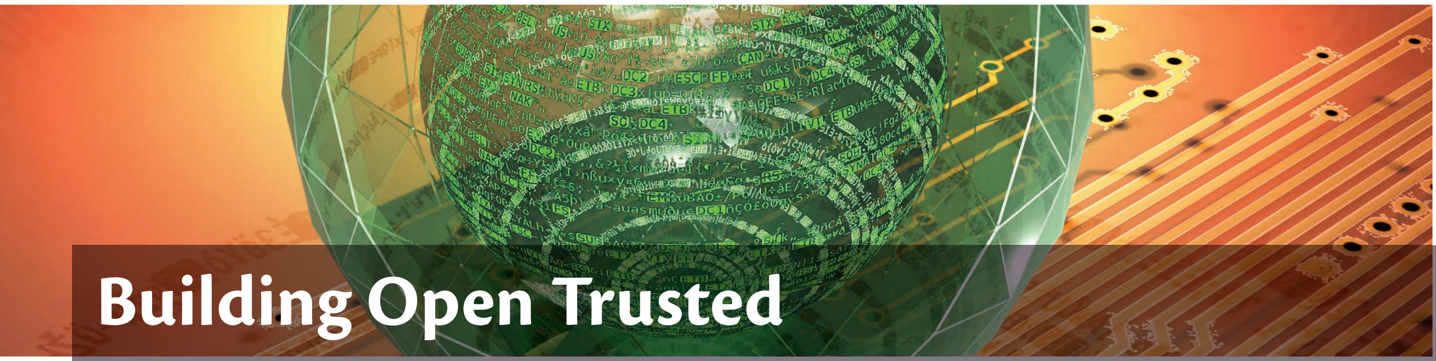

Building Open Trusted Execution Environments

David Kohlbrenner, Shweta Shinde, Dayeol Lee, Krste Asanović, and Dawn Song | University of California, Berkeley

Trusted execution environments (TEEs) are a growing part of the security ecosystem. Unfortunately, widely available TEEs are hampered by closed designs and a lack of flexibility. We outline the challenges to TEEs, advocate for extensible and portable open TEEs, and detail current efforts.

Trusted execution environments (TEEs), a rapidly developing part of the security ecosystem, are deployed on nearly every ARM smartphone, with cloud providers offering early support for Intel's and AMD's CPU TEEs. TEEs provide exceptional levels of isolation and protection to high-risk software while still sharing hardware and other resources with untrusted components. At their best, TEEs promise significant reductions in the number and size of trusted components and the elimination of trust in the hardware operator, even for remote computation. As with the advent of widely available hardware virtualization support, TEEs open up new usage models for shared hardware.

However, the demands on TEEs are quickly exceeding the capabilities established by commercially accessible options. To compound this, the closed-source and deep hardware integration in these designs limits their ability to evolve. We believe that the progress TEEs have made, while extensive, has been hampered by closed designs and a lack of flexibility. We thus advocate for open TEEs, systems that are open source, extensible, portable, and amenable to research as well as productization. We believe that the impact of other open efforts like RISC-V and Linux show potential gains if TEEs are opened up. In this article, we provide a background on TEE use cases and development, the challenges we see them facing, and the technical shifts for which we advocate.

Why Use TEEs?

Today's software stacks include a large amount of inherent design complexity. Even simple and sensitive applications, like a cryptographic tool, depend on numerous libraries and operating system (OS) services. Further, these code components are sourced from a wide variety of entities (e.g., standard libraries, device drivers, hypervisors, and kernels), many without a specific security focus. To compound this, the end user often does not have control over the infrastructure and software used for the remote deployment of their applications in cloud computing. Thus, most user applications have no choice other than to trust the underlying privileged computing stack. For a standard Linux environment on a bare-metal machine, this amounts to ~10 million lines of code or more. More importantly, it is well established

Digital Object Identifier 10.1109/MSEC.2020.2990649
Date of current version: 28 May 2020

that larger code bases are significantly harder to validate as "bug-free." Empirically, all of the code components in the described stack have been regularly susceptible to severe vulnerabilities. TEEs attempt to mitigate this ever-increasing number of components by reducing the amount of trusted code and other components, commonly referred to as the *trusted computing base* (*TCB*).

Partitioning

The partner problem to this is resource sharing, where an adversary (e.g., another cloud tenant) may operate on the same hardware as you. The solution to this, privilege separation, has long been a fundamental security principle generally solved by a privileged OS. Hardware supports this well, with in-built separation for the kernel and user-space software (e.g., rings 0 and 3 in Intel, exception levels 1 and 0 in ARM, and S- and U-modes in RISC-V). Extending the separation principle, user applications can be further partitioned, where parts of code and data that need stronger security to execute are in an isolated region protected from the rest of the program. This is accomplished in software via partitioning into multiple processes or through sandboxing methods. For example, consider a web server that uses HTTPS for connections that need secure cryptographic keys to establish secure channels.

