), and record re-write (avg. 55.1%) operations. Since (b) is a fixed cost per system call, it increases the overhead for the smaller record sizes.

Cache Partitioning. The mix of pure-CPU and large working-set benchmarks in **RV8** are ideal to evaluate the impact of caceh partitioning. We granted 8 of the 16 ways in the L2 cache to the enclave during execution (see Figure 8). Small working-set tests show low overheads from cache flush on context switches whereas large working-set tests (primes, miniz, aes) show up to 128% overhead due to a smaller effective cache. Enclave initialization latency is unaffected.

Physical Attacker Protections. We ran the **RV8** suite with on-chip execution, enclave self-paging, page encryption, and a DRAM backing page store (Table 4). A few eapps (sha512, dhrystone), which fit in the 1MB on-chip memory, incur no overhead and are protected even from A_{Phy}. For the larger working-set-size eapps, the paging overhead increases depending on the memory access pattern. For example, primes

	Overhead (%)				# of Page
Benchmark	∅	**C**	**O, P**	**O, P, E**	**Faults**
primes	-0.9	40.5	65475.5	*	66×10^6
miniz	0.1	128.5	80.2	615.5	18341
aes	-1.1	66.3	1471.0	4552.7	59716
bigint	-0.1	1.6	0.4	12.0	168
qsort	-2.8	-1.3	12446.3	26832.3	285147
sha512	-0.1	0.3	-0.1	-0.2	0
norx	0.1	0.9	2590.1	7966.4	58834
dhrystone	-0.2	0.3	-0.2	0.2	0

Table 4. RV8 Overhead for different TEE design instances. ∅: baseline, **C**: cache partitioning, **O**: on-chip scratch pad execution (1MB), **P**: enclave self-paging, **E**: software-based memory encryption. *: does not complete in ~10 hrs.

incurs the largest amount of page faults because it allocates and randomly accesses a 4MB buffer causing a page fault for almost every memory access. Software-based memory encryption adds 2 − 4× more overhead to page faults. These overheads can be alleviated by the Keystone framework if a larger on-chip memory or dedicated hardware memory encryption engine is available as we discussed in Section 5.

7.4 Case Studies

We demonstrate how Keystone can be adapted for a varied set of devices, workloads, and application complexities with three case-studies: (a) machine learning workloads for the client and server-side usage, (b) machine learning for varied RTs, (c) a small secure computation application written natively for Keystone. The evaluation for these case-studies was performed on the HiFive board. We used the unmodified application code logic, hard-coded all the configurations and arguments for simplicity, and statically linked the binaries against glibc or musl libc supported by the Eyrie RT. We ported the widely used cryptographic library libsodium to both Eyrie and seL4 RT trivially.

Case-study 1: Secure ML Inference with Torch and Eyrie. We ran nine Torch-based models of increasing sizes with Eyrie on the Imagenet dataset [39] (see Table 5). They comprise 15.7 and 15.4KLoC of TH [3] and THNN [5] libraries from Torch compiled with musl libc. Each model has an additional 230 to 13.4 KLoC of model-specific inference code [88]. We performed two sets of experiments: (a) execute the model inference code with static maximum enclave size; (b) with dynamic resizing support to allow the enclave size to increase on-demand. Figure 9 shows the performance overheads for both configurations and non-enclaved execution baseline.

Initialization Overhead is noticeably high for both static size and dynamic resizing. It is proportional to the eapp binary size due to enclave page hashing. Dynamic resizing reduces the initialization latency by 2.9% on average as the RT does not map free memory during enclave creation.

Eapp Execution Overhead was between −3.12%(LeNet) and 7.35%(Densenet) for all the models with both static size and dynamic resizing. The causes of this are: (a) Keystone loads

Model	# of Layers	# of Param	App LOC	Binary Size	Memory Usage
Wideresnet	93	36.5M	1625	140MB	384MB
Resnext29	102	34.5M	1910	123MB	394MB
Inceptionv3	313	27.2M	5359	92MB	475MB
Resnet50	176	25.6M	3094	98MB	424MB
Densenet	910	8.1M	13399	32MB	570MB
VGG19	55	20.0M	1088	77MB	165MB
Resnet110	552	1.7M	9528	7MB	87MB
Squeezenet	65	1.2M	914	5MB	52MB
LeNet	12	62K	230	0.4MB	2MB

Table 5. Torch model specification, workload characteristics, binary object size, and total enclave memory usage.

