Abstract

Trusted execution environments (TEEs) are being used in all the 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 application features. 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.

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 create a secure execution environment, commonly referred to as an enclave [24, 79, 96]. On the consumer end, TEEs are now being used for secure cloud services [25, 31], databases [115], big data computations [37, 59, 121], secure banking [91], blockchain consensus protocols [9, 92, 98], smart contracts [32, 49, 141], machine learning [106, 133], network middleboxes [67, 68], and so on. These use-cases have diverse deployment environments ranging from cloud servers, client devices, 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 [96] requires statically sized enclaves, lacks secure I/O and syscall support, and is vulnerable to significant side-channels [53]. Thus, to execute arbitrary applications, the systems built on SGXv1 have inflated the Trusted Computing Base (TCB) and are forced to implement complex workarounds [25, 31, 43, 115, 124]. As only Intel can make changes to the inherent design trade-offs in SGX, users have to wait for changes like dynamic resizing of enclave virtual memory in the proposed SGXv2 [95]. Unsurprisingly, these and other similar restriction have led to a proliferation of TEE re-implementations on other ISAs (e.g., OpenSPARC [42, 89], RISC-V [54, 120]). However, each such redesign requires considerable effort and only serves to create another fixed design point.

In this paper, we advocate that the hardware must provide security primitives and not point-wise solutions. 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 use-cases and research innovation. We believe a similar paradigm shift in TEEs will pave the way for effortless customization. It will allow features and the security model to be tuned for each hardware platform and use-case from a common base. This motivates the need for customizable TEEs to provide a better interface between entities that create the hardware, operate it, and develop applications. Customizable TEEs promise quick prototyping, a shorter turn-around time for fixes, adaptation to threat models, and use-case specific deployment.

The first challenge in realizing this vision is the lack of a programmable trusted layer below the untrusted OS. Hypervisor solutions result in a trusted layer with a mix of security and virtualization responsibilities, unnecessarily complicating the most critical component. Firmware and micro-code are not programmable to a degree that satisfies this requirement. Second, previous attempts at re-using hardware isolation to provide TEE guarantees do not capture the right notion of customization. Specifically, these solutions (e.g., Komodo [61], seL4 [84]) rely on the underlying hardware not only for a separation mechanism but also for pre-made decision about the boundary between what is trusted and untrusted. They use a verified trusted software layer for customization on top of this hardware [61]. To overcome these challenges, our insight is to identify the

Keystone is available at https://keystone-enclave.org/
appropriate hardware memory isolation primitives and then build a modular software-programmable layer with the appropriate privileges.

To this end, we propose Keystone—the first open-source framework for building customizable TEEs. We build Keystone on unmodified RISC-V using its standard specifications [22] 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. More importantly, enclave operates in its own isolated physical memory region and has its own runtime (RT) component which executes in supervisor mode and manages the virtual memory of the enclave. With this novel design, any enclave-specific functionality can be implemented cleanly by its RT while the SM manages hardware-enforced guarantees. This allows the enclaves to specify and include only the necessary components. The RT implements the functionality, communicates with the SM, mediates communication with the host via shared memory, and services enclave user-code 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 via compile-time plugins. 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 plugins 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 a highly configurable cache controller to transparently enable defenses against physical adversaries and cache side-channels.

We build Keystone, the SM, and an RT (called Eyrie) which together allow enclave-bound user applications to selectively use all the above features. We demonstrate an off-the-shelf microkernel (seL4 [84]) as an alternative RT. We extensively benchmark Keystone on 4 suites with varying workloads: RV8, IOZone, CoreMark, and Beeps. 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 test Keystone on different RISC-V systems: the SiFive HiFive Freedom Unleashed, 3 in-order cores and 1 out-of-order core via FPGA, and a QEMU emulation—all without modification. Keystone is open-source and available online [12].

Contributions. We make the following contributions:

- **Need for 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 build and instantiate customizable 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 the advantages of Keystone TEEs which can minimize the TCB, adapt to threat models, use hardware features, handle workloads, and provide rich functionality without any micro-architectural changes. Total TCB of a Keystone enclaved application is between 12-15 K lines of code (LoC), of which the SM consists of only 1.6 KLoC added by Keystone.

