A Review of the Keystone Trusted Execution Framework

Markus Switalla
Friedrich-Alexander-Universität Erlangen-Nürnberg
Germany

ABSTRACT

For isolating security-sensitive code from an untrusted environment, it has become common to make use of trusted execution environments (TEEs). Software running inside such an environment is isolated from the potentially compromised host system, making it possible to shield security-sensitive applications from malicious underlying software. Keystone is a software-based TEE, designed to be flexible and highly customisable, unlike hardware-based solutions that cannot be changed once implemented. This review will explain the way Keystone functions and what features it can offer to its users and TEE developers.

1 INTRODUCTION

The tremendous code sizes of the common general-purpose operating systems can pose a great security risk when executing highly security-sensitive programs. Ruling out security flaws inside an operating system becomes increasingly difficult the more complex the system is. So-called trusted execution environments (TEEs) have been introduced to address the issue of having to trust complex host systems when running security-sensitive code. Utilising a TEE, critical code can be executed in isolation of the host, making attacks through the host system unfeasible, even if it has been compromised in an arbitrary way. Keystone [11] is one such TEE implementation, suited for creating so-called enclaves on RISC-V based platforms that sensitive software can be executed in. In contrast to common hardware-based TEE implementations, Keystone is designed to be freely customisable and not be tied to constraints imposed by the hardware designer. Keystone aims on full flexibility on the enclave developer's side, making it possible for the developer to thoroughly fit the TEE to a certain use case. As a result, Keystone should be regarded as a TEE architecting framework rather than a fixed TEE implementation.

Keystone's source code is publicly available [5], providing developers with a base TEE implementation, yet allowing them to customise it in any way possible. As a software-based solution, Keystone does not require any changes made to the underlying hardware, given that the minimum hardware requirements to employ Keystone are satisfied.

This review aims on giving insights into Keystone's inner workings, its applicability and its flexibility. Keystone's hardware requirements, the way Keystone works, what features it offers, how it is used and how it can be extended are all covered in detail throughout this work.

2 BACKGROUND

In order to understand how Keystone works, basic knowledge of the RISC-V instruction set architecture (ISA) and the necessary RISC-V Standard Extensions is required. To provide the necessary background information, this section will explain some RISC-V fundamentals.

2.1 The RISC-V Instruction Set Architecture

RISC-V is a modular ISA, consisting of an integer base and various standard extensions [14]. Per the RISC-V specification, the integer ISA has to be implemented, as it comprises the basic instructions and registers needed to perform even primitive computing operations in the first place. However, standard extensions like the ones needed for multiplication and division or floating-point operations need not necessarily be implemented to achieve RISC-V conformity. A hardware manufacturer may thus choose to only implement the base integer ISA and nothing else, or to include certain standard extensions, or to even implement own custom extensions that are not covered by any RISC-V specification. This leads to a very heterogenous hardware supply, with suitable hardware implementations being available for each use case. Keystone as one such use case also requires a certain set of implemented standard and custom extensions, as outlined in the following.

2.2 Privilege Levels

In order to provide for isolation between an operating system (OS) and user applications, RISC-V defines a privilege level scheme consisting of three different privilege modes [15]. Each privilege mode is able to access different control and status registers (CSRs) in order to modify the state of the machine. CSRs are used among others to perform actions like servicing interrupt requests, managing virtual memory or restricting physical memory access.

The *M-mode* (machine mode) is the most privileged mode, able to access all CSRs. It is used for execution of bootloaders and security-sensitive interrupt handling that requires full permissions.

The *S-mode* (supervisor mode) is less privileged, yet able to access and modify a fair amount of supervisor CSRs. Software running in S-mode is able to perform virtual memory management and to service interrupt requests that are delegated from M-mode. Therefore, the S-mode is highly suitable for kernel execution.

The least-privileged *U-mode* (user mode) is used for user applications that are supposed to run in isolated address spaces and be scheduled preemptively by a supervisor. In scenarios where there is no supervisor, the U-mode can be used to restrict applications from modifying or observing certain system state.

