Keystone: A Framework for Architecting TEEs

Dayeol Lee
dayeol@berkeley.edu
UC Berkeley

David Kohlbrenner
dkohlbre@berkeley.edu
UC Berkeley

Shweta Shinde
shwetas@berkeley.edu
UC Berkeley

Dawn Song
dawnsong@cs.berkeley.edu
UC Berkeley

Krste Asanović
krste@berkeley.edu
UC Berkeley

Abstract

Trusted execution environments (TEEs) are becoming a requirement across a wide range of platforms, from embedded sensors to cloud servers, which encompass a wide range of cost and power constraints as well as security threat models. Unfortunately, each of the current vendor-specific TEEs makes a fixed choice in each of the design dimensions of deployability, trusted computing base (TCB), and threat model, with little room for customization and experimentation. To provide more flexibility, we present Keystone—the first open-source framework for building customized TEEs. Keystone uses a simple abstraction of memory isolation together with a programmable layer that sits underneath untrusted components, such as the OS. We demonstrate that this is sufficient to build reusable TEE core primitives, separate from platform-specific modifications and required application features. Thus, Keystone reduces the effort for platform providers and application developers to build only those security features they need. We implement Keystone on RISC-V, an open architecture that provides a straightforward way to realize the Keystone abstractions. We showcase the benefits of our design in executing standard benchmarks, applications, and kernels on various deployment platforms.

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 own forms of TEE to create a secure execution environment, commonly referred to as an *enclave*, including ARM TrustZone, Intel SGX, and AMD SEV [14, 17, 72]. On the consumer end, TEEs are now being used for secure cloud services [18, 24], databases [84], big data computations [27, 45, 89], secure banking [67], blockchain consensus protocols [25], smart contracts [37, 68, 100], machine learning [78, 96], network middleboxes [52], and so on. These use-cases have diverse deployment environments ranging from cloud servers, client devices, mobile platforms, network ISPs, IoT devices, and hardware tokens.

Unfortunately, each vendor TEE represents a fixed choice in the design space of deployability, trusted computing base (TCB), and threat model, leaving little room for customization. When platform provisioners and enclave programmers choose a TEE, they are locked into a limited set of possibilities even when their use-case demands flexibility. It is therefore not surprising that the current constraints within each vendor's framework pose challenges for customizations. For example, Intel SGX has a hard limit of 128 MB on the maximum size of secure memory [40]. This limitation has resulted in drastic slow-downs in large scale applications [18, 24, 34, 84, 90]. As much as the cloud providers want to use a secure enclave like SGX, they cannot address the limitation themselves and instead have to wait for Intel to increase the memory limit [71]. Similarly, there are several other challenges that remain unaddressed (e.g., dynamic resizing, secure I/O, side-channel attacks). Since it is in practice impossible to make changes to a proprietary ISA, the only path to solving these challenges is to re-implement TEE functionality from scratch in more open ISAs (e.g., OpenSPARC [33, 66], RISC-V [41, 87]), but each redesign requires considerable effort and only serves to create another fixed design point.

These observations motivate our work to develop *customizable TEEs* to provide a better interface between hardware manufacturers, platform providers, and enclave developers. Customizable TEEs promise quick prototyping, shorter turnaround time for fixes, adaptation to new threat models, and use-case specific deployment. We can draw an analogy with the move from hardware-based networking solutions to Software Defined Networking (SDN). We believe a similar paradigm shift in TEEs will pave the way for effortless TEE customization, allowing security to be tuned for each platform and use case. Specifically, hardware manufacturers can architect platforms that support the same baseline TEE standards, while platform providers can purchase and customize these TEEs, as in SDNs, to cater to diverse deployment scenarios and supported applications. Lastly, the enclave developers can choose from several such TEEs and lower their TCB via customization.

We observe that the first challenge in realizing this vision is the lack of a programmable trusted layer below the *untrusted* OS. Existing firmware models do not generally satisfy this requirement. Programming at the micro-code layer is near-impossible because the existing hardware is inflexible,

Keystone is available at https://keystone-enclave.org/

hard to customize, and closed-source. Thus, most current solutions emulate programmability via a bloated trusted hypervisor layer. Second, previous attempts at re-using ISA primitives to give TEE guarantees depend on the underlying hardware for memory isolation. These approaches are limited by the lack of flexibility in changing the hardware-defined isolation boundaries. For example, Intel SGX enforces isolation by a fixed-sized address range for encrypted RAM, and ARM TrustZone supports a custom-sized secure world but does not allow sharing of address spaces. Our insight is that given an inbuilt hardware memory isolation primitive and a software-programmable layer with the appropriate privileges, we can build a *customizable TEE* that is simple and flexible.

To this end, we propose Keystone—the first open-source framework for building customizable TEEs. Our design builds essential TEE primitives from minimal hardware abstractions such as memory isolation. In doing so, we make several conscious design choices to ensure modularity. We leverage the programmable layer exposed by the underlying hardware and demonstrate how one can build a framework which can provision on-demand, rich, and modular customizable TEEs. Keystone allows hardware manufacturers, platform providers, and enclave developers to configure various design choices such as TCB, threat models, workloads, and TEE functionality via our novel plugin interface. We implement Keystone on RISC-V, an open-source hardware ISA. RISC-V supports a *machine mode* (M-mode) which is equivalent to the micro-code layer of CISC architectures. M-mode is programmable and is thus a perfect place for adding TEE support. Further, RISC-V has recently added standard specifications [16] for physical memory protection (PMP)—a memory isolation primitive which allows M-mode to specify arbitrary protections on physical memory regions. Thus, RISC-V fulfills the desired requirements and is a natural fit.

The Keystone framework provides standard TEE guarantees such as secure boot and memory isolation by default. We build additional plugins to enable attestation, secure source of randomness, secure timers, system call interface, and standard `libc` support inside the enclave. We demonstrate the power of our programmable design by adding support for enclave-managed free memory and dynamic on-demand scaling of enclave size. By the virtue of our modular design, which allows us to transparently leverage additional hardware features, we build a cache side-channel defense for platforms which support cache partitioning. Lastly, we build two representative TEE enclave runtimes: a custom modular minimal-TCB runtime (*Eyrie*) and an off-the-shelf microkernel (seL4 [61]). We extensively benchmark Keystone on a generic RISC-V suite (RV8), an IO-intensive workload (IOZone), and two CPU-intensive workloads (Beebs and CoreMark). We showcase use-case studies where Keystone can be used for secure machine learning (Torch and FANN frameworks) and cryptographic tasks (sodium library) on embedded devices and cloud servers. Lastly, we demonstrate Keystone on different RISC-V platforms, three in-order cores, one out-of-order core, a QEMU emulation, and an FPGA implementation. Keystone is open-source and available online [3].

Contributions. We make the following contributions:

- *Customizable TEEs.* We define a new paradigm of TEE design wherein the platform manufacturer, platform provider, and the enclave programmer can customize what primitives and guarantees a TEE should employ.
- *Keystone Framework.* We present the first open-source framework to build customizable TEEs. Our principled way of ensuring modularity in Keystone allows us to customize the design dimensions.
- *Open-source Implementation.* Our RISC-V implementation minimizes the TCB, adapts to threat models, uses hardware features, handles varying workloads, and provides rich functionality. It comprises 6 K lines of code (LoC) TCB, of which only 1.6 KLoC was added by Keystone, with additional 4 KLoC crypto library.
- *Evaluation.* We report overheads up to 0.54% for CoreMark and Beebs, up to 0.15% for RV8, average 36-41% write/read for IOZone, and up to 128.19% for cache side-channel defense for RV8. Our experiments for real-world workloads for machine learning incur an overhead up to 7.35% for Torch with Eyrie and 0.36% for FANN with seL4. We demonstrate a Keystone-native secure remote computation application.