- **Benchmarking & Real-world Applications.** We evaluate Keystone on 4 benchmarks: CoreMark, Beeps, 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 enclave defenses for adversaries with physical access via memory encryption and a cache side-channel defense.

2 Customizable TEEs

We motivate the need for customizable TEEs which can adapt to various real deployment scenarios, introduce our key actors, and define our goals.

2.1 Need for a New Paradigm.

Current widely-used TEE systems cater to specific and valuable use-cases but occupy only a small part of the wide design space (see Table 1). Consider the case of heavy server workload (databases, ML inference, etc.) running in an untrusted cloud environment. One option is an SGX-based solution which has a large software stack [25, 31, 43] to extend the supported features. On the other hand, a SEV-based solution requires a complete virtualization stack. 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 applications within one domain with software-based Secure World OS solutions [7]. Thus, irrespective of the TEE, developers often compromise their design requirements (e.g., resort to a large TCB solution, limited privilege domains) or take on the onus of building their custom design.

**Hardware-software co-design.** Keystone builds on simple yet elegant primitives. We leverage multiple well-defined and existing hardware features while allowing the software layers to manage the complex responsibilities. Previous works explore the spectrum of tasks the hardware and software should do. For instance, seL4 [84] uses a single isolation domain exposed by the hardware and performs almost all resource management, isolation, and security enforcement in a verified software layer. Komodo uses the two isolation domains provided by ARM TrustZone to execute multiple isolated domains enforced by formally verified software. In Keystone, we expect arbitrary hardware-isolated domains whose boundaries can be dynamically configured in the software. With this novel choice, Keystone provides guarantees with minimum trusted software while allowing the isolated regions to manage themselves.

**Customizable TEEs.** We define a new paradigm—customizable TEEs—to allow multiple stakeholders to customize a TEE. The hardware manufacturer only provides 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 which plugs in required components and composes them to instantiate the required TEE. With this, we bridge the existing gaps in the TEE design space, allow researchers to independently explore design trade-offs without significant development effort, and encourage rapid response to vulnerabilities and new feature requirements.

**Trusted Hardware Requirements.** Keystone requires no changes to CPU cores, memory controllers, etc. A secure hardware platform supporting Keystone has several feature requirements: a device-specific secret key visible only to the trusted boot process, a hardware source of randomness, and a trusted boot process. Key provisioning [20] is an orthogonal problem. For this paper, we assume a simple manufacturer provisioned key.

### 2.2 Entities in TEE Lifecycle
We define six logical entities in customizable TEEs:

- **Keystone hardware designer** designs the hardware; designs or modifies existing SoCs, IP blocks and interactions.
- **Keystone hardware manufacturer** fabricates the hardware; assembles them as per the pre-defined design.
- **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 programmers who develop these specific components as SM programmer, RT programmer, eapp programmer respectively.
- **Keystone user** chooses a Keystone configuration of RT and an eapp. They instantiate a TEE which can execute on their choice of hardware provisioned by the Keystone platform provider.
- **eapp user** interacts with the eapp executing in an enclave on the TEE instantiated using Keystone.

We define these fine-granularity roles for customization options in Keystone. In real-world deployments, a single entity can perform multiple roles. For example, consider the scenario where 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 Keystone hardware designer and Keystone 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 use RISC-V to design and build Keystone. RISC-V is an open ISA with multiple open-source core implementations [26, 41]. It 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 access physical resources (e.g., interrupts, memory, devices).