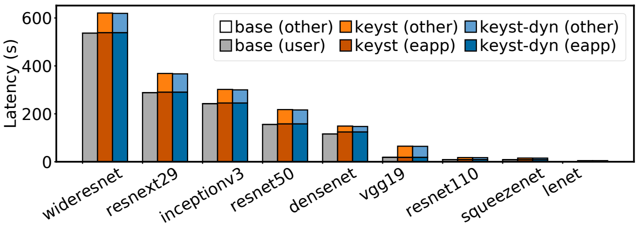


Figure 9. Inferencing time for various Torch models. Each bar consists of the duration of the application (`user` or `eapp`), and the other overheads (`other`). Keystone (`keyst`) and Keystone with the dynamic resizing (`keyst-dyn`) compared to native execution in (`base`).

the entire binary in physical memory before it beings eapp execution, precluding any page faults for zero-fill-on-demand or similar behavior, so smaller sized networks like LeNet execute faster in Keystone and (b) the overhead is primarily proportional to the number of layers in the network, as more layers results in more memory allocations and increase the number of `mmap` and `brk` syscalls. We used a small hand-coded test to verify that Eyrie RT's custom `mmap` is slower than the baseline kernel and incurs overheads. Densenet, which has the maximum number of layers (910), thus suffers from larger performance degradation. In summary, for long-running eapps, Keystone incurs a fixed one-time startup cost and the dynamic resizing is indeed useful for larger eapps.

Case-study 2: Secure ML with FANN and seL4. Keystone can be used for small devices such as IoT sensors and cameras to train models locally as well as flag events with model inference. We ran FANN, a minimal (8KLoC C/C++) eapp for embedded devices with the seL4 RT to train and test a simple XOR network. The end-to-end execution overhead is 0.36% over running in seL4 without Keystone.

Case-study 3: Secure Remote Computation. We implemented a secure server eapp (and remote client) to count words in an input message using the Eyrie and baseline SM. It performs attestation, uses `libsodium` to bind a secure channel to the attestation report, then polls the host for encrypted messages using edge-calls, processes them inside the enclave, and returns an encrypted reply to be sent to the client. The eapp has secure channel code (60 LoC), the edge-wrapping interface (45 LoC), and other logic (60 LoC). The host is 270 LoC and the remote client is 280 LoC. Keystone takes 45K cycles for a round-trip with an empty message, secure channel, and message passing overheads. It takes 47K cycles between the host getting a message and the enclave notifying the host to send a reply.

8 Related Work

Here, we survey TEEs and design trade-offs that have been explored in existing works.

TEE Architectures & Extensions. Three TEEs are closely related to Keystone: (a) Intel Software Guard Extension (SGX) executes user-level code in an isolated virtual address space backed by encrypted RAM pages [64]; (b) ARM TrustZone divides the memory into two worlds (i.e., normal vs. secure) to run applications in protected memory [1]; and (c) Sanctum uses a machine-mode SM, the memory management unit (MMU), and cache partitioning to isolate enclave memory and prevent controlled-channel and cache side-channel attacks [36]. Several other TEEs explore design at layers such as hypervisors [34, 45, 61], physical memory [30, 57, 62], virtual memory [25, 37, 80], and process isolation [38, 76, 86, 87]. Interested readers can refer to Appendix A for a summary of TEE design choices.

Re-purposing Existing TEEs for Modularity. One way to meet Keystone's design goal of customizable TEEs is to reuse the TEE solutions that are available on commodity CPUs. For each TEE, it is possible to enable a subset of programming constructs (e.g., threading, dynamic loading of binaries) by including a software management component inside the enclave [12, 22, 31]. Alternatively, adding hardware extensions which are specifically designed and implemented for adding TEE capabilities requires lot of efforts [36, 71]. Another approach is to simulate the programmable layer, say with a trusted hypervisor layer, which then executes an untrusted OS, but potentially inflates the TCB.