2 Problem & Approach

We motivate the need for customizable TEEs which can adapt to various design requirements prevalent in real deployment scenarios. We then highlight the main insights that make customizable TEEs feasible and the necessary abstractions.

Motivation. Although TEE systems are widely used, the threat models, TCB size, expressiveness, target devices, and application workloads that these solutions target are points in the design space covering only a subset of useful configurations. For instance, consider a scenario where one wants to deploy a secure IoT edge device which collects sensor data at a hub and relays data analytics to a remote server. This requires a TEE that can execute on memory and energy-constrained embedded devices while supporting secure network communications in the enclave, and ideally with low TCB. This does not overlap with any of the existing design points in the current TEE solution space. Thus, end users (e.g., platform provider, enclave developer) are left with the choice of either compromising on one of the design requirements (e.g., resort to a large TCB solution) or take the onus of building their own custom design.

Customizable TEEs. We define a new paradigm—*customizable TEEs*—wherein both the platform provider and the

enclave programmer can customize what primitives and guarantees a TEE should employ. An analogous concept to customizable TEEs is SDN, where the ability to easily program the control layer opens up far more flexibility in deployment. In our model, the manufacturer of the platform is not the sole arbiter of the TEE design and instead only provides a set of primitives for the platform provider to programatically interact with. The task of realizing a specific TEE instance is then a combination of the platform provider's implementation of the hardware interfacing and trust model, and the enclave programmer's feature requirements. Their choices are offloaded to a framework which plugs in required components and composes the expected TEE.

Customizable TEEs bridge existing gaps in the TEE design space with a framework wherein the users and researchers can independently explore design trade-offs without significant development effort. Additionally, they allow for rapid response to vulnerabilities and new feature requirements.

2.1 Abstractions for Customizable TEEs

The goal of customizable TEEs is to avoid enumeration of all possible points in the design space while allowing instantiation of any given point. We outline the important abstractions necessary for customizable TEEs. A customizable TEE has three logical actors: The *platform builder* develops and fabricates the hardware. The *platform provider* operates the hardware and makes it available to a user. The *user/enclave programmer* develops software and runs it on hardware provided by the platform provider.

Hardware-enforced Memory Isolation. The hardware must provide flexible memory isolation as a native primitive, where isolation must be customizable at runtime but only by a suitably privileged mode.

Programmable Mode Below & Above Untrusted Code. A trusted entity executing on the hardware should be able to manage and provision isolated memory regions. Specifically, the hardware should allow programming of trusted software which has higher privileges than the untrusted components (e.g, OS, hypervisor). This allows the customizable TEE framework to enforce custom isolation policies in software while offloading policy enforcement to the hardware. This mode must also have minimal non-security responsibilities to minimize TCB and present clean abstractions. Most commodity platforms implement this highest privilege mode as firmware or microcode, which do not meet our programmability requirements.

Modularity. Each layer should be independent and provide an abstraction to the layers above it. For example, the trusted software defines the isolation boundaries, while leaving the complex resource management within each isolation to the untrusted components. Further, each layer should enforce a set of guarantees, which can be easily checked by the lower layers. For example, the software manages address spaces while the hardware merely enforces configured isolation. Lastly, the framework should maintain compatibility with existing notions of privilege levels. For example, applications that are programmed to execute in the user-level code should not be expected to re-implement kernel-level functionality in order to execute in a TEE environment.

To this end, we envision a design of customizable TEEs which allows for the maximum degree of freedom with minimum efforts. More importantly, to enable this new paradigm, we need hardware which provides the above abstractions.

Customizable TEEs with RISC-V. RISC-V is an open ISA with multiple open-source core implementations [19, 32]. RISC-V currently supports up to three privilege modes: U-mode (user) for normal user-space processes, S-mode (supervisor) for the kernel , and M-mode (machine), which can directly manage physical resources such as I/O, memory, or devices. Keystone utilizes three aspects of M-mode. First, M-mode is programmable by platform operators and meets our needs for a minimal highest privilege mode. Second, M-mode controls hardware delegation of the interrupts and exceptions in the system. All lower privilege modes can only receive exceptions or use CPU cycles only when the M-mode allows it. Third, physical memory protection (PMP), a recently introduced feature in the RISC-V Privilege ISA, allows M-mode to apply simple access policies to physical memory. This enables M-mode to isolate regions of memory at runtime or disable access to memory-mapped control features. Only M-mode may configure the boundaries and access to PMP regions. Thus, RISC-V provides the abstractions required to build customizable TEEs.

2.2 Design Dimensions for Customizable TEEs

We next outline design dimensions of customizable TEEs.

Platform Support & Workloads. There is a stark split in the TEE platforms and the workloads they support. TrustZone is deemed better suited for embedded devices and mobile platforms, partly because most of these devices have ARM processors. Intel SGX has become a popular TEE for server applications [17]. AMD SEV is targeted at securing large workloads and has seen adoption in virtual machines [14]. As opposed to these workload-targeted platform designs, Keystone can support various workloads and frees platform providers and enclave developers from inadvertent lock-in to a specific TEE design based on their workloads.

TCB Size & Expressiveness. Even within a TEE, there is a diverse set of functionality one may want to execute. For example, server applications can vary from simple in-memory databases which do not use fork [18] to complex databases with load-balancing via fork-exec [34, 84]. Given the various enclave programming frameworks available, a cloud service

provider may want to choose the one with maximum expressiveness to support a large fraction of user applications. The downside of this approach is that the enclave TCB increases with complex functional support, irrespective of whether the application uses the functionality. Thus, the same database can be executed with a TCB of few thousand LOC [18] or a million LOC [24, 34], depending on the design choice of the underlying programming infrastructure. With Keystone's modular design, the platform provider can instantiate a TEE with minimal TCB required to execute the enclave. The enclave programmer can further optimize the TCB by on-the-fly customization of what is included in the enclave based on the needs of the enclave logic.

Threat Models. Enclaves are primarily designed to prevent an attacker from compromising the confidentiality and integrity of the enclave logic. However, the attack surface depends on the environment which hosts the enclaves. A *physical attacker* can intercept, modify, or replay signals that leave the chip package. Keystone assumes that the physical attacker is not capable of intercepting or modifying on-chip signals. A *software attacker* can control host applications, the untrusted OS, network communications, launch adversarial enclaves, arbitrarily modify any memory not protected by the TEE, or it may add/drop/replay enclave messages. A *side-channel attacker* can glean information by passively observing interactions between the trusted and the untrusted components. Not all platforms will be potentially vulnerable to all side-channel attacks because they may not contain shared/multiple caches and out-of-order execution. A *denial-of-service attacker* can attempt to take down the enclave or the host OS at any time. Keystone allows denial-of-service attacks against enclaves because traditionally the OS can refuse services to user application at any time. On the other hand, a DoS attack from the enclave is undesirable.

Keystone allows flexible usage of the underlying defense mechanisms to adapt against a subset of the threat model. One might argue that the TEE should always defend against as many possible threat models at all times. However, if both the platform provider and the enclave developer are sure that a certain class of attacks are out of scope, turning off a defense mechanism can return significant performance or power improvements. For example, if the user is deploying TEEs in its own private data centers or home appliances, a physical attacker may not be a threat. Current individual TEE proposals [17, 26, 41, 72] have distinct threat models, but are restricted to a specific choice.

Any Keystone TEE instance will guarantee confidentiality and integrity of enclave code and data from a software attacker.

In summary, the Keystone allows platform operators and developers to customize design aspects of TEEs.