#### 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 security monitor (SM) to enforce TEE guarantees on the platform using three properties of M-mode: (a) it is programmable by platform providers and meets our needs for a minimal highest privilege mode. (b) It controls hardware delegation of the interrupts and exceptions in the system. All the lower privilege modes can receive exceptions or use CPU cycles only when the M-mode allows it. (c) Physical memory protection (PMP),
Table 1. Trade-offs in existing TEEs and extensions. C1-2: Category and specific systems. ●, ○, □ represent best to least adherence to the property in the column respectively. C3-6: resilience against software adversary, hardware adversary, side-channel adversary, control-channel adversary respectively; the symbols indicates complete protection, only confidentiality protection, and no protection respectively. C7: zero, thousands of LoC, millions of LoC of TCB. C8: zero or non-zero hardware / micro-architectural modifications. C9: allows the enclave to do self resource management with complete, partial, or no flexibility. C10: the range of applications supported are maximum; specific class; only written from scratch. C11: expressiveness includes forking, multi-threading, syscalls, shared memory; partial; none of these. C12: developer effort for porting is zero (unmodified binaries); partial (compiling and / or configuration files); substantial (significant re-writing).

<table>
<thead>
<tr>
<th>System</th>
<th>Software Adversary Protection</th>
<th>Physical Adversary Protection</th>
<th>Side Ch Adversary Protection</th>
<th>Control Ch Adversary Protection</th>
<th>Low SW TCB</th>
<th>No HW Mods</th>
<th>Flexible Resource Mgmt</th>
<th>Range of Apps Supported</th>
<th>Supports High Expr</th>
<th>Low Porting Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGX [96]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Haven [31]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Graphene-SGX [43]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Scone [25]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Varys [107]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>TrustZone [24]</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Komodo [61]</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>OP-TEE [7]</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Sanctuary [36]</td>
<td>○</td>
<td>■</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>SEV [79]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>SEV-ES [78]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Sanctum [54]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>TIMBER-V [120]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>MultiZone [5]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Keystone (This paper)</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

3.2 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 RT trusts the SM, and the eapp trusts the SM and the RT.

4.1 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 to selectively include TCB only when required. Keystone can instantiate TEEs with minimal TCB for each of the specific expected 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.

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.

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 lowers its TCB and allows it to present clean abstractions. Our S-mode runtime (RT) and U-mode enclave application (eapp) are mutually trusting, 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 2) 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 enclave.

Adaptive Threat Model. Keystone can operate under diverse threat models, each requiring different defense mechanisms. For this reason, we outline all the 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 own 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 the enclave code and data at all points during...
execution. 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. $A_{phy}$ is for confidentiality, $A_{phy}$ 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 passively observing interactions between the trusted and the untrusted components via the cache side-channel ($A_{cache}$), the timing side-channel ($A_{time}$) or the control channel ($A_{cntn}$). A **denial-of-service attacker** $A_{des}$ can take down the enclave or the host OS. Keystone allows these attacks against 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 [39, 69, 85, 99, 122, 134]. We recommend Keystone users to avoid deployments on out-of-order cores. Existing and future defenses against this class of attacks can be retrofitted into Keystone [35, 138]. Keystone does not natively protect the enclave or the SM against timing-side-channel attacks. SM, RT, and eapp programmers should use existing software solutions to mask timing channels [66] and Keystone hardware manufacturers can supply timing-side-channel-resistant hardware [83]. The SM exposes a limited API to the host OS and the enclave. We do not provide non-interference guarantees for this API [61, 126]. 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 [43, 109, 125], and the SM, RT, and eapp are bug-free.

### 4 Keystone Internals

We discuss the Keystone design details and the enclave lifecycle. Our enclave consists of a kernel-like component in the S-mode (the runtime—RT), and an application in the U-mode (the enclave application—eapp). The RT and the host OS share a dedicated buffer that is accessible only to specific RT functions and the host.

#### 4.1 Keystone Memory Isolation Primitives

Keystone has significantly better modularity and customizability because of our conscious design choices. First, we expect the hardware to provide only simple security primitives and interfaces. Second, we assign minimum responsibilities of the trusted software components executing at the highest privilege (e.g., bootloader, SM). Finally, we push bulk of the resource management logic either to the untrusted software or the enclave.