Differences from Trusted Hypervisor Keystone executes the enclave logic in the supervisor mode (RT) and the user mode (eapp), while the machine mode code (SM) merely checks and enforces isolation boundaries. Although Keystone may seem similar to a trusted hypervisor, it does not implement or perform any resource management, virtualization, or scheduling in the SM. It merely checks if the untrusted OS and the enclave (RT, eapp) are managing the shared resources correctly. Thus, Keystone SM is more analogous to a reference monitor [16, 78].

TEE Support. Several works enhance existing TEEs. At the SM layer they optimize program-critical tasks [21, 36, 80]. At the hypervisor layer they add support for multiplexing the secure isolation enforced by hardware or use nested virtualization for isolation [23, 37, 47]. At the RT layer, they target portability, functionality, security [11, 12, 18, 22, 31, 67, 81]. At the eapp layer they reduce the developer efforts [20]. Although these systems are a fixed configuration in the TEE design space, they provide valuable lessons for Keystone features and optimization.

Enhancing the Security of TEEs. Better and secure TEE design has been a long-standing goal, with advocacy for security-by-design [48, 75]. We point out that Keystone is not vulnerable to a large class of side-channel attacks [28, 90] by design, while speculative execution attacks [27, 56] are limited to out-of-order RISC-V cores (e.g., BOOM) and do not affect most SOC implementations (e.g., Rocket). Keystone can re-use known cache side-channel defenses [24, 54] as we demonstrated in Section 4.6. Lastly, Keystone can benefit from various RISC-V proposals underway to secure IO operations with PMP [74]. Thus, Keystone either eliminates classes of attacks or allows integration with existing techniques.

Formally Verified Hardware & Software. TEE-like guarantees can be achieved orthogonally by a hardware and software stack which is formally verified as resistant against all classes of attacks that TEEs prevent. A careful and ground-up design with verified components [43, 55, 69] may provide stronger guarantees and Keystone can help explore designs which combine these with hardware protection [41, 85].

Resemblance with traditional kernel designs. Despite being designed for the TEE threat model, Keystone borrows and builds on well-known principles from a long line of work in OS design. Specifically, our choice of separating isolation (SM) and functionality (RT) has been explored mainly in micro-kernels [60]. Further, like many other works, our SM is inspired by the concept of reference monitors [16, 78]. Lastly, the modularity of abstraction between the host OS, the RT and eapp is similar to exokernels [40].

9 Conclusion

We present Keystone, the first framework for customizable TEEs. With our modular design, we showcase the use of Keystone for several standard benchmarks and applications on illustrative RTs and various deployment platforms. Keystone serves as a framework for both TEE research and future deployment of novel TEE designs.

Acknowledgement

We thank our shepherd, Yubin Xia, and the anonymous reviewers for their insightful comments. We thank Srini Devadas and Ilia Lebedev for sharing the Sanctum codebase and initial discussions. We thank Alexander Thomas, Stephan Kaminsky, and Gui Andrade for their help in building Keystone and Jerry Zhao for help in running Keystone on BOOM. Thanks to the members of UCB BAR for their help with the HiFive Freedom Unleashed and FireSim setup. We thank Paul Kocher and Howard Wu for their feedback on the early versions of this draft. Thanks to the Privado team at Microsoft Research for sharing the torch code for our case-study. This material is in part based upon work supported by the National Science Foundation under Grant No. TWC-1518899, Center for Long-Term Cybersecurity, and DARPA N66001-15-C-4066. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Research partially funded by RISE Lab sponsor Amazon Web Services, ADEPT Lab industrial sponsors and affiliates Intel, HP, Futurewei, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Availability

The Keystone implementation for all platforms (QEMU, FireSim, and HiFive Freedom Unleashed) is available at https://github.com/keystone-enclave/keystone. The modified seL4 runtime is available at https://github.com/keystone-enclave/keystone-seL4. General information and documentation is available at https://keystone-enclave.org.

A Trade-offs in existing TEEs

Table 6 shows the trade-offs in the existing TEE or TEE-based systems.