A typical web server comprises thousands of lines of code, however, only a few functions require access to the cryptographic keys. Here, isolating the parts of data (that is, cryptographic keys) and code (in essence, cryptographic library implementation) significantly reduces the TCB and the attack surface for the most critical components. The advantage of this identification and partitioning is that even if such secure isolation incurs performance penalties, it is affordable because only a small part of the application requires such isolation.

Reducing and Protecting the TCB

Principles such as privilege separation and program partitioning help reduce the amount of code that needs safeguarding. There are several well-established techniques used to protect pieces of code and data. Purely cryptographic approaches (e.g., full/partial homomorphic encryption and multiparty computation) allow for direct operations over encrypted data to produce encrypted outputs. Because the data are never decrypted, all of the code processing the data need not be trusted. Although promising, these techniques are expensive and limited in expressiveness to the extent that they are not yet practical for all but a few real-world applications. Alternatively, verification techniques can ensure that the software-based isolation (e.g., a kernel isolating user libraries) is implemented and enforced correctly. The output of such a process is standard performant code, which is much faster than purely cryptographic approaches. However, such techniques are only as strong as the model verified against and can limit the nature of applications protected (e.g., cannot run multithreaded applications). Further, such verification is not automatic and has been costly with respect to human effort on a per-application basis. The iterative design and implementation required by a verification effort also adds significant costs. Lastly, purely software-based security hardening techniques attempt to achieve complete memory safety to ensure bug-free code.

Despite advances in safe programming languages (e.g., Rust), legacy software still often uses unsafe languages (e.g., assembly and C). Scaling hardening techniques to such code bases has well-known limitations that either require solving known-hard problems (e.g., pointer analysis) or approximating at the cost of losing the soundness and completeness of the analysis. More importantly, because a single bug in a hardened code base can eliminate all guarantees (e.g., buffer overflow in Heartbleed), such hardening techniques are best seen as part of a defense in depth.

A Cleaner Solution

The design philosophy of privilege separation uses a divide-and-conquer approach. By isolating the sensitive components in higher-security compartments, it directly reduces the amount of code assumed to be bug-free for secure operation. TEEs are an embodiment of this approach to the extreme possible while still sharing hardware. Specifically, TEEs provide a trusted hardware primitive wherein one can execute code in complete isolation from the rest of the software, including otherwise privileged code (e.g., an OS). TEEs are different from a standard ring-based hardware isolation because they can isolate low-privilege code from high-privilege code, if desired. Earlier incarnations of TEEs aimed at simply securing the integrity of small fractions of security-critical code (e.g., disk encryption) or entire execution stacks (including drivers, kernels, and user applications). Solutions such as trusted platform modules (TPMs) are exemplar implementations of these designs.

Modern and popular TEEs are more geared toward isolating and maintaining the integrity of user code from the rest of the software stack on the system. In these TEEs, the reversal of isolation and trust boundaries enables TEEs to execute small pieces of user code while completely removing the existing software stack of hypervisors, OSs, device drivers, and user libraries from the TCB. Figure 1 shows the schematic and trust model for a legacy stack versus the modern TEE stack. TEEs block

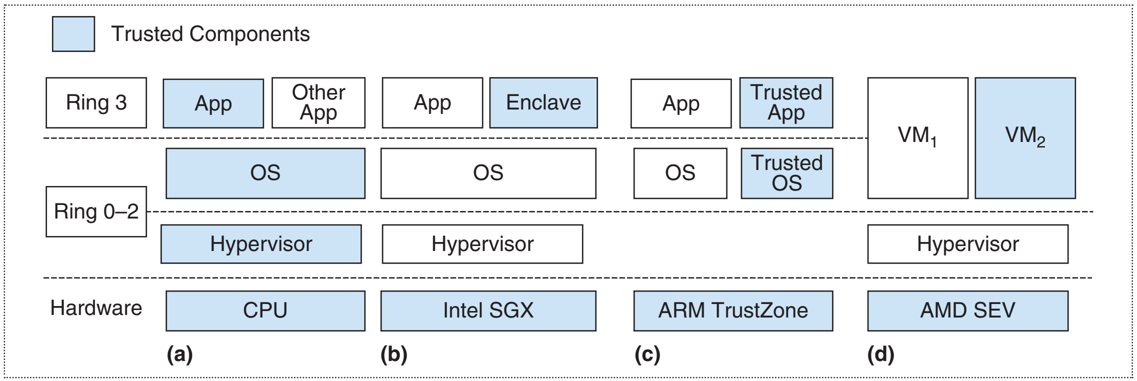

Figure 1. The schematic and trust stack for (a) a legacy stack, (b) Intel SGX, (c) ARM TrustZone, and (d) AMD SEV. The shaded components are trusted.

all access attempts from any non-TEE entity (e.g., an OS) to protected TEE code and data. TEEs are able to maintain backward compatibility with existing legacy software stacks not executing within a partition. For additional safety, the TEE-bound code can and should leverage all of the existing hardening techniques to safeguard the smaller TCB inside the TEE. Thus, TEEs present a realistic solution for protecting the TCB while preserving scalability, performance, expressiveness, and legacy support.