<table>
<thead>
<tr>
<th>Caller</th>
<th>SM API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>create</td>
<td>Validate, and measure the enclave</td>
</tr>
<tr>
<td></td>
<td>run</td>
<td>Start enclave and boot RT</td>
</tr>
<tr>
<td></td>
<td>resume</td>
<td>Resume enclave execution</td>
</tr>
<tr>
<td></td>
<td>destroy</td>
<td>Release enclave memory</td>
</tr>
<tr>
<td>RT</td>
<td>stop</td>
<td>Pause enclave execution</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td>Terminate the enclave</td>
</tr>
<tr>
<td></td>
<td>attest</td>
<td>Get a signed attestation report</td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>Get secure random values</td>
</tr>
<tr>
<td>OS &amp; RT</td>
<td>plugin*</td>
<td>Call SM plugins (e.g., dynamic resizing)</td>
</tr>
</tbody>
</table>

**Table 2.** The SBI API functions (SBI) provided by the Keystone SM. * indicates that the SBI is provided by an optional SM plugin.

**Background: RISC-V Physical Memory Protection.** Keystone uses physical memory protection (PMP) 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 2). Each PMP entry controls the U-mode and S-mode permissions to a customizable region of physical memory.

PMP makes Keystone memory isolation enforcement flexible for three reasons: (a) multiple contiguous enclave memory regions can coexist instead of saving one large memory region shared by all enclaves; (b) PMP entries can cover a region between sizes 4 bytes or maximum DRAM size so Keystone enclaves utilize page-aligned memory with an arbitrary size; (c) PMP entries can be dynamically reconfigured during execution such that Keystone can dynamically create a new region 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). The SM disallows all access to its memory from 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 any 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 SM.** PMP makes Keystone memory isolation enforcement flexible for three reasons: (a) multiple contiguous enclave memory regions can coexist instead of reserving one large memory region shared by all enclaves; (b) PMP entries can cover a region between sizes 4 bytes or maximum DRAM size so Keystone enclaves utilize page-aligned memory with an arbitrary size; (c) PMP entries can be dynamically reconfigured during execution such that Keystone can dynamically create a new region 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). The SM disallows all access to its memory from U-mode and S-mode using this first entry and configures the last PMP entry (lowest priority) to cover all memory and with all permissions enabled. The OS thus has default full permissions to all memory regions 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

---

1The pmaddr and pmcfg CSR registers have up to 16 M-mode configurable PMP entries.
2Currently processors have up to 16 M-mode configurable PMP entries.
region. On receiving a valid enclave creation 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 2), 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.

At a CPU 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, Keystone adds a PMP entry to disallow everyone from accessing the enclave. These changes during the creation 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. When Keystone destroys or creates an enclave, all the other cores are notified to update their PMP entries. There are no other times when the PMPs need to be synchronized via IPIs.

4.2 Enclave Lifecycle

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

**Enclave Creation.** 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. Then the SM hashes the page contents along with the virtual addresses and the configuration data.

**Enclave Execution.** The SM sets the PMP entries and transfers the control to the enclave entry point.

**Enclave Destruction.** On an OS initiated tear-down, 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.

4.3 Post-creation In-enclave Page Management

Keystone has a different memory management design from the existing TEEs (see Figure 4). It uses the OS generated...
enclaves may occupy a fixed contiguous physical memory allocated by the OS, with a statically-mapped virtual address space at load time. Although this is suitable for some embedded applications, it limits the memory usage of most legacy applications. To this end, we describe several optional plugins which enable flexible memory management of the enclave.

**Free-memory.** We built a plugin that allows the Eyrie RT to perform page table changes. It also lets the enclave reserve physical memory without mapping it at creation time. This unmapped (hence, free) memory region is not included in the enclave measurement and is zeroed before beginning the eapp execution. The free-memory plugin is required for other more complex memory plugins.

**Dynamic Resizing.** Statically pre-defined maximum enclave size and subsequent static physical / virtual memory pre-allocations prevent the enclave from scaling dynamically based on workload. It also complicates the process of porting existing applications to eapps. To this end, Keystone allows the Eyrie RT to request dynamic changes to 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 such an allocation, the SM increases the size of the enclave and notifies the Eyrie RT who then uses the free memory plugin to manage the new physical pages.

**In-Enclave Self Paging.** We implemented a generic in-enclave page swapping plugin 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 plugin uses a simple random eapp-only page eviction policy. It works in conjunction with the free memory plugin 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 [107–109].