	System	C3. Software Adversary	C4. Physical Adversary	C5. Side-Channel Adversary	C6. Ctrl-Channel Adversary	C7. Low Software TCB	C8. No Hardware Modification	C9. Resource Management	C10. All Applications	C11. High Expressiveness	C12. Low Porting Efforts
Intel	SGX [64]	●	●	○	○	●	○	○	○	○	○
	Haven [22]	●	●	○	○	○	●	◑	◑	●	●
	Graphene-SGX [31]	●	●	○	○	○	●	◑	◑	●	●
	Scone [18]	●	●	○	○	◑	●	○	◑	●	◑
ARM	TrustZone [1]	●	○	○	○	○	○	●	●	●	○
	Komodo [41]	●	●	○	●	◑	●	◑	●	◑	●
	OP-TEE [12]	●	●	○	○	○	●	●	●	●	●
	Sanctuary [25]	●	○	◑	○	○	●	◑	●	●	●
AMD	SEV [51]	◑	◑	○	○	●	○	●	●	●	●
	SEV-ES [50]	●	●	○	○	●	○	●	●	●	●
RISC-V	Sanctum [36]	●	○	●	●	◑	○	◑	◑	◑	●
	TIMBER-V [80]	●	●	○	○	●	○	●	◑	●	◑
	MultiZone [10]	●	●	●	●	◑	●	○	○	○	◑
	Keystone (This paper)	●	●	●	●	◑	●	●	●	◑	●

Table 6. Trade-offs in existing TEEs/extensions. ●, ◑, ○: best to worst respectively. **C3-6**: resilience to software adversary, hardware adversary, side-channel adversary, controlled-channel adversary respectively. indicates complete protection; confidentiality only; no protection. **C7**: zero; thousands LoC; millions LoC. **C8**: zero; non-zero hardware; micro-architectural modifications. **C9**: enclave self resource management; partial; no flexibility. **C10**: range of apps supported are maximum; specific class; only written from scratch. **C11**: expressiveness includes forking, multi-threading, syscalls, shared memory; partial; none of these. **C12**: dev-effort for porting is unmodified binaries; compiling and/or configuration files; re-writing.

References

[1] 2013. ARM TrustZone. infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[2] 2015. AES. https://github.com/B-Con/crypto-algorithms.

[3] 2015. Torch Tensors. https://github.com/torch/TH.

[4] 2016. Tiny SHA3. https://github.com/mjosaarinen/tiny_sha3/.

[5] 2017. Torch NNs. https://github.com/torch/nn/tree/master/lib/THNN.

[6] 2019. Ed25519. https://github.com/mit-sanctum/ed25519.

[7] 2019. Hypervisor draft v0.5. https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20191030-899457c.

[8] 2020. cloc - count lines of code. https://github.com/AlDanial/cloc.

[9] 2020. HiFive Unleashed. https://www.sifive.com/boards/hifive-unleashed.

[10] 2020. MultiZone Hex Five Security. https://hex-five.com/.

[11] 2020. Open Enclave SDK. https://openenclave.io/sdk/.

[12] 2020. Open Portable TEE. https://www.op-tee.org/.

[13] 2020. RISC-V Proxy Kernel. https://github.com/riscv/riscv-pk.

[14] 2020. unifdef. http://dotat.at/prog/unifdef/.

[15] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *HASP*.

[16] James P Anderson. 1972. *Computer Security Technology Planning Study*. Technical Report. Anderson (James P) and Co Fort Washington PA.

[17] Krste Asanović Andrew Waterman. 2017. The RISC-V Instruction Set Manual Volume II: Privileged Architecture. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*.

[19] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17.

[20] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution. In *EuroSys*.

[21] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *CCS*.

[22] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*.

[23] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI*.

[24] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *MICRO*.

[25] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. Sanctuary: ARMing TrustZone with User-space Enclaves. In *NDSS*.

[26] Ernie Brickell, Jan Camenisch, and Liqun Chen. 2004. Direct Anonymous Attestation. In *CCS*.

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

[28] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*.

[29] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Technical Report UCB/EECS-2015-167.

[30] D. Champagne and R. B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA*.

[31] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *ATC*.

[32] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS*.