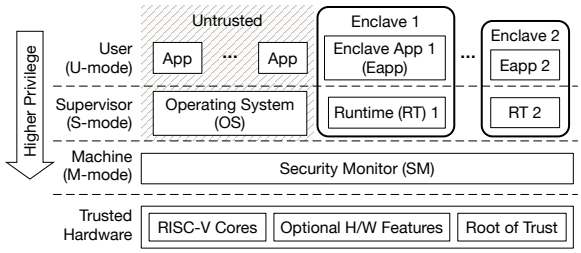

Figure 1. Keystone system with host processes, untrusted OS, security monitor, and multiple enclaves (each with runtime and eapp)

3 Keystone Design

We discuss the Keystone design components, outline our design principles, and how we enable customizable TEEs.

3.1 Overview

The Keystone usage model is that an untrusted OS executes an untrusted host application, which in turn launches one or more trusted enclaves (similar to SGX usage model). In addition, Keystone introduces two new components, namely security monitor and runtime. Figure 1 shows the detail of the interactions between these components.

Building TEE Primitives from Hardware Abstractions. Keystone uses the isolation and programmable mode provided by RISC-V to build critical TEE guarantees. Specifically, we design a security monitor (SM) which enforces the TEE guarantees across the entire platform. The SM executes in M-mode and is programmed in C/assembly. For customization, the platform operator specifies the SM configuration and trusts that the SM will enforce all relevant security boundaries between enclaves and the host/OS. Such a design allows Keystone to transparently extend the guarantees given by the hardware. For example, if the device has additional features (e.g., cache partitioning, crypto engine, source of randomness), the SM can enforce additional protection or improve the performance without changing the rest of the layers (e.g., OS, user applications). More importantly, it ensures that untrusted layers (e.g., OS) executing above the SM cannot circumvent enforcement.

On-demand, Rich, and Modular Programming Model. A Keystone enclave comprises a runtime (RT) which executes in S-mode and an enclave application (eapp) executing in U-mode. Both RT and eapp are part of the enclave address space which is isolated from the untrusted OS and other user applications. Most importantly, the RT is specified by the user and can be configured according to the needs of the target eapp it needs to support. The RT manages the lifecycle of the user code executing in the enclave, manages memory, services syscalls, etc. Specifically, the RT uses a limited set of API functions exposed by M-mode via the RISC-V supervisor binary interface (SBI) to exit or pause the enclave (See Table 1 for a full list of SM SBI calls). Further, the RT can

SM Func.	Caller	Description
`create`	OS	Isolate, create, validate, and measure the enclave.
`run`	OS	Start (enter) enclave execution and boot RT
`resume`	OS	Resume (re-enter) enclave execution
`destroy`	OS	Clean and release enclave memory.
`extend`*	OS	Extend enclave memory
`stop`	Enclave	Stop (exit) enclave execution
`exit`	Enclave	Terminate (exit) and invalidate the enclave
`attest`	Enclave	Sign the enclave measurement/data and retrieve attestation reports
`random`	Enclave	Get cryptographically secure random values

Table 1. The SBI functions the SM provides. Extending enclave (*) is optional (see Section 4.1).

request the SM to perform operations on behalf of the eapp (e.g., attestation, get random values) via the SBI. Note that each enclave instance may choose its own RT configuration which is not shared with any other enclave. This separation allows for significant modularity in application support for Keystone while decreasing the per-eapp TCB.

Maintaining Developer-friendly Interface. Keystone can execute entire application logic or developer-specified parts as an eapp. Keystone does not allow the eapp to directly access any untrusted memory or use untrusted OS APIs (e.g., system calls). This restriction may break the common program semantics or, even worse, force the eapp programmer to re-design their application specifically for Keystone. Our design goal is to avoid any such additional effort. Thus, we follow the design principle of maintaining strict compatibility with existing user-code programming notions. We bridge the gap between Keystone restrictions and traditional program via the RT. Specifically, an eapp may continue to assume a fully functional OS, and the RT can provide the necessary safe functionality. This design allows for the use of unmodified eapp programs in Keystone with a sufficiently feature-rich RT.

3.2 Keystone Memory Isolation Primitives

We describe how the simple design of PMP is sufficient to implement flexible memory isolation of Keystone enclaves.

Background: RISC-V Physical Memory Protection. Keystone uses physical memory protection (PMP) feature provided by RISC-V. PMP allows restrictions on the physical memory access of S-mode and U-mode to certain regions defined via *PMP entries*.[2] Figure 2 shows the PMP details. Each PMP controls the U-mode and S-mode permissions to a customizable region in the physical memory.[3] Specifically,

[2] `pmpaddr` and `pmpcfg` CSRs are used to specify PMP entries

[3] Currently, a RISC-V processor may have up to 16 PMP entries, which can be configured by M-mode.

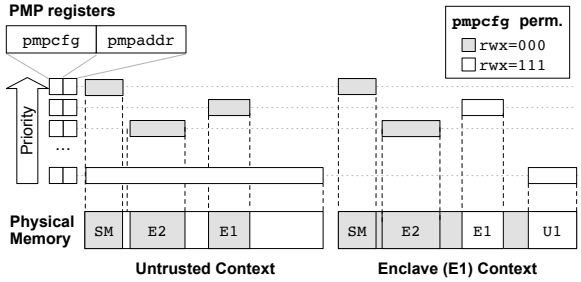

Figure 2. How Keystone uses RISC-V PMP for the flexible, dynamic memory isolation. The SM uses a few PMP entries to guard its own memory (`SM`) and enclave memories (`E1`, `E2`). Upon enclave entry, the SM will reconfigure the PMP such that the enclave can only access its own memory (`E1`) and the untrusted buffer (`U1`).

the PMP address registers encode the address of a contiguous physical region. The PMP configuration bits specify the r-w-x permissions and two addressing mode bits. Each permission bit set determines if U-mode or S-mode can operate on the memory region. PMP has three addressing modes such that the base address and the size of the memory region can be both a power of two or an arbitrary number. PMP entries are statically prioritized where a lower numbered PMP entry has higher matching priority—a lower numbered entry can over-rule the permission bits of a higher entry.

Enforcing Memory Isolation via SM. PMP makes Keystone memory isolation enforcement flexible for three reasons. First, one can configure enclave regions proportional to the number of PMP registers in the CPU. Thus, multiple discontiguous enclave memory regions can coexist, instead of reserving one large memory region shared by all enclaves. Second, PMP supports flexible addressing modes. A memory region can cover a region as small as 4 bytes, or as large as the entire DRAM. This enables Keystone enclaves to utilize page-aligned memory with an arbitrary size. Lastly, PMP entries can be dynamically reconfigured during execution. Instead of occupying a set of memory regions permanently, Keystone can dynamically create a new region, or release a region to the operating system as needed.

Figure 2 shows how Keystone utilizes them for memory isolation. Specifically, when the SM boots, it configures the first PMP entry (highest priority) to cover its own memory region containing the code, stack, and the secure data such as enclave metadata and keys. Further, the SM disallows access to its memory from U-mode and S-mode. The SM then configures the last PMP entry (lowest priority) to cover all memory and with all permissions enabled, such that the OS can access the non-SM memory.

When a host application requests to create an enclave, the OS is in charge of allocating a contiguous physical region. Upon successful creation of an enclave, 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 2), the OS or

any other user processes cannot access the enclave region. Further, the enclave regions are not allowed to overlap with each other or with the SM region.