**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 6(d)) or by a dedicated trusted hardware unit—a memory encryption engine (MEE) [72]—with an SM plugin (Figure 6e). Admittedly, this incurs significant design challenges in efficiently storing the metadata and performance optimizations. Keystone design is agnostic to the specific integrity schemes and can reuse the existing mechanisms [97, 118].
The adversary cannot insert cache lines in this partition. (a) the L2 cache controller’s way masking primitive similar to Intel’s CAT [104]; (b) PMP to way-partition the L2 cache memory are in the partition and are thus protected by PMP. 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. The net effect is that the 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.

5.3 Functionality Plugins

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 plugin can be used to add support for syscalls, IPC, enclave-enclave communication, and so on.

Proxyed System Calls. We allow the proxying of some 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 lago attacks [44] via this interface [43, 109, 125].

In-enclave System Calls. For appropriate syscalls (e.g., mmap, brk, getrandom), Keystone invokes an SM interface or executes them in Eyrie to return the results to the eapp.

Multi-threading. We run multi-threaded eapps by scheduling all the threads on the same core. We do not support parallel multi-core enclave execution yet.

5.4 Other Keystone Primitives

Keystone supports following other standard TEE 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 are standard operations, can be implemented in numerous ways [82, 87]. Keystone does not rely on a specific implementation. For
We argue the security of the enclave, the OS, and the SM via the SM.

**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 [100] if applicable.

**Trusted Timer.** The SM naturally distrusts all the lower-privilege software components (eapps, RTs, host OS, etc.). It is protected from re-use the existing shielding systems [25, 43, 125] as plugins to defend the enclave against these attacks.

**Remote Attestation.** 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 [61, 87] by including limited arbitrary data (e.g., Diffie-Hellman key parameters) in the signed attestation report. Key distribution [20], revocation [75], attestation services [77], and anonymous attestation [38] are orthogonal challenges.

**Secure Delegation.** Keystone delegates the traps (i.e., interrupts and exceptions) raised during the enclave execution to the RT via the interrupt delegation registers. The RT invokes the appropriate handlers inside the enclave for user-defined exceptions and may forward other traps to the untrusted OS via the SM.

**Monotonic Counters & Sealed Storage.** Enclaves may need monotonic counters for protection against rollback attacks and versioning [53]. Keystone can support monotonic counters by keeping a limited counter state in the SM memory. Keystone can support sealed storage [20] in the future.

**6 Security Analysis**

We argue the security of the enclave, the OS, and the SM based on the threat model outlined in Section 3.2.

**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 the enclave data structures can only be updated in the enclave or the SM, both of them are isolated from direct access. Subtle attacks such as controlled side channels (\( A_{Cntlr} \)) are not possible in Keystone because 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, does not intentionally create malicious virtual to physical address mappings [74] 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 plugin, 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 [44, 114]. Keystone can re-use the existing shielding systems [25, 43, 125] as plugins to defend the enclave against these attacks.

**Side-channel Attacks.** Keystone thwarts cache side-channel attacks (Section 5.2). 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 [127, 128]. 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 \( A_{SW} \) is 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 SM watchdog timer assures that the enclave returns 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 [61, 102]. The SM is only a reference monitor, it does not require scheduled execution time, so an \( A_{Pos} \) is not a concern. The SM can defend against an \( A_{Cache} \) and an \( A_{Time} \) with known techniques [66, 83].
6.4 Protection Against Physical Attackers

Keystone can protect against a physical adversary by a combination of plugins and a proposed modification to the bootloader. Similar to [47], we use a scratchpad to store the decrypted code and data, while the supervisor mode component (Eyrie plugins) handles the page-in and page-out.

The enclave itself is protected by the on-chip memory plugin (in SM) and the paging plugins (in Eyrie), 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 complete DRAM control.

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-C000, this would involve repurposing a portion of the L2 loosely-integrated memory (LIM) via a modified trusted bootloader.

With all of these plugins and techniques in place, the only content present 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 hardware modifications.