[33] Xi Chen, Robert P Dick, and Alok Choudhary. 2008. Operating system controlled processor-memory bus encryption. In *DATE*.

[34] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS*.

[35] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086.

[36] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*.

[37] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS*.

[38] Mark Horowitz David Lie, Chandramohan A. Thekkath. 2003. Implementing an Untrusted Operating System on Trusted Hardware. In *SOSP*.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[40] Dawson R. Engler. 1998. *The Exokernel Operating System Architecture*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0800457.

[41] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*.

[42] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2018).

[43] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*.

[44] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204.

[45] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS*.

[46] R. Housley, W. Polk, W. Ford, and D. Solo. 2002. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile.

[47] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *USENIX Security*.

[48] Galen Hunt, George Letey, and Ed Nightingale. 2017. *The Seven Properties of Highly Secure Devices*. Technical Report. https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/

[49] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel Software Guard Extensions: EPID Provisioning and Attestation Services.

[50] David Kaplan. 2017. AMD SEV-ES. http://support.amd.com/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf.

[51] David Kaplan, Jeremy Powell, and Tom Woller. 2016. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[52] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *ISCA*.

[53] Pierre Selwan Ken Irving. 2018. Revolutionizing the Computing Landscape and Beyond. https://content.riscv.org/wp-content/uploads/2018/12/RISC-V-MultiCore-Secure-Boot-Ken-Irvining-and-Pierre-Selwan.pdf.

[54] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*.

[55] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*.

[56] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*.

[57] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A Security Architecture for Tiny Embedded Devices. In *EuroSys*.

[58] Ilia Lebedev, Kyle Hogan, and Srinivas Devadas. 2018. Secure Boot and Remote Attestation in the Sanctum Processor. In *CSF*.

[59] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security*.

[60] J. Liedtke. 1995. On Micro-kernel Construction. In *SOSP*.

[61] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE S&P*.

[62] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*.

[63] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave. In *HASP*.

[64] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP*.

[65] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[66] Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson. 2013. Welcome to the Entropics: Boot-time entropy in embedded devices. In *IEEE S&P*.

[67] Jason Garms Nelly Porter. 2019. Advancing confidential computing with Asylo and the Confidential Computing Challenge. https://cloud.google.com/blog/products/identity-security/advancing-confidential-computing-with-asylo-and-the-confidential-computing-challenge.

[68] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Serval: Scaling Symbolic Evaluation for Automated Verification of Systems Code. In *SOSP*.

[69] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP*.

[70] Khang T Nguyen. 2016. Introduction to Cache Allocation Technology in the IntelÂő XeonÂő Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[71] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *ATC*.

[72] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*.

[73] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. 2019. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *ATC*.

[74] Nate Graff Palmer Dabbelt. 2018. SiFive's Trusted Execution Reference Platform. https://content.riscv.org/wp-content/uploads/2018/12/SiFives-Trusted-Execution-Reference-Platform-Palmer-Dabbelt-1-1.pdf.

[75] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. 2010. Bootstrapping Trust in Commodity Computers. In *IEEE S&P*.

[76] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *ASPLOS*.

[77] Dan R. K. Ports and Tal Garfinkel. 2008. Towards Application Security on Untrusted Operating Systems. In *HOTSEC*.

[78] S. r. Ames, R. Schell, and M. Gasser. 1983. Security Kernel Design and Implementation: An Introduction. *Computer* 16, 07 (1983).

[79] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *MICRO*.

[80] Samuel Weiser and Mario Werner and Ferdinand Brasser and Maja Malenko and Stefan Mangard and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS*.

[81] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*.

[82] Shweta Shinde, Shengi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. 2020. BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof. In *USENIX Security*.

[83] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sanjit Seshia, Sriram Rajamani, and Kapil Vaswani. 2016. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI*.

[84] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *CCS*.

[85] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *CCS*.

[86] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. 2005. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. *SIGARCH Comput. Archit. News* (2005).

[87] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*.

[88] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. 2018. Privado: Practical and Secure DNN Inference. *ArXiv* (2018). arXiv:1810.00602

[89] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *ISCA*.

[90] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*.

[91] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*.