When the CPU transfers the control to enclave execution, the SM flips the PMP permission bits of the relevant enclave memory region. At the same time, the SM also removes all permissions from the OS PMP entry to protect the OS memory from the enclave. Both these steps together allow the enclave to access its own memory safely. When the CPU performs a context switch from enclave to non-enclave, SM flips the permission bits and isolates the enclave memory 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, Keystone adds a PMP entry to disallow everyone from accessing the enclave. These changes during creation must be propagated to all the cores via inter-processor interrupts (IPIs). This ensures that the other cores are disallowed from accessing the enclave. During enclave execution, changes to the PMP entries (e.g., context switches between enclave and host) are local to the core executing it and need not be propagated to other cores. When Keystone destroys the enclave, all the other cores are notified to update their PMP entries. There are no other times that PMPs must be synchronized via IPIs.

3.3 Other Keystone TEE Primitives

We outline the remaining standard TEE primitives supported in Keystone. Keystone provides well-defined interfaces for each of the following components. Our implementation directly supports secure boot, trusted timer, and secure source of randomness. For the remaining primitives, Keystone provides stubs which can be used to integrate existing solutions.

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 measures the SM image, generates a fresh attestation key from a cryptographically secure source of randomness, and stores it to the SM private memory. The root-of-trust then signs the measurement and the public key with a hardware-visible secret. These are standard operations, can be implemented in numerous ways [59, 64], and Keystone does not rely on a specific implementation. For completeness, currently, Keystone simulates secure boot via a modified first-stage bootloader that performs all the above steps.

Secure Source of Randomness. The Keystone framework provides access to a trusted source of randomness via the SM SBI (`random`) which returns 64 bits of random value. Keystone should use a hardware source of randomness to fulfill requests if available or can fallback to well-known options (e.g., CPU jitter) or exotic methods (e.g, DRAM decay [73]).

Trusted Timer. Keystone allows the enclaves to access the timer registers maintained by the RISC-V hardware via the `rdcycle` instruction. More importantly, these timers are not writable, and may not be modified by the untrusted OS. The SM also supports a set of standard timer SBI calls.

Remote Attestation. Keystone SM performs measurement and attestation based on the provisioned key. When an enclave asks for an attestation report, the SM signs the enclave measurement with the attestation key and copies the signature to the enclave memory. Keystone uses a standard scheme to bind the attestation with a secure channel construction [48, 64]. The SM allows the enclave to include limited arbitrary data (e.g., Diffie-Hellman key parameters) to be included in the signed attestation report. Key distribution [15], revocation [54], attestation services [56], and anonymous attestation [28] are orthogonal and can be implemented on top of the Keystone.

Secure Delegation. Keystone delegates traps (i.e., interrupts and exceptions) raised during enclave execution to the RT via RISC-V hardware interrupt delegation registers. While doing so, Keystone ensures that it does not directly leak any enclave state and does not allow the OS to start executing at arbitrary entry points in the enclave. For user-defined exceptions, the RT invokes the appropriate handler inside the enclave. For others, the RT forwards them to the untrusted OS via the SM. The SM performs the context switch by (a) saving the enclave context including general purpose and control-and-status registers (CSRs); (b) returning the execution control to the untrusted OS to handle the trap. The OS can the return the execution control to the enclave via the SM SBI, where the SM examines the OS return values, sanitizes them to ensure they are safe to deliver to the enclave, and resumes the enclave execution.

Monotonic Counters & Sealed Storage. Enclaves may need monotonic counters for protection against rollback attacks and versioning [40]. Keystone framework can support monotonic counters by keeping the counter state in the SM memory. This is safe because the OS cannot overwrite or remove the data from the M-mode where the SM executes. In the future, Keystone interface can interact with non-volatile memory for persistent counters [91] as well as TPM and NVRAM for better latencies and durability [81, 92]. Going forward, Keystone can support sealed storage such that the eapp can use untrusted storage memory or device store encrypted content which is tied to the enclave identity.

3.4 Enclave Lifecycle

We summarize the end-to-end life cycle of a Keystone enclave in Figure 3, including the key steps in the enclave lifetime and the corresponding PMP changes.

Enclave Creation & Execution. During enclave creation, Keystone measures the enclave memory to ensure that the

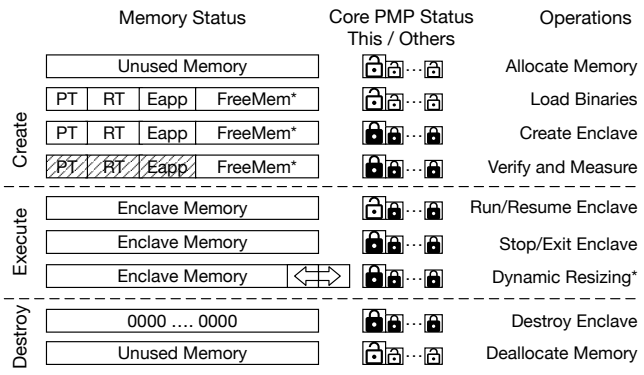

Figure 3. Lifecycle of an enclave. The enclave memory and the corresponding PMP entry status (accessible or not) are shown per each operation. For PMP status, *This* means the PMP status of the core performing the operation and *Others* is PMP of other cores.

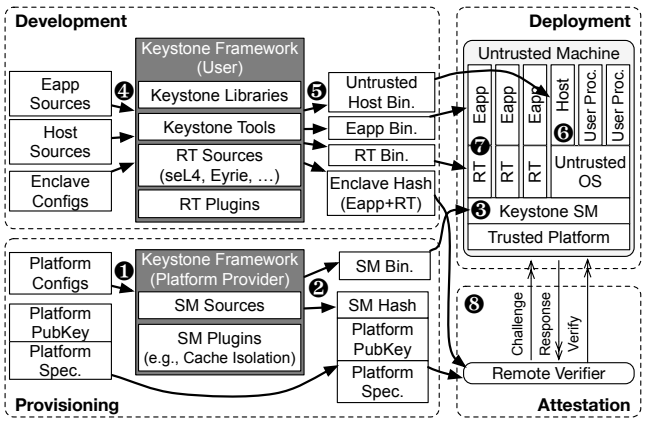

Figure 4. Keystone End-to-end Overview. ❶ The platform provider configures the SM. ❷ Keystone compiles and generates the SM boot image. ❸ The platform provider deploys the SM. ❹ The developer writes an eapp and configures the enclave. ❺ Keystone builds the binaries and computes measurements. ❻ The untrusted host binary is deployed to the machine. ❼ The host deploys the RT and the eapp and initiates the enclave creation. ❽ A remote verifier can perform attestation based on the known platform specifications, keys, and SM/enclave measurements.

OS has loaded the enclave binaries correctly to the physical memory. Keystone uses the initial virtual memory layout for measurement as the physical layout can legitimately vary (within limits) across different executions. Specifically, the SM expects the OS to initialize the enclave page tables and allocate physical memory for the enclave. During the enclave creation, the SM uses the initial page table provided by the OS and SM checks if there are invalid mappings, ensures unique virtual to physical mapping. Then the SM hashes page content along with virtual addresses and configuration data. For execution, the SM sets the PMP entries and transfers control to the enclave entry point.

Enclave Destruction. The OS or the enclave itself may initiate enclave tear-down. During this phase, the SM clears the enclave memory region before it returns memory to the OS. All enclave resources, PMP entries, and enclave metadata are cleaned and then freed.

4 Keystone Framework

We describe the usage, programming, and deployment model of our novel Keystone framework. In doing so, we highlight the programmability aspect of our framework and how our modular design allows us to transparently extend Keystone functionality with *plugins*. Figure 4 shows the steps involved in an end-to-end eapp deployment.

The platform builder provides a RISC-V core with PMP support, secure key storage, and may add other hardware functionality (e.g., source of secure randomness, caches, memory encryption engines). The platform provider deploys RISC-V cores via this off-the-shelf RISC-V hardware. To instantiate customizable TEEs with Keystone, the provider configures the SM for core TEE primitives (e.g., memory isolation). They may also configure SM plugins which expose additional functionality of the platform.