7 Implementation

We implemented our SM on top of the Berkeley Boot Loader (bbl) [10]. It supports M-mode trap handling, boot, and other features. We simulated the primitives that require hardware support (randomness, root-of-trust, etc.) and provided well-defined interfaces. All the plugins in Section 5 are available as compile-time options. Memory encryption is implemented using software AES-128 [1] and integrity protection is partially implemented. We implemented the Eyrie RT from scratch in C. We ported the sel4 microkernel [84] to Keystone with 290 LoC modifications to sel4 for boot, memory initialization, and interrupt handling. There are no inherent restrictions to these two RTs, and we expect future development to add further options. Our user-land interface for interactions with the enclaves is provided via a Linux kernel module that creates a device endpoint (/dev/Keystone).

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, RT into a single binary. We have a complete top-level build solution to generate a bootable Linux image (based on the tooling for the SiFive HiFive Freedom Unleashed) for QEMU, FPGA softcores, and the HiFive containing our SM, the driver, and the enclave binaries. Keystone is open-source and available at https://github.com/keystone-enclave/.

Writing eapps. The eapp developers can choose to port their legacy applications along with one of the default RTs provided by Keystone. This requires near-zero developer efforts. Alternatively, the developers can use our SDK to write eapp specifically for Keystone. Lastly, Keystone supports manually partitioning the applications such that the security-sensitive parts of the logic execute inside the enclave, while the rest of the code can execute in the untrusted host. Thus, Keystone framework gives the developers ample flexibility in designing and implementing eapps.

8 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 Keystone-instantiated TEE in various deployment modes?

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

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

Experimental Setup. We used four different setups for our experiments; the SiFive HiFive Freedom Unleashed [3] with a closed-source FU540-C000 (at 1GHz), and three open-source RISC-V processors: small Rocket (Rocket-S), default Rocket [26], and Berkeley Out-of-order Machine (BOOM) [41]. See Table 3 for details. We instantiated the open-source processors on FPGAs using FireSim [80] which is responsible for emulating the cores at 1GHz. The host OS is buildroot Linux (kernel 4.15). All evaluation was performed on the HiFive and the data is averaged over 10 runs unless otherwise specified.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Core</th>
<th>Cache Size (KB)</th>
<th>Latency (cycles)</th>
<th># of TLB Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket-S</td>
<td>1 in-order</td>
<td>8/8</td>
<td>512</td>
<td>24</td>
</tr>
<tr>
<td>Rocket</td>
<td>1 in-order</td>
<td>16/16</td>
<td>512</td>
<td>24</td>
</tr>
<tr>
<td>BOOM</td>
<td>1 OoO</td>
<td>32/32</td>
<td>2048</td>
<td>24</td>
</tr>
<tr>
<td>FU540</td>
<td>4 in-order</td>
<td>32/32</td>
<td>2048</td>
<td>12-15*</td>
</tr>
</tbody>
</table>

Table 3. Hardware specification for each platform. L2 cache latency in FU540 (*) is based on estimation.

8.1 Modularity & Support

We outline the qualitative measurement of Keystone flexibility in extending features, reducing TCB, and using the platform features. Table 4 shows the TCB breakdown of various components (required and optional) for the SM and Eyrie RT.
Keystone

<table>
<thead>
<tr>
<th>Component</th>
<th>Runtime</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>1800</td>
<td>1100</td>
</tr>
<tr>
<td>Memory Isolation</td>
<td>—</td>
<td>500</td>
</tr>
<tr>
<td>Free Memory</td>
<td>300</td>
<td>—</td>
</tr>
<tr>
<td>Dynamic Memory</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>Edge-call Handling</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>Syscalls</td>
<td>450</td>
<td>—</td>
</tr>
<tr>
<td>11bc Environment</td>
<td>50</td>
<td>—</td>
</tr>
<tr>
<td>IO Syscall Proxying</td>
<td>300</td>
<td>—</td>
</tr>
<tr>
<td>Cache Partitioning</td>
<td>—</td>
<td>300</td>
</tr>
<tr>
<td>In-enclave Paging</td>
<td>300</td>
<td>—</td>
</tr>
<tr>
<td>On-chip Memory</td>
<td>—</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 4. TCB Breakdowns (in LoC) for Eyrie RT and SM features.