Overview

TEE Guarantees

TEEs leverage trusted hardware to enforce strong isolation over code and data. TEEs must trust various entities, such as hardware designers, manufacturers, fabrication processes, and so on. Specifically, we assume that these parties faithfully design the hardware features, do not insert back doors, and that the cryptographic designs and implementations are bug-free. TEEs ensure code and data isolation throughout the lifecycle of a safeguarded process. This includes secure boot, execution, storage, provisioning, and trusted input–output (I/O) paths. TEEs also measure and report the content of each stage of the system, from boot to application loading. Finally, TEEs can attest to the validity of the execution platform by providing a cryptographically signed proof with relevant measurements to the remote party. Thus, a remote party can first verify the signature and then deploy private computation on a TEE system. Note that solutions that provide properties, such as secure boot or dynamic root of trust (AMD skinit and Intel TXT), are necessary but not sufficient to instantiate TEEs. Such systems are generally a part of TPM-based systems and provide guarantees at the platform level, rather than at the CPU level.

In summary, TEEs typically provide the following three guarantees:

1. *Integrity*: The code and data cannot be tampered with (e.g., by running arbitrary code within a partition).
2. *Confidentiality*: The attacker cannot learn the runtime content of the application (e.g., secret keys and code control flow).
3. *Attestation*: Proof is provided to a remote party that the environment has not been tampered with and is safe.

Most TEEs explicitly put availability guarantees out of scope, mainly because even the benign execution of the OS needs complete control over system resources. However, specific designs built on top of TEEs can guarantee availability tailored for Real-Time Operating System deployments. Another important guarantee that is not considered for all TEEs is side channels. Because the adversary to a TEE includes privileged software (e.g., the OS), hardware-based side channels are a more serious threat than in a classical threat model.

Modern TEEs

TEEs derive from decades of similar efforts, but three designs have emerged from vendors to commercial deployment: ARM TrustZone,[1] Intel Software Guard Extensions (SGX),[2] and AMD secure encrypted virtualization (SEV).[3] These designs are refinements of proposals, such as XOM,[4] AEGIS,[5] Bastion,[6] and SecureBlue++.[7] For concrete understanding, we summarize the three major TEE designs in the next sections.

ARM TrustZone

This design divides the entire computing stack into two worlds: secure and nonsecure. Sensitive applications are run in the secure world as "trusted applications" and are isolated from access by the normal world. Isolation is enforced by the presence or absence of the not-secure (NS) bit on all of the operations, e.g., peripheral accesses, memory bus, and so forth. TrustZone-aware peripherals are expected to abide by the NS bit and disallow access across worlds. Although possible, TrustZone does not encrypt secure-world content by default. As TrustZone has only two hardware-enforced partitions (secure and nonsecure worlds), executing multiple trusted applications requires multiplexing via a dedicated secure-world OS. Further, the secure world must provision and manage its own resources because it cannot directly use services from the normal world.

Intel SGX

Intel SGX approaches the problem using a different granularity and creates an isolated virtual address space (an "enclave") to execute portions of user-level code. Each enclave is isolated from access by the normal content as well as all other enclaves. Enclave memory in SGX is backed by encrypted random-access memory pages, providing strong protection against adversaries with even physical access. Notably, SGX uses Merkle tree constructions managed in hardware to enforce integrity protection as well as confidentiality on protected memory. Because these enclaves still run on top of an untrusted OS and an untrusted host application, they must rely on the rest of the (potentially malicious) computing stack for resource management.

AMD SEV

The AMD SEV design takes yet a third approach and is focused on isolating entire virtual machines (VMs) from an untrusted hypervisor. SEV extends support for encrypted memory operations based on keys in the memory controller. By applying unique, nonvisible keys to each VM, SEV ensures that the hypervisor cannot inspect the content of the VMs it services. This protects entire VMs, rather than specific user-level applications and is thus more targeted at protection between the tenants in cloud environments. SEV has not historically supported memory integrity protections but has announced a further extension (SEV-SNP) that will offer some form of integrity.