Not all modes need to be implemented by a hardware provider. The only mandatory privilege level is the M-mode (implying no privilege separation at all), while U-mode and S-mode are optional. However, Keystone requires that all three modes be implemented.

2.3 Binary Interfaces

Ordinarily, applications can interact with the OS via a system call interface. The user application can invoke a system call that is serviced by the OS. In terms of RISC-V privilege levels, system calls act as an interface between S-mode and U-mode. Similarly, there exists another such interface between S-mode and M-mode, called the supervisor binary interface (SBI). Software running in S-mode

can call M-mode routines via the SBI in the same way a U-mode application can issue a system call to the OS. RISC-V specifies a standard SBI [9], implemented e.g. by the OpenSBI software [7].

2.4 Physical Memory Protection

RISC-V physical memory protection (PMP) is an optional (but required by Keystone) standard extension that enables M-mode software to restrict S-mode and U-mode memory access. In M-mode, specific PMP CSRs can be written to in order to designate physical memory regions that cannot be accessed by the less-privileged modes. These CSRs have different priorities, such that a high-priority PMP entry can specify that a region must not be accessible, even if said region is contained within a memory region whose access is allowed by a lower-priority PMP entry. Any region not covered by any PMP entry is unaccessible by S-mode and U-mode.

2.5 Virtual Address Spaces

In addition to the constraints imposed by PMP on S-mode and U-mode, RISC-V offers a standard extension to support virtual address spaces with paging [15]. Page tables are managed by S-mode software in order to spatially isolate user applications from each other and the rest of the system. Although M-mode software cannot use virtual addresses (memory is always addressed by physical addresses in M-mode), it can freely inspect and modify any paging structure and even swap root page tables arbitrarily. Providing means to enable paging is optional for a hardware platform, and although Keystone does not conceptually rely on it, the reference implementation [5] expects paging to be enabled.

2.6 Root of Trust

Basic TEE functionalities include attestation and data sealing. Attestation refers to the proof towards a possibly remote verifier that the system is running in a defined state, i.e. that the enclave is properly loaded and has not been tampered with. Data sealing describes a feature that enables software in an enclave to persistently store data without the possibility of any software outside the enclave to observe or modify its contents.

To make these features available, the lowest-level M-mode Keystone module (described in Section 3.2) must be hashed and signed by tamper-proof software or hardware before being executed. Such a mechanism is referred to as the *root of trust (RoT)* and is among the first things to be done after system reset. An RoT is commonly needed with TEEs to be able to establish a chain of trust. As all code that an enclave depends on must be trusted, there is a need for some inherently trusted component, i.e. a *root* of all trust.

Keystone itself provides no such means, as it is the hardware manufacturer's responsibility to provide an RoT to the system. Additionally, the RoT must provide Keystone with an asymmetric key pair, signed by the RoT itself. The RoT can be implemented as an unmodifiable zeroth-order bootloader that has exclusive access to a hardware provided private key.

As there is no RISC-V specification addressing RoTs, a custom extension specified by the platform provider is necessary.

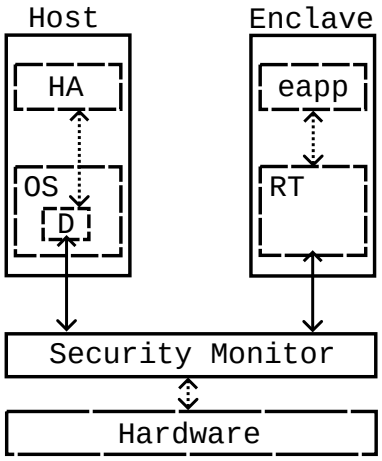

Figure 1: The architecture of a Keystone system. As indicated by the solid arrows, all communication between the host and the enclave is managed by the security monitor. Components shown in the figure include the runtime (RT), eapp (application running inside the enclave), host application (HA) and host driver (D).