**Extending RTs.** 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 8.3). Keystone passes all the tests in sel4 suite (4K-11K cycles/page).

**Extending the SM.** The advantage of an easily modifiable SM layer is noticeable when the 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 (bb1 and SM), the RT, the untrusted host application, the eapp, and the helper libraries, of which only one is in the TCB. The M-mode component is 10.7 KLoC with a cryptographic library (4 KLoC), pre-existing trap handling, boot, and utilities (4.7 KLoC), the SM (1.6 KLoC), and SM plugins (400 LoC). A minimum Eyrie RT is 1.8 KLoC, with plugins adding further code as shown in Table 4 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 [2] and undef [17].

**8.2 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 plugin 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.

**Common Operations.** Figure 7 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 [15]. The remaining enclave creation time totals 20k-30k cycles. Similarly, the attestation is dominated by the ed25519 [8] signing software implementation with 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 take longer on the FU540 (4-core), which can be attributed to the multi-core PMP synchronization.

**Standard Benchmarks Used as Unmodified eapp Binaries.** Beebs, CoreMark, and RV8. As expected, Keystone incurs no meaningful overheads (+0.7%, excluding enclave creation/destruction) for pure CPU and memory benchmarks.
Figure 9. Total execution time comparison for the RV8 benchmarks. Each bar consists of the duration of the application (user or eapp), and the other overheads (other). Keystone (Keystone) and Keystone with cache partitioning plugin (Keystone-cache) are compared with the native execution in Linux (base).

Table 5. Overhead of various plugins on RV8 benchmarks over native execution. ⊗: no plugin, 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.

<table>
<thead>
<tr>
<th>Plugins</th>
<th>Overhead (%)</th>
<th># of Page Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O, P</td>
<td>O, P</td>
</tr>
<tr>
<td>primes</td>
<td>-0.9</td>
<td>40.5</td>
</tr>
<tr>
<td>miniz</td>
<td>0.1</td>
<td>128.5</td>
</tr>
<tr>
<td>aes</td>
<td>-1.1</td>
<td>66.3</td>
</tr>
<tr>
<td>bigint</td>
<td>-0.1</td>
<td>1.6</td>
</tr>
<tr>
<td>qsort</td>
<td>-2.8</td>
<td>-1.3</td>
</tr>
<tr>
<td>sha512</td>
<td>-0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>norx</td>
<td>0.1</td>
<td>0.9</td>
</tr>
<tr>
<td>dhrystone</td>
<td>-0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

8.3 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) a cryptographic library porting effort 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.

Porting Efforts & Setup. 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 in 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 [58] (see Table 6). They comprise 15.7 and 15.4KLoC of TH [13] and THNN [16] libraries from Torch compiled with musl libc. Each model has an additional 230 to 13.4 KLoC of model-specific inference code [133]. We performed two sets of experiments: (a) execute the model inference code with static maximum enclave size; (b) turn on the dynamic resizing plugin so that the enclave extends its size on-demand when it executes. Figure 10 shows the performance overheads for these two configurations and non-enrolled 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. We report an overhead between ~3.12%(LeNet) to 7.35%(Densenet) for all the models with both static enclave size and dynamic resizing. The cause for this is: (a) Keystone loads the entire binary in physical memory before it being eapp execution, precluding any page faults for zero-fill-on-demand or similar behavior, so smaller
We used a small hand-coded test to verify that Eyrie RT’s workload characteristics, binary object size, and the total enclave memory usage. The model specification, shown in Table 6.