The Keystone framework provides RTs which interface with the platform provider's SM to provide user code abstractions to an eapp. Additionally, RT plugins may be used to provide syscall interfaces and memory management features. The enclave programmer writes eapps, and can configure what platforms they expect to execute on, what SM and RT it accepts, and the set of plugins the eapp requires. The developer specifies all such requirements in an eapp configuration file and carries out the rest of the development procedure as any other non-enclave application. The developer can choose to execute existing non-enclave applications inside an enclave without any modification. Otherwise, they can use the Keystone SDK which provides common enclave utilities to write native Keystone eapps.

Keystone acts as an intermediary between the above entities. Specifically, our framework takes in the eapp source-code or binaries, along with a configuration specifying the RT, architecture features, and the plugins that the enclave will require at the RT level. The Keystone toolchain can then be used to compile a RT for the eapp, as well as a corresponding host application. Developers can deploy their eapp by launching a Keystone enclave on a platform of their choice. The enclave can send a signed enclave measurement (an attestation report) to remote parties and prove that the platform provider is indeed executing on authentic hardware with a known SM, and the expected RT and eapp.

Plugin Support. Keystone provides all the basic building blocks for customizable TEEs described in Section 3.2. We point out that a programmable M-mode along with per-enclave RTs allows Keystone to enable a variety of memory management, functionality, and security features in the SM as well as the RT. We now demonstrate various Keystone plugins we have built on top of Keystone building blocks.

4.1 Enclave Memory Management Plugins

Keystone enclaves may occupy anywhere from several tens of KBs up to several GBs as long as the OS can reserve a contiguous physical memory region for the enclave at the time of creation.[4] This flexibility in enclave physical memory size enables a wider range of applications and use cases for Keystone. For example, one can run batch workloads such as deep learning inferencing with a large enclave size in order to avoid paging overheads. In the default Keystone design after the OS gives contiguous physical memory to an enclave, the enclave manages this memory using the RT S-mode privileges. Thus, Keystone does not rely on the OS for virtual memory management other than the limited initial mappings. Note that, the enclave has to specify the maximum physical memory size it needs, as well as the size of the stack, heap, code, and data section so that the OS can create initial virtual-to-physical mappings for the enclave. This requirement, although suitable for small eapps, may limit some larger eapps. To this end, we describe several memory management plugins which add additional virtual memory support to the RT. The eapp developer, if required, can optionally enable these plugins in the RT to alleviate the default-mode restrictions. Relevant SM plugins must be enabled by the platform provider.

Free Memory. We introduce a plugin which allows the enclave to reserve physical memory without mapping it to any virtual address by specifying a new region called *free memory*. Free memory is not associated with any virtual address during initialization and is not included in the enclave measurement. Instead, the RT simply ensures that free memory is zeroed before beginning the eapp execution. When the eapp asks for more stack or heap memory during its execution, the RT uses pages from the free memory region to dynamically allocate stack or heap pages, create anonymous mmaps, and execute binaries which do not have a static virtual layout. This free memory plugin allows the eapp to make better use of the fixed-size enclave memory.

Dynamic Resizing. The size of the physical memory that an eapp uses at runtime depends on the workloads. For example, certain applications (e.g., machine learning inference) may have a fixed memory footprint, whereas other applications (e.g., databases) have arbitrary memory usage. Asking the eapp developer to specify the maximum enclave size and then statically pre-allocating the physical memory for it can prevent scaling and compatibility. To this end, Keystone allows the RT to request dynamic changes to the physical memory boundaries of the enclave. Specifically, the RT may request that the OS make an `extend` SBI call to add physical pages to the enclave memory region as able. If the OS succeeds in such an allocation, the SM increases the size of the enclave and notifies the RT. The RT can then use these pages as a part of the free memory and allocate them to satisfy the eapp requests (e.g., `brk`). Note that by combining with the free memory plugin the eapp can start with a small static stack and heap size in the initial configuration and then scale resources based on workloads instead of pre-determining all the memory needs. Dynamic resizing may be a side-channel for some applications and is entirely optional.

[4]By default, our untrusted kernel uses the Linux contiguous memory allocator (CMA)

4.2 Functionality Plugins

Edge Call Interface. The eapp cannot access non-enclave memory in Keystone, so if it needs to read or write data outside the enclave (e.g., sending attestation reports), the RT can perform *edge calls* on their behalf. Our edge call plugin is functionally similar to RPC. Specifically, an edge call consists of an index to a function implemented in the untrusted host application and the parameters to be passed to the function. The RT 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 the RT to have access to a buffer shared with the OS by: (a) before the SM creates the enclave, the OS allocates a shared buffer in its memory space and passes the address to the SM; (b) the SM then binds the address to the enclave so that the enclave RT can access the memory; (c) the SM uses a PMP entry to control the shared buffer permissions. We enforce all edge calls passing through the RT by not making the shared memory virtual mappings available to the eapp.

System Calls. System calls operate as a subset of edge calls and Keystone piggybacks on the mechanism described above to tunnel calls between the OS and the eapp. Specifically, the user application invokes the system call on behalf of the eapp, collects the return values, and forwards them to the eapp. The plugin forwards only specific defined system calls. For other syscall operations (mmap, brk, random generation, etc) Keystone either invokes a SM interface or performs the operation in the RT and returns the values to the eapp.

Multi-threading. Keystone supports multi-threaded eapps by scheduling all the threads on the same core. We do not support hyper-threading or parallel multi-core execution of the enclave yet. The RT performs the thread context switch, and all the thread local storage is protected by the enclave.

4.3 SM Plugins

We demonstrate a platform-specific side-channel defense which can be completely implemented in the SM. If the platform builder can produce hardware with the required features, the platform provider can configure its usage in deployment. Such plugins are implemented via our platform-specific build options for the SM.

Cache Partitioning. If the platform has a shared cache, it renders the enclaves potentially vulnerable to cache side-channel attacks from the untrusted OS and other applications in other cores. Fortunately, the FU540-C000, one such RISC-V SoC with a shared L2 cache, provides a highly customizable L2 cache controller with a *waymasking* primitive similar to Intel's CAT [75]. Our plugin uses the controller to partition the cache such that the enclave and the adversary do not share any cache ways. Specifically, the SM plugin combines waymask management and PMP to way-partition the L2 cache transparently to the OS and the enclaves.

5 Implementation

We implement our SM on top of the Berkeley Boot Loader (bbl) [8]. We provide a tool which generates the expected measurements of the eapp and RT. We build a simple RT called **Eyrie** to perform the minimal task of loading and executing enclaves and provide various plugins. We port the **seL4 microkernel**[61] to Keystone by making 290 LoC changes to the original seL4 code for booting, initializing memory, and interrupt handling in Keystone. Note that Keystone does not limit the developer choice to these two RTs, they can write a custom RT if necessary.

We ensure that Keystone can execute on various RISC-V platforms and simulation environments with PMP support. First, Keystone executes on real CPUs (e.g., FU540 [1]) and can be deployed on various IoT devices [2]. Second, Keystone can execute on FPGA implementations of popular RISC-V cores [19, 32] both locally and on commodity cloud platforms which support FPGA instances. This allows developers to perform data-center scale benchmarking and testing. Third, we integrate Keystone changes to RISC-V QEMU [7], which allows developers to debug the RT and the eapp before deployment. Finally, we add support for executing Keystone on a RISC-V cycle-accurate simulation platform [57] to introspect the execution of the eapp, RT, and SM. Keystone is available on github at https://keystone-enclave.org/.

6 Evaluation

We aim to answer the following questions in our evaluation:

(***RQ1***) **Modularity.** Is the Keystone framework easily modified, and modular enough to be viable in different configurations for real applications?