3 KEYSTONE FUNCTIONALITY

In this section, the structure of Keystone is discussed, along with how Keystone operates, what it is capable of and what adversaries it offers protection against.

3.1 Adversary Model

Through the use of PMP, Keystone can protect an enclave from an attacker capable of arbitrarily controlling the host OS, host applications and other enclaves. Additional custom hardware extensions offer further protection against a physical attacker able to read and modify DRAM contents and against shared-cache side-channel attacks. Denial of service carried out by the host (i.e. the preemptive destruction or ceased resumption of the enclave) cannot be protected against with Keystone. A Keystone enclave can also utilise host system calls, as outlined in Section 3.9, and may therefore be susceptible to attacks making use of the system call interface. Lee et al. mention in [11] that such attacks can be countered by known defenses. As the host controls an enclave's life cycle and is well able to measure time spent with enclave execution, it may be able to carry out timing attacks. An analysis of such attacks and possible countermeasures is not straightforward and therefore out of this work's scope.

3.2 Architecture

Keystone consists of three main modules that constantly work together to allow enclave operability. While there is a publicly available implementation of each module [4, 5], Keystone allows for great customisation to the extent of even creating completely custom modules. The modules' source codes can easily be modified to better suit a Keystone user's requirements and the hardware

that a Keystone instance should be employed on. In the following, all three modules are described in detail. An overview of the components and their interactions is shown in Figure 1.

The *security monitor (SM)* is fundamentally important for how Keystone works, as it is the only component operating in M-mode. It is responsible for ensuring that neither the OS nor the enclave can observe or manipulate each other. The SM mainly functions by restricting physical memory access to either the host or the enclave, whichever is running at a given moment. Additionally, the SM extends the system's SBI to offer enclave management services to both the host and the enclave, as shown in Section 3.4.

The *runtime* is responsible for servicing applications inside the enclave (eapps). It acts as a supervisor to these applications, offering system calls, memory management and SM communication via the SM's SBI. The reference runtime implementation is called *Eyrie* [4] and offers basic kernel functionality. Modifying Eyrie or coming up with other runtime implementations may be useful to better tailor the runtime to an eapp's requirements, as shown in Section 4.1.

The *host driver* enables the host application to create an enclave and have it executed. While there is no reason why Keystone functionality could not be an integral part of the host OS (given that one could develop a supervisor specially designed for Keystone), current kernels do not offer such features and have to be retrofitted to support Keystone. There is a reference implementation of a Linux driver, supplementing Linux with Keystone capabilities. The driver will setup, launch and destroy enclaves at request. In order to achieve this, it has to communicate with the SM via SBI calls.

3.3 Startup

On system startup, the SM's binary has to be loaded and subsequently measured by the RoT. More precisely, all memory that the trusted M-mode software comprises has to be measured. That of course also includes other software like the SBI implementation and the bootloader, as these programs can freely modify any memory addresses due to their privilege level. For the sake of simplicity, the hash generated by the RoT in this regard is referred to as the *SM hash*.

The RoT generates an asymmetric keypair needed for attestation, using the device's private key (requires platform support) and the SM hash as inputs to the key generator. As discussed in Section 3.8, the generated SM keypair must be deterministic but unforgeable. The RoT signs both the SM hash and the SM public key and stores the hash along with the keypair and the signature in a location accessible by the SM. Then, any device setup can be performed by a bootloader. Before handing control over to the firmware payload (an S-mode second-stage bootloader or the host system), the SM must be executed initially for it to initialise. During initialisation of the SM, a PMP region is defined via the highest-priority PMP CSR to cover the SM's memory and restrict any S-mode access to it. In addition, another PMP region (lowest priority) is setup to cover the rest of the physical address space, permitting S-mode and U-mode access (this is mandatory for the host to function). It is unimportant whether the bootloader initiates the SM startup before, during or after its own operations, as long as it does not modify the PMP entries or any SM memory. At any time after SM initialisation, S-mode can be activated and control can be transferred to the firmware payload.