Figure 1 shows the trust differences of these three designs. All of the designs use a secure boot process to establish root of trust and can perform attestation processes. The trusted environments created by TEEs are commonly referred to as *secure enclaves*, a term popularized by Intel SGX. The user code is thus said to execute inside a secure enclave.

Challenges to TEEs

As TEEs are prevalent in many commodity CPUs, they provide a strong set of security options across a diverse set of hardware. However, TEEs have not always managed to meet the security and flexibility demands required of them.

The Problem With Monolithic TEEs

Existing commercial TEEs are tailored to specific hardware designs by the relevant manufacturers. Understandably, this is driven by the security challenges that manufacturers see as pressing to its customers. This can be seen in the way a given TEE determines how to split trusted from untrusted, the threats it addresses, how keys and trust are established, and what happens when a compromise occurs. As a result of this approach, TEEs, such as SGX, SEV, and to a lesser extent, TrustZone, are *monolithic*, meaning that they provide a complete and static solution from threat model to application interface. For applications that match these expectations, monolithic TEEs provide an attractive option for a secure system.

However, the space of desired, and possible, application needs far outstrips the design space occupied by monolithic TEEs. With no ability to change the fundamental tradeoffs offered by a closed system like SGX, an innovative design is stuck layering unwieldy software stacks on top to work around limits. We believe that these limitations have slowed and constrained innovation in TEE internals. Experimental designs that have emerged from these constraints required significant reimplementations to merely simulate proof of concepts. Even the simplest design or implementation changes adopted have been delayed and subject to long timelines at the mercy of a few hardware vendors. Fundamentally, nontechnical limitations have posed a major hurdle in TEE growth and adoption.

Subverting TEE Security Guarantees

TEEs, like any other hardware and software artifact, are not immune to vulnerabilities; however, vulnerabilities in the trusted components affect TEEs more severely because of their expansive threat model. Note that software-level vulnerabilities in most components on the system (e.g., kernel bugs) have no effect on TEEs because they are considered untrusted. On the other hand, TEEs supporting firmware and basic I/O system (BIOS)-level components are trusted; bugs in the software at these layers may potentially compromise TEE guarantees. Thus, one of the design goals has been to reduce the amount of trusted software that is assumed to be bug-free.

TEE designs have gone wrong at several levels. This includes TPM flaws, a lack of integrity protections in AMD SEV, software infrastructure design bugs (e.g., SDKs, drivers, BIOS, and trusted libraries), and the fundamental limitations on underlying cryptographic protocols or assumptions (e.g., attestation design or anonymity attacks). They have further been subjected to new and unanticipated side channels (e.g., control channels in Intel SGX), the amplified effects of traditional attacks (e.g., cache attacks), emerging attacks (e.g., speculative side channels), and novel applications of well-known theoretical attacks (e.g., Iago attacks).

Implementation bugs in TEEs are less common or known, perhaps because most of them are closed source. As a recent example, an implementation bug in processors with Intel SGX along with Intel Processor Graphics [Common Vulnerability and Exposure (CVE)-2019-0117] leaked enclave information via DWORD0 and DWORD1 of a cache line. A quick survey of CVEs shows 10 and 49 vulnerabilities affecting SGX and TrustZone, respectively. We point out that other than publicly filed CVEs, there are several fundamental implementation bugs that are continuously iterated on, notably VM state protections in AMD SEV. All of the TEEs face challenges in verifying the correctness and safety of their most privileged components, but in closed-source designs, the community has a limited ability to engage.

To their credit, TEEs' R&D efforts have shown an agile and rapid response to most of the design and implementation threats. Even when rolling out hardware-level fixes to proprietary hardware is cumbersome, researchers have been successful in prototyping the mitigation techniques in principle. In comparison, coming up with patches for buggy software components has been relatively straightforward. The major hurdles in their discovery and mitigation have almost always been the inability to inspect causes and then independently test or deploy fixes. Thus far, TEE designers and manufacturers have been cognizant of flaws communicated via these feedback loops and have shown initiative in addressing them.

Expanding and Accelerating the Adoption of TEEs

The positive takeaway, despite threats to the validity of TEE security, is the evidence showing that there is a high demand for TEEs. Nearly all smartphones now employ some form of TEE (TrustZone or Apple's design), multiple companies offer products for SGX systems (Graphene, Fortanix, and so on), and major academic security conferences publish significant numbers of TEE-based designs and proposed modifications. This leads us to believe that a better way of designing and implementing TEEs will certainly accelerate their innovation and adoption. To this end, our main observations are fourfold. First, there is a diverse set of platforms and use cases that are not covered by existing TEEs. Embedded platforms are restricted to ARM's vision for TEEs and servers to AMD and Intel's specific threat models. Most of the effort at the moment is dedicated to making these modifications on a case-by-case basis. Second, porting legacy code is not easy or obvious because of the restrictions enforced by TEEs (e.g., syscalls are not supported in Intel SGX). Third, any design that requires hardware-level changes has a higher barrier to entry. For example, it is nearly impossible to "backport" many changes to older hardware. Finally, because TEEs do not cater to future platforms, they pose uncertainty as to what TEE designs will appear in future devices. In summary, manufacturers are a single point of failure and bottleneck. To make matters worse, they are the root of trust. This combination is a long-term threat to the advancement of TEEs.

Open TEEs

Our Proposal

We believe that future TEEs will benefit greatly from being as open as possible in access, scrutiny, and extension. We envision an open-by-design system that is easily adapted to new demands. With modularity baked into the design principles, we can take the established TEE primitives and methods that are known to work well and integrate them as needed. We hope that open TEEs will thrive along with open hardware and other open infrastructure. This includes supporting existing tools for formal verification, which has already shown benefits in efforts like Komodo. To us, *open*, in this context, means several things:

- *Open source*: The core of the TEE must be open source and available for developers and users to examine. Without this, trust is further centralized in the developer of the TEE, and it cannot be independently evaluated. This is also critical for verification efforts.
- *Flexible*: TEEs should be easily modified, repurposed, and updated. We have seen that, thus far, commercial TEEs have tended to be overly focused on a specific use case. A flexible TEE should build complexity tradeoffs and modularity into the core of the TEE system rather than leave it to later efforts.
- *Portable*: New TEE systems should be as hardware agnostic as possible. Relying on commonly implemented standards, rather than unique hardware support, allows for improvements to be applied widely.
- *Applicable to research and industry*: The more open a TEE system is, the easier it is to make advances in research prototypes and apply them to industrial products.

Toward Modular TEEs

From a technical standpoint, TEEs can satisfy the desirable requirements listed in the previous section by embracing a modular design as opposed to a monolithic one (e.g., as done in microkernels). A modular TEE is one in which the decisions involving its threat model and functionality are not set by the hardware manufacturer at design time, but by multiple parties, each modifying well-defined layers of the system.

This requires rethinking and redesigning existing tightly coupled components and building blocks. Once we achieve this, we can easily adapt a TEE to a use case, rather than adapting use cases to TEEs as is the norm today. A modular design will make it easier to integrate open source contributions and proprietary modules for products. More importantly, it will simplify the security arguments and help make additions amenable to verification. In fact, several efforts in this direction have already showed promising results. We provide pointwise examples of effort in this direction as well as the challenges faced and the technical approaches employed to overcome them.

Current Efforts

Recent projects have built the groundwork for either modular or open TEE systems by reusing existing building blocks. Komodo is one such modular system based on ARM TrustZone.[8] The majority of TrustZone-based systems were designed to operate on a "two-worlds" model, with a full OS operating in each. Instead, Komodo uses a small, but maximum, privilege monitor component to multiplex the "secure world" into many independent user-level enclaves. This approach is portable to a variety of TrustZone-enabled platforms and makes few assumptions about the applications that will make use of it. A significant focus of this effort is to promote modularity in enclave features and defenses by separating the hardware support (TrustZone) from the software support. More importantly, Komodo showcases the feasibility of verification as a result of decoupling hardware and software. On the openness front, Sanctum[9] is an open source TEE built on RISC-V, an open source Instruction Set Architecture. It resembles Intel SGX design but makes several novel security improvements, including side-channel defenses that are deemed out of scope by SGX at the cost of required hardware changes. MI6,[10] a follow-up to Sanctum, adds an additional layer of speculative side-channel defenses.

Hardware Integration Challenges

Apart from openness and modularity at the software layer, TEE systems require significant support from security features at the hardware layer. Intel SGX is built entirely in microcode and employs custom hardware additions tailored for performance (e.g., memory encryption engine[11]). Although such proprietary optimizations impart performance benefits, they obscure the internals of SGX. Worse yet, they lock SGX into a limited set of processors even within the Intel ecosystem. In comparison, ARM TrustZone offers relative flexibility in hardware peripherals and extensions. It supports multiple classes of ARM cores with a variety of optional hardware modules such as custom memory controllers (e.g., the TrustZone Address Space Controller). This has led to rapid innovation in TrustZone-based TEE designs and the subsequent adoption in commercial products (e.g., mobiles), despite the constraints from ARM. A larger degree of openness will foster and accelerate innovations in hardware including systems on chip, architectural/microarchitectural components, accelerators, and controllers. The vulnerabilities in TEEs present an additional challenge to closed TEEs. It is nearly impossible for vendors to fully disclose the details of a vulnerability and the subsequent fix for proprietary hardware without exposing internals. Hence, even if manufacturers are willing to share new design details, business needs prevent them and thus make it difficult for users and developers to trust them.

Hardware Modularity

We advocate for modularity, not only at the software layer but also at the hardware layer. By allowing the management of TEEs to evolve in software, independent of hardware revisions and manufacturing, we can accelerate TEE development and increase trust. This poses several technical and nontechnical challenges from an integration perspective.

The hardware guarantees for TEEs require unique defensive mechanisms. The challenge is in finding the right balance and abstraction such that the hardware primitives can be widely available and still support strong isolation and attestation. Building on the smallest possible set of hardware requirements while affording for future additions and optional features is an explicit objective of both Komodo (on ARM)[8] and Keystone (on RISC-V).[12] The end goal of these types of projects is to decouple the specific hardware from the TEE system, hopefully allowing for new hardware to be trivially added. The partner to these TEEs is open hardware efforts, such as RISC-V, that allow easy customization of IP blocks. A manufacturer can then assemble use-case-specific configurations while maintaining a standard hardware interface and modular software support. Such a design requires no application-level changes or redevelopment of software. For instance, a specific device can include a memory encryption engine if it is expected to face physical adversaries without changes to the software that already targets the base TEE system.

Ensuring the security of TEEs built on top for such varied hardware is more nuanced. The TEE system must be capable of using additional hardware features in a transparent way, e.g., including information about their configuration in the attestation report. For example, if the core is capable of aggressive speculation, the

TEE must more carefully manage other defenses and may need to disable speculation.

Smart integration with open hardware solves many challenges and encourages community trust. A TEE that is flexible on the specific hardware implementation allows for independent experimentation by manufacturers, operators, and developers.

Open TEE Middleware

Another approach taken by the community has been to create TEE middleware designed to enable applications to be developed against a platform-agnostic model. Google Asylo (https://asylo.dev/) and (formally Microsoft's) OpenEnclave (https://openenclave.io/) are the two most popular frameworks available. Both are open source, support Intel SGX systems, and have begun adding support for ARM TrustZone. These offer a flexible and significantly improved interface for application developers targeting enclaved execution at the cost of additional layers of abstraction and code. For example, OpenEnclave's TrustZone support requires the use of the Open Portable Trusted Execution Environment (OP-TEE) an open source secure-world OS for TrustZone (https://www.op-tee.org/). The stack for an application running on ARM with OpenEnclave now includes the TrustZone hardware, OP-TEE OS, the OpenEnclave framework, and any needed libraries. As the number of frameworks, libraries, and vendors involved grows, the objective of minimizing the TCB becomes far more difficult. It is still early for all of these frameworks, and it will take integration for several different platform's TEE mechanisms to see whether the promise holds up.

The Keystone Framework

We envision TEEs as an abstraction that guarantees a set of security properties (e.g., confidentiality and integrity). The corresponding trustworthy hardware is expected to provide several fundamental TEE operations (e.g., root of trust, secure boot, secure key store, and attestation) to achieve these guarantees. However, as we have summarized previously, the task of enabling meaningful use cases (e.g., executing applications) on top of these TEEs has dictated both the hardware and software implementations. Our insight is to decouple the hardware guarantees from the software abstractions required to enable specific use cases. Concretely, we propose focusing on identifying the basic primitives that a TEE requires to provide guarantees and expect the hardware to support these. We can then compose these TEE building blocks and use software to tailor the design for the needs of each use case. While doing so, we ensure the high-level security properties expected from a TEE abstraction with the smallest TCB footprint.

This is complementary to standardization efforts [e.g., Global Platform (https://globalplatform.org/)]. These efforts are beneficial for outlining what constitutes a TEE (e.g., for certification) and help retain interoperability across various TEE implementations via common interfaces. These proposals define the interaction with a TEE but do not solve the underlying technical challenges to actually achieve a secure TEE implementation.

In spirit, our proposal is more aligned with efforts such as Komodo; however, we select a different set of primitives (outlined in the following sections) because our primary goal is to achieve the fast prototyping of new TEE designs. These differences empower us to build a more flexible system on RISC-V, rather than on TrustZone.

To this end, our project, the Keystone TEE Framework,[12] endeavors to solve many of the discussed challenges by providing a platform for future TEE development. Keystone consists of a set of software components, guidelines, and tooling that allows for the creation of TEEs for standard RISC-V-based platforms. As in SGX-style enclaves, Keystone isolates each application into a distinct partition at runtime. Although SGX requires the host to do all of the resource management, Keystone allows each enclave to execute user- and supervisor-level code. It uses a simple and extensible reference monitor (the Security Monitor)[13] similar in concept to Komodo and Sanctum, running below the host OS to enforce TEE security guarantees.

Instantiating a TEE

Given a specific hardware platform, Keystone provides for the instantiation of a customized TEE environment entirely from software, with additional security guarantees and features available based on the hardware. Similarly, based on the intended use case, the functionality and security tradeoffs can be customized at software build time. Once the custom Security Monitor is complete, the measurement and source can be published to allow the validation of attestation reports. This flexibility extends after deployment, allowing for a new Security Monitor to be deployed via a software update. Any validation of future attestation reports originating from an updated device will then need to trust the new recorded Security Monitor measurement. We expect that the device manufacturer would generally be the one responsible for developing and updating the Security Monitor, but this is not a requirement.

Keystone additionally permits each enclaved application to run a private supervisor-mode component to manage virtual memory, support syscalls, and so on without relying on the host OS. This can be used to instantiate a similar split to TrustZone, where one secure OS manages multiple applications, or can allow for minimal enclaves with only a small shim in supervisor mode to communicate between the application and host.

Benefits

With Keystone, we hope to enable significantly faster TEE development and reuse. Keystone requires little from the hardware: merely a standard RISC-V core, a way to store device keys, and a secure bootloader. Due to RISC-V's privilege model and physical memory protection standard,[14] the rest can be handled straightforwardly in the software. This allows Keystone to be deployed on a large number of platforms and easily tested on those that do not support all the required hardware features. However, Keystone is not constrained to these features and can integrate easily with additional hardware for platform-specific features. As an example, Keystone currently supports several cache and physical adversary defenses on one development board by leveraging a configurable L2 cache controller. This lets Keystone have a flexible threat model, where the specific defenses applied can be tailored to the use case and hardware in question, without changes to the core primitives or applications.

Future Development

Keystone is an open project and encourages external contributions (https://keystone-enclave.org/). We have already integrated early contributions by adding build support for other hardware platforms and expect to receive more as new RISC-V platforms become available. By keeping all of Keystone's development open source, we encourage other research groups to use it as the basis for the development of security features and TEE designs.

Evolving TEEs

As TEEs are incorporated into new secure system designs, the requirements on those TEEs evolve. In the following section, we outline a few areas in which TEE design exploration is needed and that Keystone provides a good base to build on.

Distributing Trust

Current TEE trust and attestation systems rely on the perpetual trust of the hardware manufacturer. This occurs because, without external authority it is impossible to differentiate between a device manufactured in the past and a simulated device forged by the manufacturer today. Ideally, this lifetime-of-the-device trust could be separated from the initial trusted manufacturing, allowing trust in the manufacturer to be confined to manufacturing time. We see this as a natural extension of the TEE objective to minimize the required trust.

Accomplishing this requires a number of specific features in any proposed attestation scheme. First, the authenticity of a key presented by a device must be tied to an entity other than the manufacturer. This can be as simple as a third-party authority attesting to the time of creation of a device's key or as complex as multiple mutually distrusting parties each providing independent attestations. Second, the keying of a device must not involve the manufacturer observing any private device-specific key material. Some hardware already accomplishes this by having each device generate its own keys at provisioning time and only ever releasing public key material. This prevents any future compromise of the manufacturer from impersonating an older device but does not prevent them from emulating new ones. There are many possible approaches to distributing trust in a manufactured device's authenticity, each with its own challenges. Increased engagement from stakeholders in the TEE ecosystem will allow us to explore various approaches along with their tradeoffs. We aim to use Keystone as a prototyping framework to instantiate such designs.

Private TEE Infrastructure

A further extension of distributed TEE trust is the ability to use common hardware to build and maintain a completely separate TEE design and trust chain. In such a system, the hardware is provided without provisioned keys or with a key reprovisioning method. An organization can then develop and reprogram devices using their own TEE software and keys. At this point, the hardware manufacturer no longer has any role to play in the system. Trust is established through an attestation system designed and maintained by the organization deploying the devices. The threat model and design of the TEE software components is specialized to the needs of this organization as well. An open TEE system allows for the root of trust and trusted manufacturer to be changed as needed. Instead of being tied to a specific manufacturer, one can use commodity hardware and tie the attestation and trust model to whatever root they wish.

Partitioning Approaches

An area of consistent debate is the balance of functionality allocated to the host OS as compared to the enclave. Any functionality (e.g., virtual memory and I/O) managed by the host OS is inherently untrusted, but reduces the size, complexity, and attack surface of the enclave application. Numerous projects have used Intel's SGX to propose various tradeoffs up to a full OS running in user space inside the enclave. Similarly, not every application is optimally split into a single trusted and untrusted component and may benefit from further partitioning into multiple trusted components. Open TEEs allow for a wider range of experimental models for how programs are partitioned and responsibility is assigned. Notably, the inclusion of multiple privilege modes within a single enclave and simple secure enclave-to-enclave communications open up more fine-grained partitioning than was previously possible.

Although opening up existing commercial TEE systems will likely not occur, there are ample opportunities for building new, open TEEs. Open hardware (like RISC-V-based) platforms have already seen numerous academic proposals and several open frameworks targeting them. We believe that the closed and manufacturer-oriented design of current commercial TEEs will continue to slow advancement in this area.

TEEs offer an unusual opportunity for security engineering: applications are willing to segment themselves and take performance penalties for protection. We should make sure that TEEs can evolve at the pace needed to enable the growing interest in them. Open TEE frameworks on open hardware are the right way to do that. ■

Acknowledgments
We thank our anonymous reviewers for their insightful comments. This article is based, in part, on work supported by the National Science Foundation under grant TWC-1518899, the Center for Long-Term Cybersecurity, and DARPA N66001-15-C-4066. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The research was partially funded by RISE Lab sponsor Amazon Web Services, ADEPT Lab industrial sponsors, and affiliates Intel, Hewlett-Packard, Futurewei, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this article are solely those of the authors and do not necessarily reflect the positions or the policies of the sponsors.

References

1. "ARM security technology: Building a secure system using TrustZone technology," ARM Ltd., Cambridge, White Paper," 2013.
2. F. McKeen et al., "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware and Architectural Support Security and Privacy*, 2013, Art. no. 10. doi: 10.1145/2487726.2488368.
3. David Kaplan, Jeremy Powell, and Tom Woller, "AMD memory encryption," AMD Inc., Santa Clara, CA, 2016. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf
4. D. L. Chandramohan Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. 9th Int. Conf. Architectural Support Programming Languages and Operating Systems*, 2000, pp. 168–177. doi: 10.1145/378993.379237.
5. G. Edward Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the AEGIS single-chip secure processor using physical random functions," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 25–36. 2005. doi: 10.1145/1080695.1069974.
6. D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proc. 16th Int. Symp. High-Performance Computer Architecture* (*HPCA-16 2010*), Jan. 2010, pp. 1–12. doi: 10.1109/HPCA.2010.5416657.
7. R. Boivie, "SecureBlue++: CPU support for secure execution," IBM, Armonk, NY, 2012.
8. A Ferraiuolo, A Baumann, C Hawblitzel, and B Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proc. 26th Symp. Operating Systems Principles*, 2017, pp. 287–305. doi: 10.1145/3132747.3132782.
9. V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. doi: 10.5555/3241094.3241161.
10. T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 42–56. doi: 10.1145/3352460.3358310.
11. S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security Privacy*, vol. 14, no. 6, pp. 54–62, Nov. 2016. doi: 10.1109/MSP.2016.124.
12. D Lee, D Kohlbrenner, S Shinde, K Asanovic, and D Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th European Conf. Computer Systems*, 2020, pp. 1–16. doi: 10.1145/3342195.3387532.
13. J. P. Anderson, "Computer security technology planning study," James P. Anderson Co., Fort Washington, PA, Tech. Rep., 1972. [Online]. Available: https://apps.dtic.mil/docs/citations/AD0758206
14. Krste Asanović and Andrew Waterman, "The RISC-V instruction set manual Volume II: Privileged architecture," RISC-V International, Switzerland, May 2017. [Online]. Available: https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

David Kohlbrenner is a postdoctoral scholar at the University of California (UC), Berkeley. His research interests include the ways in which hardware design and implementation affects software security. Kohlbrenner received his Ph.D. from UC San Diego. Contact him at dkohlbre@berkeley.edu.

Shweta Shinde is a postdoctoral scholar at the University of California, Berkeley. Her research is broadly at the intersection of trusted computing, system security, program analysis, and formal verification. Shweta received her Ph.D. from the National University of Singapore. Contact her at shwetas@berkeley.edu.

Dayeol Lee is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His research interests include hardware/system-level security. Contact him at dayeol@berkeley.edu.

Krste Asanović is a professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences at the University of California (UC), Berkeley. His research interests include computer architecture, very large-scale integration design, parallel programming, and OS design. Asanovic received his Ph.D. in computer science from UC Berkeley. Contact him at krste@berkeley.edu.

Dawn Song is a professor in the Department of Electrical Engineering and Computer Science at the University of California (UC), Berkeley. Her research interests include artificial intelligence and deep learning, security, and privacy. Song obtained her Ph.D. from UC Berkeley. She is a Fellow of the IEEE and ACM. Contact her at dawnsong@berkeley.edu.

Digital Object Identifier 10.1109/MSEC.2020.3015405