<table>
<thead>
<tr>
<th>Model</th>
<th># of Layers</th>
<th># of Param</th>
<th>App LOC</th>
<th>Binary Size</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wideresnet</td>
<td>93</td>
<td>36.5M</td>
<td>1625</td>
<td>140MB</td>
<td>384MB</td>
</tr>
<tr>
<td>Resnext29</td>
<td>102</td>
<td>34.5M</td>
<td>1910</td>
<td>123MB</td>
<td>394MB</td>
</tr>
<tr>
<td>Inceptionv3</td>
<td>313</td>
<td>27.2M</td>
<td>5339</td>
<td>92MB</td>
<td>475MB</td>
</tr>
<tr>
<td>Resnet50</td>
<td>176</td>
<td>25.6M</td>
<td>3094</td>
<td>98MB</td>
<td>424MB</td>
</tr>
<tr>
<td>Densenet</td>
<td>910</td>
<td>8.1M</td>
<td>13399</td>
<td>32MB</td>
<td>570MB</td>
</tr>
<tr>
<td>VGG19</td>
<td>55</td>
<td>20.0M</td>
<td>1088</td>
<td>77MB</td>
<td>165MB</td>
</tr>
<tr>
<td>Resnet110</td>
<td>552</td>
<td>1.7M</td>
<td>9528</td>
<td>7MB</td>
<td>87MB</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>65</td>
<td>1.2M</td>
<td>914</td>
<td>5MB</td>
<td>52MB</td>
</tr>
<tr>
<td>LeNet</td>
<td>12</td>
<td>62K</td>
<td>230</td>
<td>0.4MB</td>
<td>2MB</td>
</tr>
</tbody>
</table>

Table 6. Summary of the Torch Models. The model specification, workload characteristics, binary object size, and the total enclave memory usage.

![Figure 10](image-url) Comparison of inference timing for various Torch models with and without dynamic resizing. Each bar consists of the duration of the application (user or eapp), and the other overheads (other). Keystone (Keystone) and Keystone with the dynamic resizing plugin (Keystone-dyn) are compared with the native execution in Linux (base).

Keystone is the first framework for customizing TEEs. Here, we survey existing TEEs and alternative approaches.

### TEE Architectures & Extensions
We summarize the design choices of the most recent TEEs and their extensions in Table 1. Of these, three major 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 [96]; (b) ARM TrustZone divides the memory into two worlds (i.e., normal vs. secure) to run applications in protected memory [24]; and (c) Sanctum uses the memory management unit (MMU) and cache partitioning to isolate memory and prevent cache side-channel attacks [54].

Several other TEEs explore design layers such as hypervisors [30, 45, 48, 52, 63, 74, 81, 89, 93, 131, 140, 142], physical memory [35, 42, 86, 88, 94, 110, 119], virtual memory [36, 50, 55, 60, 120], and process isolation [56, 64, 113, 117, 130, 132].

### 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 [7, 11, 14, 31, 43]. On the other hand, all of the existing TEEs 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 this approach inflates the TCB. Lastly, none of these potential designs allow for adapting to threat models and workloads.

### 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 [21, 70].

**TEE Support.** Several existing works enhance existing TEE platforms. At the SM layer they optimize program-critical tasks [29, 54, 120], at RT they target portability, functionality, security [4, 6, 7, 14, 25, 31, 43, 65, 101, 105, 124, 135], and at eapp layer they reduce the developer efforts [27, 28]. Although these systems are a fixed configuration in the TEE design space, they provide valuable lessons for Keystone feature design and future optimization.

**Enhancing the Security of TEEs.** Better and secure TEE design has been a long-standing goal, with advocacy for security-by-design [76, 112]. We point out that Keystone is not vulnerable to a large class of side-channel attacks [40, 123, 137] by design, while speculative execution attacks [39, 85] 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 [35, 83] as we demonstrated in Section 5.2. Lastly, Keystone can benefit from various RISC-V proposals underway to secure IO operations with PMP [111]. 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 [18, 19, 23, 33, 34, 46, 51, 57, 62, 71, 73, 84, 90, 103, 116, 136, 139] may provide stronger guarantees, and Keystone can help to explore designs which combine these with hardware protection [61, 129].

10 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. Keystone can serve as a framework for research and prototyping better TEE designs.

Acknowledgments

We thank Srinivas Devdas and Ilia Lebedev for sharing the Sanc- tum 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. 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 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

[9] Proof of elapsed time (poet) 1.0 specification &ÅÁ sawtooth v1.0.5 documentation. https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html.