Table 1: SBI calls provided by the SM. Some can only be used by the enclave, others by the host. The `call_plugin` SBI call is generically used to activate custom plugins that either the host or the enclave (or both) can utilise.

Caller	SBI Call	Description
Host	`create`	Request enclave creation
	`run`	Run the enclave initially
	`resume`	Resume enclave after suspension
	`destroy`	Zero and free enclave memory
Enclave	`stop`	Suspend execution
	`exit`	Exit cleanly
	`attest`	Request attestation report
	`random`	Retrieve a random number
	`get_sealing_key`	Retrieve the sealing key
Both	`call_plugin`	Invoke a custom plugin

3.4 Extended SBI

A RISC-V system that Keystone should be employed on will typically supply an M-mode SBI implementation such as OpenSBI. While the system's default SBI offers the host a certain amount of functionality (e.g. hardware information, peripheral control), it will not satisfy Keystone's requirements. Therefore, the SM implements another set of SBI calls in addition to the already existing ones. The RISC-V SBI specification allows for custom SBI calls to be implemented along standardised ones.

The SBI calls implemented by the SM can be categorised in two groups: calls offered to the host OS for enclave initialisation and control, and calls offered to the enclave for additional features such as attestation, random number generation or data sealing. Note that the enclave is able to use the host's system call interface using a mechanism called *edge calls*. This edge call mechanism does not depend on its own SBI call, instead the `stop` call is used. How edge calls work is thoroughly described in Section 3.9.

The individual SBI calls are listed and described in Table 1. The SM can differentiate between calls issued by the host and calls issued by the enclave using internally stored status data. Thus, the host cannot e.g. access sealed data by issuing an enclave SBI call.

3.5 Basic Operations

As described above, the custom SBI implemented by the SM is crucial for Keystone's fundamental operations. This section will explain the lifecycle of an enclave in detail.

Enclave *creation* is initiated by the host (more specifically, the host application and the host's Keystone driver). The host finds a continuous physical memory region not used by itself and copies the enclave's code and data into that region. It also creates and populates paging structures for the enclave, enabling the runtime to manage the enclave's virtual memory on its own. The root page table's physical address along with the base and size of the enclave memory region are transferred to the SM via the `create` SBI call.

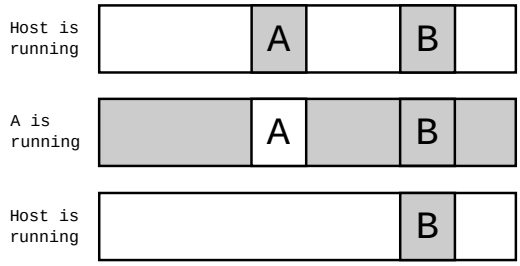

Figure 2: The same large memory region schematically shown at three different points in time: after creation of enclave A (top), at execution of A (centre), after destruction of A (bottom). Shaded regions are inaccessible by the current context because of PMP settings. Enclave B does not change its status in this example.

The SM verifies that the enclave memory region does not overlap with any other PMP region and measures the enclave's contents by walking the host-provisioned page tables and hashing the page contents. Finally, a high-priority restrictive PMP entry is created to disallow any further access to the enclave memory. The entry is propagated to other CPUs by sending an inter-processor interrupt (IPI). At this point, the enclave is runnable but not yet running, so that control is handed back to the host.

Execution of the enclave is started by the host issuing a `run` SBI call. The enclave to be run is specified through an ID generated at enclave creation. When handling the SBI request, the SM sets the host's PMP region to restrictive and the enclave's PMP region to permissive. Consequently, any enclave access to host memory will fail, while the enclave is now able to access its own memory. This change is not propagated to other CPUs to allow them to continue their host/enclave execution. Likewise, all other enclaves' PMP regions are left restrictive locally. After PMP configuration, the SM replaces the host root page table with the enclave one and then switches context into the enclave, effectively launching the runtime. To prevent the enclave from keeping control indefinitely, the SM sets up a hardware timer before entering the enclave.