(***RQ2***) **TCB.** What is the TCB size of Keystone in various deployment modes?

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

(***RQ4***) **Real-world Applications.** Does Keystone provide sufficient expressiveness to execute various eapps?

Experimental Setup. We use four different platforms for our experiments; SiFive HiFive Freedom Unleashed board [2] with a closed-source RISC-V SoC (FU540), and three open-source RISC-V processors: small Rocket (Rocket-S), default Rocket (Rocket) [19], and Berkeley Out-of-order Machine (BOOM) [32] (Table 2). All cores are configured to execute at one GHz frequency. For the open-source processors, we instantiate each platform with an FPGA-synthesized core using FireSim [57]. The host OS is buildroot-based Linux (kernel 4.15). All the evaluation was performed on SiFive HiFive Freedom Unleashed and the data is averaged over 10 runs unless otherwise specified. Additional data is available [3].

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.

Component	Runtime	SM
Base	1730	1000
Memory Isolation	—	530
Free Memory	310	—
Dynamic Memory	100	70
Edge-call Handling	300	30
Syscalls	190	—
`libc` Environment	50	—
IO Syscall Proxying	260	—
Cache Partitioning	—	300

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

6.1 Modularity & Platform Support

Our configuration-based interface allows the eapp developer to turn on / off various plugins. We outline the qualitative measurement of this flexibility for extending features, reducing TCB, and platform deployability.

Extending RTs. First, our exemplary modular RT, Eyrie, has a number of plugins for running eapps. Table 3 shows the LoC for the all optional plugins in Eyrie, with a current maximum TCB of 2940 LoC. Many of these modifications (e.g., additional edge-call features) require no modifications to the SM, and the eapp developer may enable or disable them as needed. Adding syscall proxy support requires no SM modifications and uses the edge-call interface to pass data to the host OS. Similarly, future additions such as ports of interface shields may be implemented exclusively in the RT. Second, we add support for a new RT by porting seL4 to Keystone and use it to execute various eapps (See Section 6.3). Keystone passes all seL4 test suite with $< 1\%$ overhead.

Extending the SM. The advantage of an easily modified SM layer is noticeable when features require interaction with core TEE primitives like memory isolation. Our first example is an SM plugin for secure cache-partitioning on

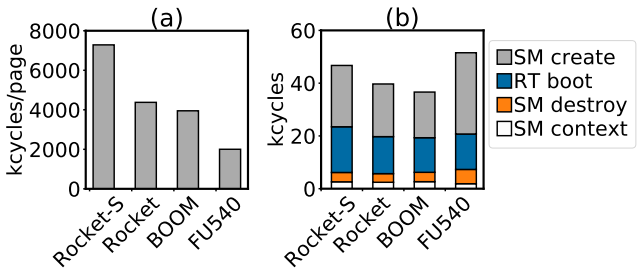

Figure 5. Breakdown of operations during the enclave life-cycle. (a) shows enclave validation and hashing duration, and (b) shows the breakdown of other operations. (b) does not include duration of size-dependent operations such as measurement in `create` (Shown in (a)) and memory cleaning in `destroy` (4K-11K cycles/page).

the FU540 L2 cache. It requires only 200 LoC in a platform-specific module and is completely compatible with other SM or RT features without additional changes. No RT or eapp changes are necessary to enable the cache partitioning. As a second example, we added the ability to resize enclaves during execution as needed. This required less than three engineer-days of work (across the driver, SM, and RT). It involved only 80 LoC changes in the SM, 100 LoC changes in the RT, and no changes to the eapps.

TCB Breakdown. Keystone implementation comprises SM, RT, untrusted host application, eapp, and the helper libraries from the SDK. Only a fraction of this implementation is a part of the TCB. First, the SM is 15, 400 LoC, of which the BBL is 80 LoC, the optional soft float implementation is 4800 LoC, the cryptographic library for measurement is 4400 LoC (in both secure boot and SM). Trap handling, emulation, and utilities are 4600 LoC. Only the remaining 1600 LoC is added by Keystone. Second, the minimal RT Eyrie comprises 3000 LoC. Table 3 shows the TCB breakdown of each component if the eapp enables the plugin. Although Keystone requires the driver, host application, and host OS, these components are not trusted and do not contribute to the total TCB. Further, the eapp developer can tune the enclave TCB. Thus, Keystone TCB amounts to 10, 000 LoC which is within the realm of formal verification in the future.

6.2 Benchmarks

We conduct a detailed performance analysis of Keystone across several configurations. In particular, we first show a breakdown of the commonly required operations on each of the four platforms in Table 2. We instrument the Eyrie RT and SM for collecting timing data on the FU540 and use FireSim measurements for the three FPGA platforms. This experiment is designed to identify bottlenecks in Keystone and explain the performance of macro benchmarks in the subsequent evaluation. Next, we show performance overhead with four standard benchmarks: **Beebs** (CPU-bound), **CoreMark** (CPU-bound), **RV8** (well-known RISC-V benchmark), and **IOZone** (I/O and FS) on the FU540. We then report the performance impact of the cache partitioning plugin with RV8 as an example of Keystone trade-offs.

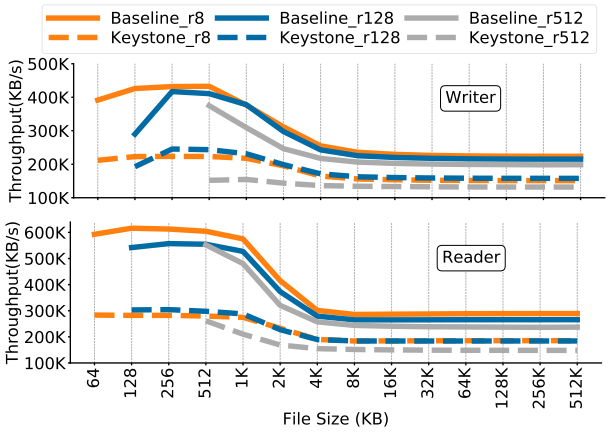

Figure 6. IOZone file operation throughput in Keystone for various file and record sizes (e.g., r8 represents 8KB record). We only show write and read results due to limited space, other data at [3]

Common Operations. Figure 5 shows the breakdown of various enclave operations. The initial validation and measurement take 2M-7M cycles/page (Figure 5a). It dominates the cost of all other creation operations because Keystone uses a software implementation of SHA-3 [12], which can be replaced with an optimized version or (preferably) dedicated cryptographic hardware. Similarly, the attestation operations are dominated by the unoptimized ed25519 [6] signing software implementation (not shown in the graph, 0.7M-1.6M cycles). Both these steps are one-time operations per enclave execution and do not impact the execution time of eapp code after initialization. Next, we highlight the latency of an SM context switch (Figure 5b) from enclave to the host OS, which is the most frequent SM operation during the eapp execution. Keystone currently takes 1.8K-2.6K cycles depending on the platform. Lastly, FU540 (4-core) takes more cycles for enclave creation and destruction than other platforms (single-core), which can be attributed to multi-core PMP synchronization.

Standard Benchmarks. We use several unmodified standard benchmarks as eapps to demonstrate the CPU overheads and the impact of I/O proxy in Keystone.

Beebs, CoreMark, and RV8. As expected Keystone incurs no meaningful overheads (±0.7%) for pure CPU benchmarks as they run unmodified as eapps.