Destruction of the enclave requires the enclave to have stopped. This can happen either by the enclave itself issuing an `exit` or a `stop` SBI call, or by a timer interrupt. After gaining control, the SM reverts the PMP changes made before enclave execution and resumes the host. The host calls `destroy`, making the SM zero all enclave memory and delete the corresponding PMP entry and all enclave metadata. Upon returnal of the SBI call, the host can now access the enclave memory region, enabling the host to reuse it for its own purposes. Any held enclave metadata can also be freed.

Figure 2 shows how the memory layout and permissions change over the lifecycle of an enclave. The other controlling SBI calls, `stop` and `resume`, work similar to `exit` and `run`, but without destruction (`exit`) or initialisation (`run`) procedures. They allow the enclave to suspend its execution to communicate with the host application via edge calls.

3.6 Multiple Enclaves

Keystone allows for a theoretically arbitrary number of enclaves to exist and run concurrently. In practice, the number of PMP CSRs restricts how many enclaves can be created, as each enclave occupies at least one PMP entry. It is hardware implementation specific how many PMP CSRs there are.

After enclave creation, the host can at any time create another enclave with a separate memory region. When one enclave is stopped, another enclave can be resumed or run while the previously executed enclave is suspended. On multi-CPU systems, the host can continue running on one CPU while at the same time an enclave is being executed by another CPU. It is also possible for a CPU to run an enclave while another enclave is being run by another CPU at the same time. This behaviour is possible because PMP changes are only propagated to other CPUs at enclave creation and destruction, but not when resuming or stopping an enclave. That way, a CPU that has access to a specific enclave's memory region will keep that permission while still being restricted from accessing memory regions of enclaves or the host running on other CPUs.

On single-CPU systems, the enclave suspension mechanisms (`stop` and timer interrupts) can be used to allow for pseudo-parallel multiple-enclave execution. It should also be noted that the Keystone reference implementation does not allow for the same enclave to be executed on multiple CPUs at the same time, thus restricting in-enclave threading to one core only. However, by making modifications to the SM, this restriction could be lifted by the enclave developer.

3.7 Attestation

Attestation in Keystone relies deeply on an existing and trustworthy RoT and thus requires custom hardware extensions. The RoT-signed SM hash and public key are accessible to the SM at any time. It cannot reliably forge the measurement due to the device's signature key only being accessible to the RoT and not the SM itself. Therefore, the SM measurement can be trusted as long as the RoT can be trusted.

The SM measures the enclave's pages at creation and keeps that measurement until enclave destruction. An enclave can request an attestation report through the SBI by calling `attest`, which will prompt the SM to generate and sign an attestation report that can then be transferred to the host application (and thereby to a remote verifier) using an edge call. The attestation report contains the RoT-signed SM hash and SM public key, the enclave measurement and up to 1 KiB of arbitrary enclave data. The SM signs the entire report using its own private signature key that was generated by the RoT. The 1 KiB enclave data is chosen by the enclave itself and is supplied to the SM as part of the `attest` SBI call. This can be used to distribute e.g. an eapp's own public keys, signed host-to-enclave transmitted data for remote validation or signed computation results. Prior to the SBI call, the supplied data resides within PMP-protected enclave memory and can only leave this protected memory region as part of the signed attestation report. Thus, its integrity is protected at all times, while its confidentiality is lost when transferring the report to the host, unless the data has been encrypted by the eapp.

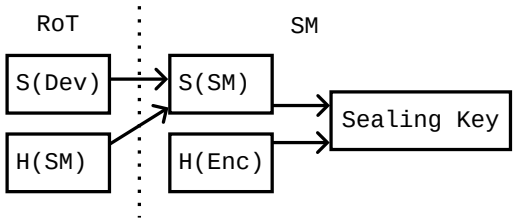

Figure 3: Various secret keys (S) and hashes (H) are used for deriving the sealing key. The RoT embeds the device's (Dev) and the SM's identity into the SM's key. The SM uses its own key with the enclave (Enc) hash (the enclave's identity) to compute the deterministic sealing key.

Attestation can be carried out at arbitrary moments during enclave execution. For example, an attestation report could be generated after finished eapp computation, on request of the remote verifier or prior to enclave exit. The attestation report is handed over to the host application, which cannot forge the report because of the trusted signature. To obtain the report, the remote verifier communicates directly with the host application. The verifier will use the device's public verification key to verify the SM hash and the SM's public key, and then use the SM's public key to verify the enclave hash and enclave data. Providing the device's verification key to the verifier is the hardware manufacturer's task, as it is the manufacturer's task to generate the device's key pair in the first place.

Of course, when verifying the components, only the authenticity of the report is checked. For validation of its contents (i.e. the hashes), the expected SM and enclave hashes have to be known. Generating the expected SM hash only requires the verifier to hash the SM binary, whereas generating the expected enclave hash is more complicated. Because the enclave hash calculated by the SM is not based on the enclave binary, but rather on the enclave's page contents only known after initialisation by the host, statically calculating the enclave hash is nontrivial for the verifier. The currently advertised approach [6] incorporates running a virtual machine on a device trusted by the verifier, executing a host OS and a modified host application along with the desired enclave binary. The modified host application makes the enclave request an attestation report and outputs the enclave hash part of the report to be obtained by the verifier.

3.8 Data Sealing

Data sealing in Keystone works by enabling the SM to compute a deterministic encryption key for the enclave to use in any way, requested by the enclave using the `get_sealing_key` SBI call. The key could for example be used to encrypt data that is then transmitted to the host application to be stored persistently. The encryption key must be deterministic for the enclave to be able to decrypt data even after a system reset, because the exact same key is needed for both sealing and unsealing. A given combination of a device, an SM and an enclave will always yield the same key, whereas a combination including any differing component will yield a different, yet also deterministic key.

The encryption key is derived using the SM's private key and the enclave hash (see Figure 3). Because the SM's private key is derived by the RoT using the SM hash and and the device's private key, it is bound to both the device's and the SM's identity. Including the enclave hash to compute the encryption key makes it impossible for another enclave to be given the same key. In order for an enclave to request multiple keys for different purposes, a freely chosen key identifier can be supplied when calling `get_sealing_key`.

If any component changes, the encryption key also changes. Any change to the enclave binary or the SM binary, or employing the same setup on different hardware, will lead to a different key.

3.9 Edge Calls

Edge calls are Keystone's means to offer host-enclave communication. The edge call mechanism is realised using shared memory between the enclave and the host. Upon enclave creation, in addition to the enclave memory region, the host allocates a shared memory region and includes it into its virtual address spaces. On `create`, the physical address and size of the region are handed to the SM, who in turn installs a PMP entry for the region, accessible by both the host and the enclave.

To perform an edge call, the eapp writes instructional data to the shared buffer and requests the runtime to suspend the enclave. After gaining control, the host application inspects the return value of the original `run` or `resume` operation that has just returned, making it possible to differentiate between an edge call and a timer interrupt-based enclave suspension. The host application then reads from the shared buffer to obtain the provided commands from the eapp.

The exact way how edge calls are handled by the host application is left open, but ordinarily an RPC-like approach is used to call procedures implemented by the host application. The eapp can also instruct the host application to invoke host system calls on the eapp's behalf. That allows the eapp to perform IO and networking operations by using functionality that is provided by the host OS and made available to the eapp by the host application and the edge call interface.

4 ENCLAVE DEVELOPMENT

This section highlights the developer's perspective of Keystone. On the one hand, a Keystone developer can craft own Keystone components or modify existing ones. On the other hand, an enclave application developer can develop eapps and host applications to be used with Keystone.