IOZone. Next, we use IOZone[77] to measure the overheads for various file operations requested by an eapp. We tunnel all the file related system calls to the untrusted OS because the target file system is situated in the untrusted host. Figure 6 shows the throughput plots of common file-content access patterns. Although the trends are roughly the same, Keystone experiences high throughput loss for both write (avg. 36.2%) and read (avg. 40.9%). There are three factors which amount to the high overhead: (a) all data crossing the privilege boundary is copied via the untrusted buffer

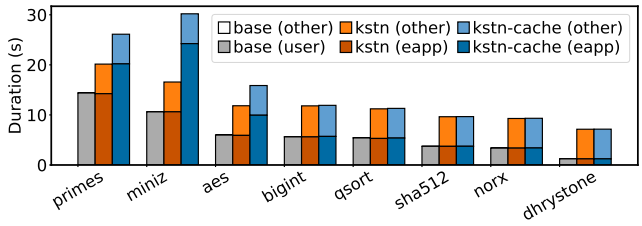

Figure 7. Full-execution time comparison for RV8 benchmarks. Each bar consists of the duration of the application (`user` or `eapp`), and the other overheads (`other`). Keystone (`kstn`) and Keystone with cache partitioning plugin (`kstn-cache`) are compared with the native execution in Linux (`base`).

(doubling the number of buffer copies), (b) each call requires the RT to go through the edge call interface, incurring a constant overhead, and (c) untrusted buffer contends in cache with file buffers, incurring more 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 apt to demonstrate cache partitioning impact. We configure a 50-50 cache partitioning, where the enclave gets 8 of the 16 ways in the FU540 L2 cache. Figure 7 shows the performance overheads for RV8 with respect to native execution when the cache partitioning turned off and on. We observe that small working-set tests show small performance overheads from cache flush on context switches, whereas large working-set tests (primes, miniz, and aes) show up to 128.19% overhead as expected. However, the enclave initialization latency is not affected.

6.3 Case Studies

We present three case-studies where Keystone proves to be suitable for deploying a TEE and demonstrate how Keystone can be adapted for a varied set of devices, workloads, and application complexities. To this end, we chose several example applications: machine learning workloads for the client and server-side usage, a cryptographic library porting effort for varied RTs, and a small secure computation application written natively for Keystone. All the evaluation for the case-studies was performed on SiFive HiFive Freedom Unleashed.

Porting Efforts & Setup. We do not make any change to the application code logic, all applications have their test configurations and arguments hard-coded for consistency. In the process of porting these applications, we add partial support in Keystone for standard libc implementations. Specifically, Keystone supports eapps statically linked with `glibc` and `musl libc` in the Eyrie RT. We port a widely used cryptographic library namely libsodium to both Eyrie and seL4 RT with zero developer effort. We measure both the setup time and internal execution time of eapps.

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

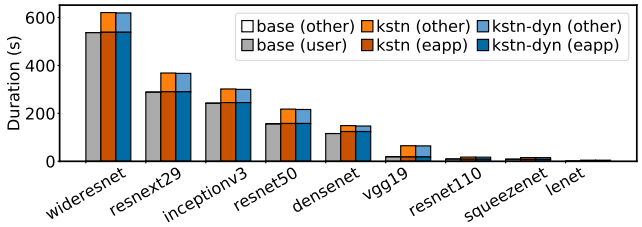

Figure 8. The inferencing duration comparison results for various Torch models. Each bar consists of the duration of the application (`user` or `eapp`), and the other overheads (`other`). Keystone (`kstn`) and Keystone with the dynamic resizing plugin (`kstn-dyn`) are compared with the native execution in Linux (`base`).

Case-study 1: Secure ML Inference with Torch and Eyrie. We execute the Torch library with Eyrie RT to perform inference using nine models with increasing sizes of parameters and layers on Imagenet dataset [44]. Specifically, we execute a total of 15, 755 and 15, 400LoC of TH [10] and THNN [13] libraries from Torch framework compiled with `musl libc`. Each model implementations comprise an additional 230 to 13, 399 LoC of model-specific inference code obtained from the ONNX model converter in Privado [96]. Table 4 shows the details of each model. We perform two sets of experiments: first we execute the model inference code for each of the nine networks with static free memory allocation where we specify the maximum enclave size; second, we turn on the dynamic resizing plugin so that the enclave extends its size on-demand when it executes. Figure 8 shows the performance overheads for these two configurations with respect to the native execution without Keystone. We measure the time for initializing the enclave (e.g., loading, hashing, page setup) and executing the eapp logic.

Initialization Overhead. We observe that the initialization for Keystone is noticeably expensive for both static enclave size and dynamic resizing and is proportional to the eapp binary size. This is expected because the cost of hashing the enclave pages dominates the execution costs, but loading the binaries and setting up virtual memory are comparatively cheaper operations (as shown in Section 6.2). Another observation is that the dynamic resizing reduces the initialization latency by 2.9% on average as the RT does not need to allocate free memory during enclave creation.

eapp Execution Overhead. We report an overhead between −3.12% to 7.35% for all the models with both static enclave size and dynamic resizing. Specifically, LeNet is faster while Densenet is the most expensive in both cases. We explain this with three phenomenons. First, Keystone loads the entire binary in physical memory before it beings eapp execution. Thus, the eapp does not incur any page faults, whereas the baseline page faults when it loads the binary pages during the execution. This explains why smaller sized networks, LeNet in this case, execute faster in Keystone compared to the baseline. Second, the overhead is primarily proportional to the number of layers in the network, because a large number of layers leads to more memory allocations (for input and output tensors of each layer). This results in an increase in `mmap`, `brk` syscalls. We see this slowdown for large size allocations because Eyrie RT's custom `mmap` implementation is not as fast as the baseline kernel. We verified that this is indeed the source of overheads by hand-coding a small test which makes large calls. This explains why Densenet, which has the maximum number of layers (910), suffers from larger performance degradation. In summary, Keystone incurs acceptable overheads for long-running ML applications with a fixed one-time startup cost and the dynamic resizing plugin is useful for larger eapps.

Case-study 2: Secure ML with FANN and seL4. FANN, a minimal ANN implementation comprising a total of 8,462 C/C++ LoC, is suitable for embedded devices. We use seL4 RT, which is also widely used for embedded devices, to train and test a simple XOR network. We report an overhead of 0.36% for the end-to-end execution over baseline seL4 measured without Keystone. This shows that Keystone can be used for small devices such as IoT sensors and cameras to train models locally as well as flag event by executing model inference.

Case-study 3: Secure Remote Computation. We design and implement a simple secure server eapp (and remote client) that counts the number of words in a given message, and execute it with Eyrie and no plugins. The eapp first performs attestation using libsodium and establish a secure channel with the remote client bound to the attestation report. The eapp then polls the host application for encrypted messages using the edge-call library, processes them inside the enclave, and returns an encrypted reply to be sent to the client. Our secure channel code using libsodium is 60 LoC, the edge-wrapping interface is 45 LoC and the rest of the server eapp is 60 LoC. The generic host is 270 LoC and the remote client is 280 LoC. Keystone takes 45K cycles for a roundtrip with an empty message which includes secure channel and message passing overheads. It takes 47K cycles between the host getting a message and enclave notifying the host to send a reply.

7 Related Work

Keystone is the first framework for customizing TEEs, equivalent to research areas such as SDNs [31], microkernels [46], and library OSes [88]. Here, we survey existing TEEs.

TEE Architecture Extensions. Several TEEs have been introduced by multiple vendors and researchers for protection against untrusted OSes, of which three major TEEs are directly related to Keystone: (a) Intel Software Guard Extension (SGX) executes user-level code in an isolated virtual address space backed by encrypted RAM pages [72]; (b) ARM TrustZone divides the memory into two worlds (i.e., normal vs. secure) to run applications in protected memory [17]; and (c) Sanctum uses the memory management unit (MMU) and cache partitioning to isolate memory and prevent cache side-channel attacks [41]. Apart from these, several TEEs explore designs options at various layers such as hypervisors [23, 35, 36, 39, 49, 53, 58, 66, 69, 94, 99, 101], physical memory [26, 33, 63, 65, 70, 79, 86], virtual memory [38, 42, 47, 87], and process isolation [43, 50, 83, 85, 93, 95].

Re-purposing Existing TEEs for Modularity. One way to meet Keystone design goals is to reuse the TEE solutions available on commodity CPUs. For each of these TEEs, 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 [5, 9, 11, 24, 34]. On the other hand, all of these are hardware extensions which are designed and implemented by the CPU manufacturer. Thus, they do not allow users to access the programmable interface at a layer underneath the untrusted OS. One way to simulate the programmable layer is by adding a trusted hypervisor layer which then executes an untrusted OS, but it inflates the TCB. Lastly, none of these potential designs allow for adapting to threat models and workloads.

TEE Programming Support. Previous works add expressiveness to TEE platforms. At the SM layer they optimize program-critical tasks (e.g., context switches, memory operations) [22, 41, 87], at RT they target portability, functionality, and / or security [4, 5, 11, 18, 24, 34, 51, 74, 76, 90, 97], and at eapp layer they reduce the developers efforts for commonly used primitives [20, 21]. Although these systems are a fixed configuration in the TEE design space, they provide valuable lessons for Keystone feature design and future optimization.

Enhancing Security of TEEs. Better and secure TEE design has been a long-standing goal, with advocacy for security-by-design [55, 82]. We point out that Keystone is not vulnerable to a large class of side-channel attacks [30, 98] by design, while speculative execution attacks [29, 62] 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 [26, 60] as we demonstrated in Section 4.3. Lastly, Keystone can benefit

from various RISC-V proposals underway to secure IO operations with PMP [80]. Thus, Keystone either eliminates classes of attacks or allows integration with existing techniques.

8 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 deployments platforms.

Acknowledgments

We thank Srini Devdas and Ilia Lebedev for sharing the Sanctum codebase and initial discussions. We thank Alexander Thomas and Stephan Kaminsky 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 SiFive HiFive Freedom Unleashed and FireSim setup. This material is in part based upon work supported by the National Science Foundation under Grant No. TWC-1518899 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, Huawei, 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.

References

[1] [n. d.]. Chip Designer - SiFive. https://www.sifive.com/chip-designer#fu540.

[2] [n. d.]. HiFive Unleashed - SiFive. https://www.sifive.com/boards/hifive-unleashed.

[3] [n. d.]. Keystone Source Code Release. https://keystone-enclave.org/.

[4] [n. d.]. MesaTEE: a trustworthy memory-safe distributed computing framework. https://mesatee.org/.

[5] [n. d.]. Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/.

[6] [n. d.]. Portable C implementation of Ed25519, a high-speed high-security public-key signature system. https://github.com/mit-sanctum/ed25519.

[7] [n. d.]. QEMU with RISC-V (RV64G, RV32G) Emulation Support. https://github.com/riscv/riscv-qemu.

[8] [n. d.]. RISC-V Proxy Kernel and Boot Loader. https://github.com/riscv/riscv-pk.

[9] [n. d.]. SierraTEE Virtualization for ARM TrustZone and MIPS. https://www.sierraware.com/open-source-ARM-TrustZone.html.

[10] [n. d.]. Standalone C TH library. C/CPU implementation of Tensors. https://github.com/torch/TH.

[11] [n. d.]. T6 - Secure OS and TEE. https://www.trustkernel.com/en/products/.

[12] [n. d.]. Tiny SHA3. https://github.com/mjosaarinen/tiny$_s$ha3/.

[13] [n. d.]. torch/nn gathers nn's C implementations of neural network modules. https://github.com/torch/nn/tree/master/lib/THNN.

[14] AMD. 2016. AMD Memory Encryption Whitepaper v7 Public. https://developer.amd.com/wordpress/media/2013/12/AMD$_M$emory$_E$ncryption$_W$hitepaper$_v$7-Public.pdf.

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

[16] 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.

[17] ARM. 2013. ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper. infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C$_t$rustzone$_s$ecurity$_w$hitepaper.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. UCB/EECS-2016-17 (Apr 2016).

[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] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David M. Eyers, and Peter R. Pietzuch. 2017. TaLoS : Secure and Transparent TLS Termination inside SGX Enclaves.

[22] 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*.

[23] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *CCS*.

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

[25] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2017. Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware. ePrint Archive, Report 2017/1153.

[26] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2018. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. (2018).

[27] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware*.

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

[29] 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*.

[30] 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*.

[31] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. 2012. Fabric: A Retrospective on Evolving SDN. In

HotSDN.

[32] 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.

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

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

[35] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. 2007. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor.

[36] 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*.

[37] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *ArXiv e-prints* (2018). arXiv:1804.05141

[38] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. 2011. SecureME: A Hardware-software Approach to Full System Security. In *ICS*.

[39] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *SOSP*.

[40] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. http://eprint.iacr.org/2016/086.

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

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

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

[44] 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*.

[45] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*.

[46] Dawson R Engler, M Frans Kaashoek, et al. 1995. *Exokernel: An operating system architecture for application-level resource management*. ACM.

[47] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. 2014. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *MICRO*.

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

[49] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A Virtual Machine-based Platform for Trusted Computing. In *SOSP*.

[50] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2003. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*.

[51] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. 2003. Flexible OS Support and Applications for Trusted Computing. In *HOTOS*.

[52] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Costa, Christof Fetzer, Pascal Felber, Peter R. Pietzuch, and Rüdiger Kapitza. 2018. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *DSN*.

[53] 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*.

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

[55] 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/

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

[57] 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*.

[58] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. 2010. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proceedings of International Symposium on Computer Architecture*.

[59] 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-Irving-and-Pierre-Selwan.pdf.

[60] 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*.

[61] 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*.

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

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

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

[65] Ilia A. Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanovic, Dawn Song, and Srinivas Devadas. 2019. Sanctorum: A lightweight security monitor for secure enclaves. *IACR Cryptology ePrint Archive* (2019).

[66] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. *SIGARCH Comput. Archit. News* (2005).

[67] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. 2014. DroidVault: A Trusted Data Vault for Android Devices. In *ICECCS*.

[68] J. Lind, I. Eyal, P. Pietzuch, and E. Gün Sirer. 2016. Teechan: Payment Channels Using Trusted Execution Environments. *ArXiv* (2016). arXiv:1612.07766

[69] 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*.

[70] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for TCB Minimization. *SIGOPS Oper. Syst. Rev.* (2008).

[71] 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*.

[72] 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*.

[73] 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*.

[74] 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.

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

[76] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Security*.

[77] W. Norcott and D. Capps. [n. d.]. IOzone file system benchmark. URL: www.iozone.org.

[78] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*.

[79] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. 2013. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *CCS*.

[80] 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.

[81] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical State Continuity for Protected Modules. In *IEEE S&P*.

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

[83] 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*.

[84] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB - A Secure Database using SGX. In *IEE S&P*.

[85] Rick Boivie. 2012. SecureBlue++: CPU Support for Secure Execution.

[86] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *MICRO*.

[87] 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*.

[88] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *OSDI*.

[89] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud. In *IEEE S&P*.

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

[91] Raoul Strackx, Bart Jacobs, and Frank Piessens. 14. ICE: A Passive, High-speed, State-continuity Scheme. In *ACSAC*.

[92] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security*.

[93] 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).

[94] Jakub Szefer and Ruby B. Lee. 2012. Architectural Support for Hypervisor-secure Virtualization. In *ASPLOS*.

[95] 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*.

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

[97] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE S&P*.

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

[99] Jisoo Yang and Kang G. Shin. 2008. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *VEE*.

[100] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *CCS*.

[101] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *SOSP*.