4.1 Extensibility

The high degree of customisation possible in Keystone allows for advanced additional features to better suit the user's needs.

As Eyrie [4] is a modular runtime, several modules can optionally be enabled when building Eyrie from source code. Not including a module decreases the amount of code eapps have to trust, while including certain modules offers functionality that may be necessary for an eapp's operation. Modules that already exist include networking and IO system calls, free memory management, encrypted page eviction and debugging capabilities. Of course, new modules can be developed, e.g. to enable thread management within an enclave.

Another way to customise the runtime is to replace it altogether. As Lee et al. highlight in [11], the seL4 formally verified microkernel [10] had previously been ported to Keystone to function as a runtime. However, its Keystone patches [2] have been deprecated, while the Keystone documentation [6] suggests an upcoming revival of the port.

The SM can also be extended by developing plugins or adding firmware hooks that are called during life cycle changes of the enclave (e.g. at creation or resumption). SM modifications are primarily used to benefit from custom hardware extensions that require M-mode. Plugins can be called via the `call_plugin` SBI call by either the host or the enclave and are implemented in the SM. An existing plugin that can be found in Keystone's source code [5] is the multi-memory plugin, offering an enclave to allocate another PMP-protected memory region for its free use. Firmware hooks have been used by Lee et al. [11] to implement hardware-dependent functionality like cache partitioning to prevent cache side channels or the usage of on-chip memory to prevent physical DRAM attacks.

Customising the Keystone host driver is only necessary to allow for changes to the SM or the runtime that depend on host support. For example, an SM plugin that can dynamically resize an enclave's memory region would depend on the host freeing adjacent memory.

Other than just expanding upon the existing SM, runtime and driver implementations, a completely new implementation of a component could be developed and employed without necessarily having to make adjustments to any other component. This would in fact be necessary for porting Keystone to a different ISA, if applicable, or to handle enclaves from other hosts than Linux.

4.2 Developing Enclave Applications

Keystone offers an SDK to eapp developers in order to decrease the amount of manual work needed to set up enclaves and run applications in them. While using the SDK is not necessary, it can be a helpful abstraction of underlying primitives, such as calls to the host driver or the runtime. The SDK comprises four categories of libraries and C/C++ header files that can be used respectively for developing host applications, eapps, remote verifiers or edge call interfaces. It should be noted that the SDK is still work in progress [6], so that there is still much manual labour required for creating enclave applications. When building applications, their binaries have to be linked against compiled SDK libraries, if the SDK is used. Other than that, eapp applications are typically ordinary ELF binaries, although the binary format depends on the runtime implementation.

5 RELATED WORK

Lee et al. [11] introduced Keystone, aiming to eliminate constraints and potential security vulnerabilities that other TEE designs impose on their users. For example, AMD SEV [13] only ensures confidentiality in regard to host-controlling adversaries or physical adversaries, but does not ensure integrity against these attacks. Intel SGX [12] on the other hand does offer full security against these adversaries, but not against side-channel attacks. ARM TrustZone [1] is resilient against neither physical nor side-channel attacks.

Another consideration when employing a TEE is the trusted computing base (TCB), which ideally should be minimal. While Intel SGX offers a small software TCB, Keystone's TCB includes the entire SM, all M-mode code (including the bootloader and SBI implementation) and arguibly the runtime. While the runtime could be minimal to only offer basic services to the eapp, the SM and other M-mode firmware cannot substantially be reduced in size. According to Lee et al. [11], Keystone's TCB lies in the range of thousands of lines of code. In contrast, TrustZones TCB comprises millions of lines of code.

In opposition to many other TEEs, Keystone is mostly software-based and does not need changes to existing hardware (although it requires certain hardware to exist in the first place). While hardware-based approaches like SGX and TrustZone obviously demand hardware modification, another TEE closer related to Keystone, namely Sanctum [8], also requires hardware to be modified. Like Keystone, Sanctum is aimed at RISC-V platforms.

From a developer standpoint, Keystone seeks to minimise the effort of porting existing applications to be used in an enclave. Largely unmodified RISC-V binaries targeting Linux can be run in a Keystone enclave through Linux system call emulation by Eyrie. For other binaries, another runtime could be used. But there do exist TEEs that cannot re-use existing code bases, but instead require applications to be modified and rebuilt or be completely re-written from scratch. Although sharing similarities with Keystone, Sanctum does not offer full application support, while SGX makes porting existing applications rather impossible.

6 EVALUATION

From the enclave developer's point of view, Keystone provides the advantage of full customisability as compared to hardware TEE solutions. The reference implementation already offers a usable code base that can be taken on to create one's own Keystone implementation.

On the other side, working with Keystone still requires a large amount of manual effort. The current state of the SDK does not offer a great amount of help yet, while some crucial tasks needed for basic functionality like attestation or edge calls are still very labour-intensive.

As mentioned before, calculating an expected enclave hash needed for verification of the attestion report is tedious and incorporates trusting a virtual machine and the host it is running on. Realising edge calls is not straightforward either, as the SDK does not provide a means to take the burden of implementing an RPC system off the developer. However, an edge call generator, Keyedge [3], is under current development[1]. This generator can automatically generate edge call code based on user-provided C header files and thereby takes the task of implementing complex encoding and marshalling procedures away from the enclave developer.

Additionally, the way data sealing works in Keystone does not allow components to be updated without prior data unsealing. As the sealing key is dependent on the hardware key, the SM and the enclave, any deliberate change to one of these components will lead to previously sealed data becoming unsealable. Keystone

[1] Although the Keystone documentation [6] mentions the Keyedge project as being in current development, the public repository [3] has not seen updates since the year 2019.

users should always keep this behaviour in mind in order to avoid unpleasant accidents.

7 CONCLUSION

Keystone proves to be a highly flexible TEE framework able to seamlessly be integrated with conventional RISC-V hardware. It offers protection against a reasonable range of adversaries, while being able to increase that range by further customisation based on hardware extensions. It can be arbitrarily altered, not imposing any design constraints on TEE architects.

The main caveat with Keystone is that it is not ready for production use, as stated in the Keystone online documentation [6]. Modifications like hardware-specific initialisation routines have to be made in order to make it deployable on a given hardware platform.

Eventually, Keystone may grow to be a mature, readily deployed, feature-rich trusted execution platform. Work is being invested in fixing issues like the impractical way to generate enclave hashes for verification, or the still labourious way of creating Keystone applications. Taking a look at projects like Keyedge [3] raises hope that Keystone's current problems are going to be smoothed out in the near future.

REFERENCES

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

[2] 2020. seL4 runtime for Keystone. (2020). https://github.com/keystone-enclave/keystone-seL4

[3] 2022. Edge Call Generator for Keystone Enclave. (2022). https://github.com/keystone-enclave/keyedge

[4] 2022. Eyrie enclave runtime kernel. (2022). https://github.com/keystone-enclave/keystone-runtime

[5] 2022. Keystone: An Open-Source Secure Enclave Framework for RISC-V Processors. (2022). https://github.com/keystone-enclave/keystone

[6] 2022. Keystone Enclave 1.0.0 documentation. (2022). Retrieved 12 December, 2022 from https://docs.keystone-enclave.org/

[7] 2022. RISC-V Open Source Supervisor Binary Interface. (2022). https://github.com/riscv-software-src/opensbi

[8] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[9] Palmer Dabbelt and Atish Patra. 2022. *RISC-V Supervisor Binary Interface Specification.* https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v1.0.0/riscv-sbi.pdf

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[11] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[12] 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. *Hasp@ isca* 10, 1 (2013).

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

[14] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.* https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[15] Andrew Waterman, Krste Asanović, and John Hauser. 2021. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture.* https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf