PIE: A Dynamic TCB for Remote Systems with a Platform Isolation Environment

***Abstract*—Trusted execution environments (TEE) remove the OS and the hypervisor from the trusted computing base (TCB) and provide isolation to applications, known as enclaves. TEEs also provide remote attestation, which allows a remote verifier to check if the proper version of the enclave is running. However, TEEs provide only a static and restricted hardware trusted computing base, which includes only the CPU. While this might be acceptable for some applications, it is too restrictive for others, and falls short when one considers external hardware entities that are connected to the platform. Current proposals to include specific external components into a TEE exist, but these remain limited to very specific use cases and cannot be used dynamically.**

In this paper, we investigate platforms where enclaves can utilize a dynamic hardware TCB. We propose new security properties that are relevant for such systems, namely, *platform-wide attestation* and *platform awareness*. These properties allow a remote verifier to verify the current state and to define how the enclave reacts upon a change in connected peripherals. Finally, we present a prototype based on RISC-V's Keystone to show that such systems are feasible with only around 350 lines added to the software TCB.

I. Introduction

Trusted execution environments (TEEs) drastically reduce the trusted computing base (TCB) and provide security to applications, known as enclaves, without having to trust the operating system and hypervisor [10], [44], [11]. Thus, the attack surface is reduced by eliminating two of the largest sources of vulnerabilities for a system [8], [37]. TEEs make use of isolation primitives provided by the CPU to exclude all software but a single target application from the trusted computing base (TCB). To achieve this trust model, only the CPU is considered trusted; while all the other hardware in the system is explicitly assumed malicious. Even memory is not included in the TCB, and can only be used in conjunction with memory encryption and integrity protections. The extra encryption overhead for every off-core memory access comes with performance costs that TEE applications need to bear, but OS protected applications do not.

The downside of such an approach is the limitation in potential use cases. TEEs cannot communicate with any external device without going through the malicious operating system. Various applications ranging from trusted path [50], [15], [13] to data center accelerators [40] are excluded from the TEE ecosystem. Any such application with high-security requirements is usually solved by one of three approaches: i) a fully dedicated system, ii) a dedicated virtual machine (with trust in the hypervisor), or iii) blind trust in the operating system. We believe none of these approaches to be satisfactory due to either cost (i) or the need to trust codebases (ii & iii) with millions of lines of code [39], [5]. The static system architecture of TEEs limits their potential applications to trusted computing use cases, where a user possesses some secret data and wants to outsource the computation to a remote system. Many TEEs actually rely on some external peripherals already, e.g., Intel SGX and the monotonic counters in the management engine [32] or various academic proposals for trusted path using ARM TrustZone [28], [29]. However, all of these systems are custom-tailored for a single use case and do not allow for any flexibility as they must be configured statically at boot time.

In this paper, we investigate systems where the TCB can be dynamically configured across multiple hardware components. We call such a system platform isolation environment (PIE). The shift towards dynamic TCBs has wide-ranging implications concerning isolation, attestation, and integrity — some of the typical security properties of traditional TEEs. In order not to reinvent the wheel, we rely on the many existing TEEs on the processor [10], [11], [27] and on the rather new proposals to create fully blown TEEs on peripherals [40], [35], and combine them into a PIE. To achieve that, isolation must be expanded to the communication with other components such as other enclave or the peripherals. Similarly, attestation must span the entire platform. The attestation not only includes the individual measurements of the enclaves or peripherals, but also the communication links between them. To guarantee integrity for PIE, we rely on integrity properties in traditional TEEs where the enclave will halt execution upon a malicious manipulation. However, in PIE, the processor must be aware of the integrity of the connected peripherals as well. It needs awareness of the state of surrounding peripherals, a property we call *platform awareness*. This allows an enclave on the processor to be aware of the state of connected peripherals and react upon changes in them.

We present a prototype implementation of a PIE based on Keystone [27] running on an application-class RISC-V core [48] connected to peripherals that are emulated on a microcontroller. We extend Keystone's isolation mechanism to shared memory regions and use the trusted security monitor to enforce a strict one-to-one mapping of shared memory. Adding peripherals also has implications on the life cycle of enclaves as interrupts, and other events can happen at all times. In total, we only extend the TCB of Keystone by around 350 lines of code (LoC). On top of that, we propose a programming model that allows the same flexibility as previous systems for peripherals, where multiple applications interact with them simultaneously. We engineer this system as a set of intended and mutually attested enclaves. In particular, we propose three distinct enclaves, an application enclave, a controller enclave, and a peripheral enclave. Each enclave is completely isolated from everything else, and by default, in the spirit of TEEs, only needs to trust the hardware on which it is running. Finally, we carefully perform an extensive security analysis of our

proposal.

Contributions.

- In this paper, we extend traditional TEEs by introducing a dynamic hardware TCB. We call these new systems platform isolation environment (PIE). We identify key security properties for PIE, namely *platform-wide attestation* and *platform awareness*.
- We propose a programming model that provides the flexibility to the developers is comparable to the modern operating systems for developing enclaves that communicate with peripherals. The programming model abstracts the underlying hardware layer.
- We demonstrate a prototype of a PIE based on Keystone [27] on an open-source RISC-V processor [48]. The prototype includes a simplified model of the entire platform, including external peripherals emulated by multiple Arduino microcontrollers. In total, our modifications to the software TCB of Keystone only amount to around 350 LoC.

The rest of the paper is organized as the following: Section II gives a brief background that our work is based on. Section III provides the problem statement, system and the attacker model. Section IV and V presents the main idea and the programming model of PIE respectively. Section VI and VII describes the PIE prototype & its performance and the security analysis of PIE. Finally, Section IX and X describes the related research works and conclude the paper respectively.

II. Background

A. Hardware Background

1) RISC-V: RISC-V is a modern open-source instruction set architecture (ISA). In recent years, RISC-V has sparked immense interest by industry and academia, with many software and hardware proposals. As a consequence, multiple open source cores that implement various subsets of the RISC-V standard have emerged [48], [3], [17], [4]. Software-wise, RISC-V is supported by many projects, e.g., by GCC, linux, and QEMU amongst others. RISC-V has also become a popular prototype platform for security research [43], [11], [27].

2) Physical Memory Protection: Physical memory protection (PMP) is part of the RISC-V privilege standard [41]. It allows to specify access policies based on a physical memory range. The access policies can individually allow or deny reading, writing, and executing for a certain memory range. For example, PMP can be used to restrict the operating system (OS) from accessing the memory of the bootloader. Every access request into a prohibited range gets trapped precisely in the core and results in a hardware exception. There are a fixed number of available PMP entries, typically 8 or 16 [41], each of which allows to specify a separate memory range. Since PMP is part of the standard, it is already included in many currently available cores [17], [3].

3) Device Tree: The device tree is a list that accurately describes the physical memory mappings of a platform. It describes the central processor, i.e., its speed, its ISA, and at what address its cache starts. It also includes the DRAM base address and various other components on the die, such as the various internal and external buses. As such it is usually used by the bootloader and the OS to bootstrap the system. As some peripherals cannot be detected automatically, they must be present in the device tree, as otherwise they will not get recognized by the OS. The device tree is usually burnt into ROM and available to the bootloader and the OS. It can therefore be considered trusted.

B. Trusted Execution Environments (TEE)

Traditional trusted execution environments (TEE) introduce a small protected program called enclave. An enclave runs on the processor isolated from the attacker-controlled OS and hypervisor. Enclaves provide code integrity and data confidentiality, even in the presence of a physical adversary. Integrity of the enclave code at the time of deployment is ensured by remote attestation while the enclave data confidentiality and integrity during runtime are provided by various hardware mechanisms. For example, Intel SGX uses memory encryption and the Memory Management Unit (MMU).

C. Keystone

Keystone [27] is a TEE framework that utilizes PMP to provide isolation. Keystone relies on a low level firmware with the highest privilege, called security monitor (SM), to manage the PMP. The SM maintains its own memory separate from the OS and protected by a PMP entry. It also facilitates all enclave calls, e.g., it creates enclaves, runs and destroys them. To protect the enclave's private memory, the SM configures a PMP entry so that the OS can no longer access the enclave's memory. Upon a context switch, the SM re-configures the PMP entries to allow or block access to the enclave. For example, during a context switch from an enclave to the OS, the SM changes the PMP configuration such that the access to the enclave memory is prohibited. Conversely, on a context switch back to the enclave, the PMP gets reconfigured to allow accesses to enclave memory. Because the SM is critical for the security of any enclave and the whole system, it aims to be very minimal and lean. As such, the security monitor is orders of magnitudes smaller than hypervisors and operating systems (15k LoC vs millions LoC [39], [5]). There are also efforts to create formal proofs for such a security monitor [26]. Keystone also provides extensions for cache side-channel protections using page coloring or dynamic enclave memory.

III. Problem Statement

TEEs have had a tremendous impact on the landscape of secure computation. They eliminate the operating system (OS) from the trusted computing base, enabling very lightweight applications to be executed securely in an hostile environment. However, TEEs generally only provide secure computation on the processor cores and cannot easily communicate with the external devices. Many current applications would benefit from such an interaction, either enabling new use-cases with inputs from sensors or unlocking massive performance gains from accelerators. Current TEEs generally do not support such use cases because they rely on the attacker-controlled OS to facilitate communication to an external device. Thus, the OS is part of the TCB again, violating one of the primary reasons to

use a TEE. These use cases are usually solved by employing one of the following three approaches:

1) Design a fully dedicated system.
2) Rent a dedicated virtual machine and place trust in the hypervisor.
3) Continue relying on the operating system.

We believe none of these approaches to be satisfactory due to cost and the need to trust codebases with millions of lines of code [39], [5].

In this paper, we investigate Trusted Execution Environments spanning an entire platform which we call *platform isolation environments* (PIE). Before delving into it, however, it is important to define what extending a TEE to an entire platform means and what the desired requirements of such a system are. Some specific examples of PIEs already exist with single static external components in the form of Intel SGX connected to the management engine for monotonic counters [32], or ARM TrustZone based solutions for trusted path [29], [28]. However, we consider systems that can extend to virtually any kind of peripherals, allowing for dynamic reconfiguration of the TCB. To narrow the scope even more, we consider a PIE to be a combination of multiple TEEs on individual components, e.g., a TEE on the CPU [27] and one on the GPU [40]. A PIE should provide similar guarantees as traditional TEEs and eventually pave the way secure computation on an entire platform.

We identify three main challenges that need to be solved to realize a PIEs. First, enclaves on the processor need to be able to securely communicate to peripherals in a way that is isolated from the attacker-controlled operating system. Second, a remote verifier must be able to attest to all critical components (including their firmware) and the communication links between them, to assert whether it is configured properly. For instance, this might include asserting that only a particular application has exclusive access to one or several peripheral. And third, a PIE must be aware of the state of its components, e.g., it must be able to detect if a peripheral device is plugged out or replaced. This is the dynamic counterpart to the attestation above. We call a processor-local enclave that is aware of the surrounding state *platform aware*. In general, the full state of a system is too detailed to be processed every time it changes or to meaningfully assess whether it is safe for a remote stakeholder to provide its data. Therefore, concretely this last challenge requires defining what parts of the state (and state transitions) of a system are relevant to ensure isolation and integrity of a PIE.

In summary, we identify the following properties that a PIE should provide:

1. **Isolated communication:** An enclave is able to communicate with other entities on the platform while the communication channel is isolated from the attacker-controlled operating system. The communication channel should be efficient and should require minimal changes to the underlying TEEs.

2. **Platform-wide attestation:** The state of the entire platform, that involves the enclaves, connected peripherals, the secure communication channel between them, and the communication configuration must be verifiable by a local and remote verifier. A platform-wide attestation also includes the authenticity and integrity of the firmware of the external peripherals.

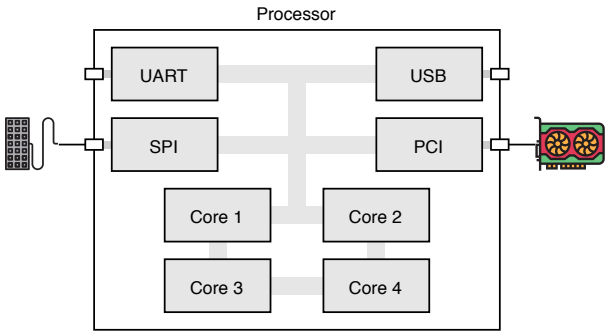

Fig. 1: **Overview of the assumed architecture of a platform.** The cores are connected to on-chip bus controllers. External peripherals are then connected, e.g., via USB or the SPI bus.

3. **Platform awareness:** Any enclave must be aware of the state of the entire platform at all times. Upon any state-change, the enclave should have the possibility to react, e.g., it may want to halt its execution when the peripheral it is communicating gets replaced.

A. System Model

We consider modern platforms that typically consist of multiple interconnected hardware components. In practice, this usually involves the central processor, memory, and various peripherals such as storage devices, graphics cards, sensors, and in general I/O devices. These components are interconnected over various buses. In reality, hardware components might themselves contain multiple parts such as independent integrated systems. However, we ignore their inner architecture and consider them as a single logical unit with a predefined interface. In the rest of the paper, we will use *processor* to indicate a central processing unit (CPU) and *peripheral* for any other external component. Figure1 depicts an overview of the platform architecture with connected peripheral devices. Most peripherals are not purely controlled by application-specific integrated circuits (ASIC), but they are managed by a microcontroller that runs some software called *firmware*. Depending on the peripheral, firmware can be of varying levels of complexity. For example, input devices such as keyboards usually only contain basic firmware. In contrast, the firmware of a data-center accelerator may be extremely complex. The processor also runs a vast software stack, from the BIOS to an operating system and user-space applications. Any application can interact with a peripheral through the driver in the operating system.

B. Attacker Model

We envision PIE to be employed in three main categories of applications and against two different attacker models: a local physical attacker, and a remote attacker. The applications are the following:

1) Local accessible peripheral: This is the case where both the platform and peripherals are owned by the user. The most common example is a trusted IO operation where the user,

the platform, and the IO peripherals are co-located. In such a scenario, the attacker cannot have physical access to the platform because the user must be present. If the user were to collude with the attacker, he could just enter malicious input directly. Hence, only a remote attacker makes sense in such a scenario.

2) Remote inaccessible peripherals: There exist a number of applications where a sensor needs to be read securely, and the user is not in physical possession of the peripherals, e.g., remote sensors. In such a scenario, the attacker could influence the environment to tamper with the sensor reading rather than compromising the platform directly, e.g., light a fire to influence a temperature sensor. Thus, in this scenario, it is in general impossible to cope with a local physical adversary.

3) Data center with an untrusted operator: The last category of applications that we consider is a data center where users want to offload high-intensity workloads. In such a scenario, one could consider a local physical attacker who has physical access to the platform because the data center provider is untrusted. However, a remote adversary could also be considered if one assumes the data center operator to be trusted.

For our attacker model, we chose to consider only a remote attacker because this threat applies to all applications above. Besides, the remote attacker makes PIE easier to integrate into today's hardware architecture: it requires only minimal changes in existing peripherals. Later in Section VIII-A we also discuss how our proposal can be extended to cope with a local physical attacker.

In summary, in this paper, we consider a remote attacker that remotely controls the entire software stack, including the OS or hypervisor. We assume that the underlying TEE governor, such as the security monitor (SM) for Keystone, is trusted. Note that operating systems and hypervisors have a much larger TCB compared to TEE governors, and an attacker that compromises them but not the TEE governor is usually considered when protecting traditional TEEs [11], [25]. Similar to existing TEE proposals, side-channel attacks remain out of scope [10] in our adversary model. Finally we consider denial-of-service attacks to be out of scope in this paper.

IV. Our Approach

In this section, we provide an overview of our approach how to dynamically extend the TCB of traditional TEEs running on the CPU cores to PIEs that also span external peripherals. PIE enables *platform-wide enclaves*, an extension of enclaves in traditional TEEs. Platform-wide enclaves consist of multiple distributed enclaves that run on various hardware components such as the CPU and peripherals as shown in Figure 2. Platform-wide enclaves also provide similar security properties to traditional enclaves, such as integrity, attestation, and data isolation from other enclaves and the attacker-controlled OS.

We will start by introducing the individual enclaves that make up a platform-wide enclave. After that, we introduce a shared memory model that allows enclaves to securely communicate to peripherals. Finally, we discuss how the enclave life cycle changes given these modifications and how a remote verifier can get proof of the state of a platform-wide enclave. We will also discuss the challenges that come along with platform-wide enclaves, namely how to handle interrupts, what to do with memory-mapped registers vs. direct memory access, and how to keep the overhead to a minimum.

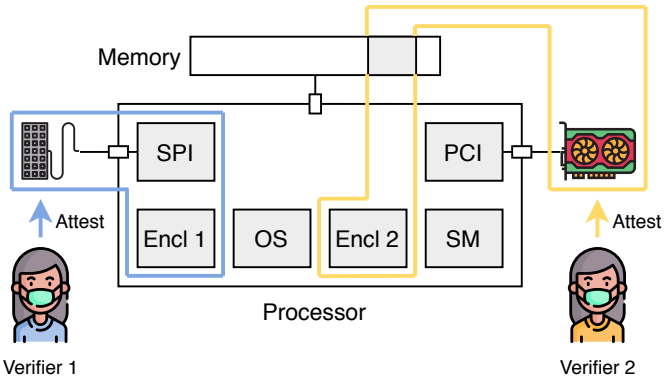

Fig. 2: **Platform-wide enclaves.** The figure shows two platform-wide enclaves that spans over multiple hardware components highlighted by blue and yellow outlines. One consists of `Encl1` and the keyboard that is connected over the memory-mapped SPI bus. Another one consists of `Encl2` that communicates with the GPU over PCI through DMA.

A. Enclaves within Platform-wide enclave

A platform-wide enclave consists of multiple distributed enclaves, e.g., a processor-local enclave and some firmware enclave that runs on the peripheral.

1) Processor-local Enclaves: Processor-local enclaves are derived from normal Keystone enclaves. As in Keystone, there is a minimal trusted security monitor (SM) that runs with the highest privilege. The SM facilitates all enclave functionality, such as enclave creation, destruction, or enclave calls.

The enclave's memory must be protected from adversaries and should only be accessible to said enclave. To achieve that, we use physical memory protection (PMP) from the RISC-V privilege standard [41] as pioneered by Keystone [27]. PMP allows the highest privilege mode to set up access policies for physical memory for lower levels of privilege. PMP entries can individually allow or deny read, write, or execute access to a fixed number of physical memory regions (typically 8 or 16 [41]). For example, a PMP entry is used to restrict the operating system from accessing memory used by the SM itself. These entries can only get modified by the SM. The SM initializes these entries upon enclave creation to protect the enclave's private memory. Any context switch to an enclave is facilitated by the SM, where it reconfigures the PMP entries to (dis)allow consequent access. More details can be found in Section II.

2) Enclaves on Peripherals: There are a large number of different classes of peripherals that may have unique behavior and would integrate differently into PIE. Because of this, we cannot exhaustively demonstrate the necessary modifications for every type of peripheral. Instead, we give a few examples of how they would be integrated and what modifications they

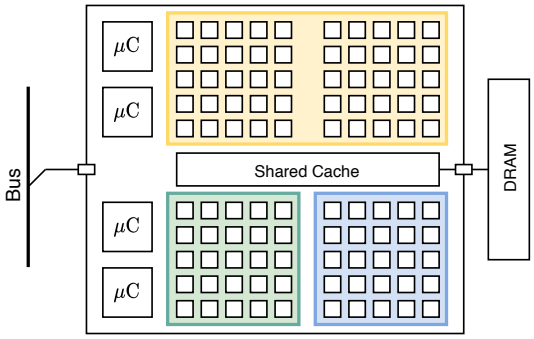

Fig. 3: **Example of a TEE on an accelerator** with multiple enclaves running simultaneously while being isolated from each other. In this example, there are three distinct enclaves running on a subset of the compute units of the accelerator indicated by a yellow, green, and blue overlay.

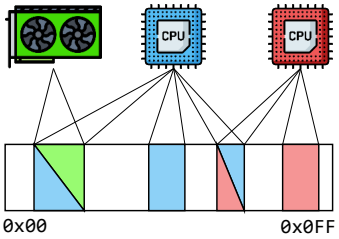

Fig. 4: **PIE shared memory model.** The enclaves on the processor core and the firmware communicate over shared memory. The solid color memory cells correspond to the private enclave memory regions. The cells with mixed color denote the shared memory regions between the corresponding enclaves.

would require. We go from the simplest peripheral we can imagine, a simple sensor, to one of the most complex, a full-blown accelerator for a data center. Most other peripheral should fall in between these two types of peripherals and thus may require modifications between these extremes. Note, however, that some special peripheral types might be entirely different and would need to be carefully analyzed.

a) Simple Sensor: Simple sensors, such as a temperature sensor, would only need to support a minimal form of attestation to be integrated into PIE. They must contain some key material to sign some statement about themselves. This is mandatory for (remote) attestation of a platform-wide enclave that includes an attestation of such a sensor. Usually, these sensors do not contain any secret data from a processor-local enclave and as such do not need to protect such data.

b) Accelerators: On the other hand, accelerators are very complex and require a wider set of modifications. Of course, they must support attestation, but they also may support isolation for secret data of multiple enclaves. Let us assume a case where multiple compute-intensive tasks are moved from the CPU and accelerated on said peripheral and they are run at the same time. The individual tasks' data should remain confidential and isolated, not only on the processor but also on the peripheral. Thus, such an accelerator requires isolated and attestable domains – in other words – enclaves that run on the peripheral. An example of multiple enclaves that run simultaneously on the same accelerator is illustrated in Figure 3. Such a sophisticated architecture on an accelerator is usually only warranted in data-center applications, where multiple stakeholders want to run accelerated applications simultaneously. Note that the same isolation could be achieved by time-multiplexing the applications that want to use the accelerator (c.f. Section V-A2). Recent research has shown how to build such a system on a graphics card [40].

3) Polling and Interrupts: Peripherals are synchronized with the processor with either polling or interrupts. Polling is a protocol solution and just requires the processor to check at a predetermined rate if new data is available from the peripheral. Such a mechanism can be used immediately in PIE. On the other hand, interrupts are a bit more complicated as they enable the peripheral to notify the processor that new data is available. Interrupts require some hardware support, which is readily available in all architectures nowadays. In RISC-V specifically, interrupts can be delegated from the highest privilege mode to lower ones. So, in our prototype of PIE, the SM can delegate individual types of interrupts either to an enclave or to the operating system[1]. Therefore, enclaves could also contain interrupt-handlers to, e.g., handle interrupts for a specific peripheral. However, in our prototype, we focus on polling.

B. Shared Memory

Platform-wide enclaves contain many individual enclaves on several hardware components. All of these enclaves must be able to communicate. We chose to facilitate this communication with shared memory regions. Note that other communication methods could also be used. However, we chose shared memory because it lends itself nicely to communication with peripherals. On our prototype, shared memory regions are centrally maintained by the processor to enforce isolation. This is due to how current platforms are designed with everything being connected to the central processing unit. We start by describing how multiple enclaves on the processor can use shared memory to communicate as shown in Figure 4. Then, we introduce how a processor-local enclave would share memory with an enclave on a peripheral. Finally, we talk about the differences in an enclave's life cycle that these modifications mandate.

1) Shared Memory between Processor-local Enclaves: As mentioned before, processor-local enclaves already rely on PMP entries for isolation. We reuse the functionality of PMPs to also protect shared memory regions. Therefore, our proposal does not require any changes to the processor itself as PMP is already part of the RISC-V standard [41], and thus, it is already part of many cores. The SM, however, requires some modifications, for example, to store the configuration details for every shared memory region in local memory. The SM needs to reconfigure the PMP entries on a context switch similar to stock Keystone. It also must guarantee that only two entities may have access to the same shared buffer at a

[1]In RISC-V external interrupts are handled by the platform interrupt controller (PLIC) and then multiplexed on top of the external interrupt signals to the core. Thus the SM has to contain a driver for the PLIC to figure out which peripheral the interrupt is from.

time. Additionally, the SM flushes the content of said buffers when one enclave is destroyed not to leak stale data.

2) Shared Memory with Peripherals: Platforms with a diverse set of peripherals are usually complex. Specifically, peripherals are not connected to the processor cores directly, but they communicate over buses, which are, in turn, controlled by bus controllers. This section describes how peripherals and processor cores can be paired in a realistic but simple platform architecture where the bus is controlled by a single hardware component on the processor die. Any communication with an external peripheral passes through an on-chip bus controller, as shown in Figure 1. Note that our proposal supports more complex platform architectures, but they are omitted here for simplicity.

Usually, a bus controller can be controlled via hard-coded memory-mapped registers. As such, access to such a bus controller can easily be isolated to a single enclave via PMP. The specific address ranges and model numbers are described in a format called device tree. The device tree tells the operating system where it can find which components in physical memory. The device tree is usually stored in on-chip ROM and is provided to the OS by a zero-stage boot-loader, and as such, it is considered trusted. Therefore, the SM has access to this specification and can verify that only one specific enclave can access the correct address range. However, such a setting limits the number of enclaves per bus type to one, e.g., only a single enclave can access all devices connected to the SPI bus at once.

Peripherals and buses that are used in a direct memory access (DMA) fashion behave differently. Such devices usually negotiate a shared memory range through memory mapped registers. The established shared memory range is then used for all future communication. Such peripherals can be easily integrated into PIE by configuring an enclave to have a shared memory region that exactly matches the shared memory region from the DMA peripheral. However, the DMA region can be established by an untrusted source, e.g., the OS. Therefore, the shared memory region supplied to the SM from the untrusted OS must be verified to detect any manipulation attempts. We propose to verify this in the SM by performing an initial attestation to the peripheral and to its DMA configuration[2]. In this way, the SM can verify that the peripheral and the enclave share the exact same memory range.

3) Enclave Life Cycle: Traditional enclaves have a very straightforward life cycle that includes three distinct states: idle, running, and paused. As an example, this is how a typical traditional enclave moves through these states: the enclave gets created and starts in the idle state. Then the enclave moves to the running state after a call from a user. Due to a timer interrupt by the scheduler in the OS, it is paused. It resumed again as soon as the scheduler yields back to the enclave.

In PIE, we introduce two further events into the life cycle to describe what happens when a shared memory region is altered. We call these events *connect* and *disconnect*. These new events are warranted by the asynchronous manner of peripherals as they can at any time prompt a disconnect event. An illustration of the new events and their interaction in our prototype of PIE is shown in Figure 5. The asynchronous disconnects are very critical as an enclave could end up continuing to use a memory region that is no longer protected due to a disconnect. Additionally, enclaves might want to provide graceful degradation and should not crash completely upon a disconnect. We solve both issues by splitting the disconnect event into an asynchronous disconnect and a synchronous disconnect. We consider both enclaves or peripherals of a shared memory region to have shared ownership over that region. If one of the entities dies, the other entity now gains sole ownership of the memory region. As such, asynchronous disconnects lead to the sole ownership of a previously shared memory region. In turn, the untrusted operating system can issue a synchronous disconnect command to the SM to free the shared memory region and notify the enclave of the disconnect. We mandate that before any connect command, the enclave must first receive a synchronous disconnect. If this was not the case, an adversary could disconnect a benign peripheral and reconnect a malicious one without the enclave noticing.

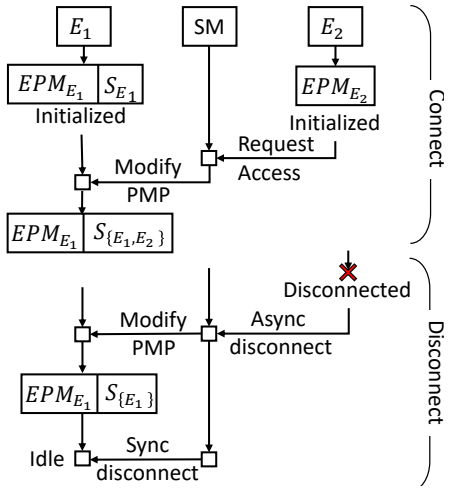

Fig. 5: **Enclave life cycle.** The figure shows the life-cycle events with two example enclaves E_1 and E_2. The security monitor (SM) acts as the intermediary in the `connect` and `disconnect` events.

C. Attestation of a platform-wide enclave

We extend the existing notion of attestation from enclaves running on the processor cores to platform-wide enclaves that run on multiple components of the platform. In the traditional sense, attestation ensures the current state of an enclave through a measurement of the code. In contrast, an attestation of a platform-wide enclave must also reflect the communication links between its components. To achieve that, we extend the individual attestation of an individual enclave within a platform-wide enclave with a list of unique identifiers of connected entities. These identifiers are assigned by the SM on the processor and can be used to specify to which enclave one wants to attest to. Of course, peripherals that want to get included in a platform-wide enclave need to support attestation in its simplest form: some key material and a certificate that proves that they are legitimate and coming from a known reputed manufacturer. Finally, the full attestation of a platform-wide enclave is built up from individual attestation

[2]We propose to do this in a single standard way over the shared memory to keep the TCB of the SM minimal.

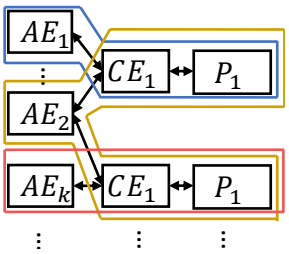

Fig. 6: **PIE programming model.** The application enclaves (AE) contains the application logic and communicate with peripherals through controller enclaves (CE) that contains the driver logic for the specific peripheral. In our programming model, we assume that there is only one controller enclave per peripheral. Multiple application enclave can connect to one controller enclave, and one application enclave can connect to multiple controller enclave. Three separate platform-wide enclaves are indicates by the blue, yellow and red outlines.

to all components and then cross-checked to verify that all communication links are set up correctly.

V. PIE Software Architecture: Unifying Programming Model

As we have seen in the previous section (Section IV), PIE enables isolated communication between different components within a platform-wide enclave over shared memory channels. Hence, we see that PIE's hardware architecture is significantly different from traditional operating systems and behaviour of existing applications. Hence, PIE poses a significant challenge to the developers to retrofit existing applications to fit within PIE, violating one of the main design goals of PIE to retain similar flexibility and usability to modern operating systems. In this section, we introduce PIE's programming model that aims to provide similar flexibility as modern OS's, namely by allowing multiple applications to access the same peripheral simultaneously. The programming model also tries to reduce the effort from developers to adapt existing applications to PIE's hardware architecture. This is achieved by providing an abstraction layer over the underlying hardware architecture that hides the lower level complexity. Our programming model dictates how developers can write applications, peripheral drivers, or peripheral firmware and how these entities talk to each other. Moreover, it allows flexible attestation of all its components. Note that the PIE's programming model is independent of the PIE's hardware architecture (refer to Section IV). For example, if a TEE implements the inter-enclave communication over the file system rather than the shared memory that we propose, or is based on Intel SGX instead, the programming model remains unchanged.

A. System components

PIE's programming model consists of three types of entities: application enclaves, controller enclaves, and peripherals as shown in Figure 6. The first two are similar to traditional enclaves that run only on the processor, as seen in Intel SGX or Keystone. Peripherals, on the other hand, are external heterogeneous devices that are attached to the platform. Even though enclaves and peripherals are very distinct entities, in PIE's programming model, we abstract them be to be homogeneous isolated execution environments. Hence, to developers, they appear to be instances of the same isolated execution environments, irrespective of whether it is an enclave running on the processor core or an external peripheral. All the enclaves and peripherals utilize shared memory to communicate with each other and use the identical system calls and attestation procedures. Together, they can form a platform-wide enclave that spans over multiple components.

1) Application Enclaves: Application enclaves are very similar to the established enclaves in Intel SGX or Keystone. In such a traditional TEE, enclaves cannot access peripherals without using the operating system as a mediator, as the OS handles all the drivers. In PIE application enclaves also cannot communicate with a peripheral directly. The application enclaves use shared memory to communicate with a controller enclave that is a peripheral-specific enclave that encapsulates the driver logic. The rationale of separating the driver from the business logic is two-fold, i) to avoid requiring the developers to ship driver code with their application, and ii) one controller enclave per peripheral allows multiple application enclaves to communicate with that specific peripheral in parallel.

2) Controller Enclave: The controller enclave contains the driver that enables communication with a peripheral. Note that application enclaves, standard (non-enclave) applications, and the OS cannot access the peripherals directly. The only way to communicate with a peripheral is through the controller enclave corresponding to said specific peripheral. Such a design choice isolates the peripheral device drivers: one compromised driver does not affect other peripherals. The controller enclave maintains an isolated communication channel over shared memory (e.g., in RISC-V, the PMP entry corresponding to a shared memory ensures that only participating enclaves have access to that shared memory) to application enclaves and the peripheral. To simplify the configuration, we assume that only one active controller enclave per peripheral exists at a time. However, any controller enclave can be replaced at the user's request.

Isolation of multi-application enclave session: In PIE, multiple application enclaves could connect to a single controller enclave in order to have simultaneous access to a peripheral. In such a scenario, the controller enclave keeps separate states corresponding to each of the application enclaves. Note that this is primarily a functional and then a security requirement as operations in one application enclave could affect the state of computation of another application enclave in case there is no isolation. For some peripherals, the controller enclave may need to reset the state of the peripheral when it switches to a session with a different application enclave (temporal separation). However, for peripherals such as GPU that support multiple isolated workloads in parallel, the state does not have to be reset.

3) Peripheral: We consider on-chip peripherals as well as external peripherals that are attached to the platform. However, we require them to store some key material from the manufacturer securely for attestation. Similar to application enclaves, peripherals also share memory regions with the controller enclave that is used as a communication channel. The firmware that runs on peripherals is also part of the platform-wide enclaves. For simple peripherals such as IO devices (keyboard,

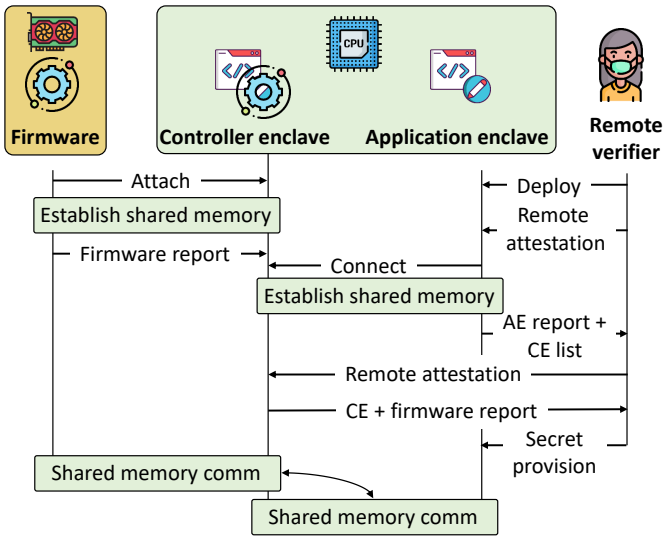

Fig. 7: **Flow of the remote attestation process** between the user, controller enclave, application enclave and the firmware.

mouse, etc.), GPS, and temperature sensors, the modification in the firmware is minimal. Essentially, this only includes adding a certificate from the manufacturer that proves that the peripheral is running the correct version of the firmware.

B. Platform-wide Attestation

In Section IV-C, we provide the hardware construct for PIE's platform-wide attestation. In this section, we describe how it related to the programming model components. The platform-wide attestation that enables a remote verifier to verify the state of the platform-wide enclave components - application enclaves, controller enclaves, and peripherals. Precisely, the state includes the state of the components of the programming model components and their communication channel (over shared memory in our case). Note that there could be more than one verifier (an example is depicted in Figure 2 with two remote verifiers and two platform-wide enclaves) corresponding to several platform-wide enclaves on the same platform. For each remote verifier, the TCB is constrained to only the specific platform-wide enclave that she is verifying. Hence a remote verifier does not need to trust any application enclave, controller enclave or peripheral that she is not using/attesting. Note that platform-wide attestation is built on top of the attestation scheme of the underlying TEE, e.g., the attestation in RISC-V Keystone or Intel SGX.

1) Attestation Components: The standard attestation report of a traditional enclave contains the following: the measurement of the enclave and the measurement of the low-level firmware (for example, the security monitor in RISC-V keystone). Both of them are signed by the platform key (known as the device root key). For platform-wide attestation, the verifier wants to acquire a signed measurement of the entire platform-wide enclave. That is a combination of signed attestation reports, including the report of the controller enclave, the configuration of the æ(initialization parameters), and the signed firmware report of the connected peripherals. We assume that the remote verifier already knows the public keys of the platform and the peripherals.

Enclave identifiers: When a new processor-local enclave is spawned, the SM assigns a unique identifier to that. This identifier uniquely determines the enclaves participating in a specific shared memory region. When the enclave is killed, the identifier is relinquished.

2) Attestation Process between Programming Model Components: Figure 7 depicts the sequence of the attestations between different components of the programming model. Note that the platform-wide attestation process starts from the verifier who initiates a remote attestation request of the application enclave. That, in turn, provide the verifier with the list for connected controller enclave and subsequently connected peripherals. The sequence of the end-to-end platform-wide attestation is the following:

a) Remote attestation of the application enclave: The platform-wide attestation process starts with the verifier who wants to attest the deployed application enclave. The purpose of the remote attestation of the application enclave is to ensure that the platform is running the intended version of the application enclave. The application enclave attestation report includes the list of identifiers of the controller enclaves that have shared memory channels with that application enclave. This report is signed by the platform's private key.

b) Remote attestation of the connected controller enclaves: As mentioned above, the user receives a list of identifiers of controller enclaves that have shared memory channel with the application enclave. The user then executes a series of individual remote attestation for the controller enclaves. The controller enclaves send the attestation report of themselves along with the certificate that is received from the connected peripherals. These reports are signed by the same platform key as of the application enclave attestation report. This proves that the application enclave and the connected controller enclaves are running on the same physical platform. Additionally, the controller enclave also states that the initiating application enclave has a shared memory channel with it. At the end of this process, the verifier who initiated the remote attestation of the application enclave receives the signed attestation report of the said application enclave, all the controller enclaves that are connected with the application enclave and all the peripherals that are governed by the controller enclaves.

c) Local attestation between controller enclave and peripheral: The local attestation process between a peripheral and a controller enclave takes place when a new peripheral is attached to the platform. This step is independent of the above mentioned two steps that only starts when the verifier wants to attest a specific application enclave. When a new peripheral is attached to the platform, controller enclave gets the list of attached peripherals from the SM (through the device tree). The peripherals come with a certificate that states the firmware version certified by the manufacturer along with the signing key from the manufacturer. The controller enclave issues a random challenge to the peripheral. It verifies if the peripheral actually owns the private key corresponding to the certificate. The peripheral signs the challenge and sends it back to the controller enclave for verification. After the local attestation between the controller enclave and the peripheral, the controller enclave knows that is it connected with a legitimate peripheral with a certified firmware.

C. Platform Awareness

As described in the previous sections, platform-awareness is one of the most important features of Platform-wide enclave, i.e., the individual enclaves know about the states of other enclaves that are connected to them. Note that it is infeasible that the enclaves keep track of all possible states of other enclaves. In our implementation of PIE, we handle specific states such as enclave termination, re-deployment, peripheral attachment, and detachment. Depending on the application enclave (AE), or the controller enclave (CE) or the peripheral, the policy on how to handle the changes in these states in other enclaves are as the following. The specific implementation of how the different components of Platform-wide enclave know about each other's states is dependent on the underlying hardware architecture. For example, for our implementation of PIE, we use the asynchronous disconnect to convey this information, as described in Section IV-B. For generality, in this section, we assume that the SM *notifies* the programming model components about each other's state. We denote a shared memory space as $S_{\{E_1,E_2\}}$ when it is shared between two enclaves E_1 and E_2.

1) AE is killed:: Assume that AE communicated with CE over $S_{\{AE,CE\}}$. In such a situation, only the specific shared memory space $S_{\{AE,CE\}}$ is destroyed. At the same time, the SM notifies the CE about the destruction of AE. A specific application may require that the peripheral is notified in the case the peripheral is handling sensitive data from AE. In such a scenario, the CE tells the specific peripheral that the AE was communicating with to report that the AE is terminated. Note that how the peripheral handles this call is also dependent on the implementation of that peripheral firmware enclave. For example, a peripheral that handles sensitive user data may decide to terminate the session completely (by zeroing out all the internal states) and destroy the shared memory between the peripheral and CE. Note that, along with the AE, multiple other application enclaves could be connected with the CE. In that case, the rest of the communication channels between the other application enclaves and the CE remain intact due to the isolation of shared memory regions enforced by the controller enclave.

2) CE is killed:: All the shared memory regions that are associated with the CE (this includes the shared memory spaces with both application enclaves and peripherals) are also zeroed out, and the respective AEs are notified by the SM. Zeroing out the shared memory also notify the peripheral that the respective CE is killed. Forcing the peripheral to reset. When the CE is reinitialized, the CE needs to reestablish the communication to the peripheral – requires the local attestation between the propel and the CE (refer to Section V-B2). The user of the AE that wants to connect with the CE also needs to perform a remote attestation of the CE to re-verify i) the CE is a legitimate one, and ii) the peripheral that is accessed by the CE is a legitimate one and the shared memory between them is set up as intended.

3) Peripheral is reattached, or the firmware enclave is changed:: The SM notifies the CE_p about the peripheral (p) changes and invalidates $S_{\{p,CE_p\}}$, the shared memory space between the CE and the peripheral. This also results in the destruction of all the shared memory that is associated with CE_p and p. All application enclaves AE that use CE_p must be notified. Similar steps are taken if the peripheral is reattached to the platform.

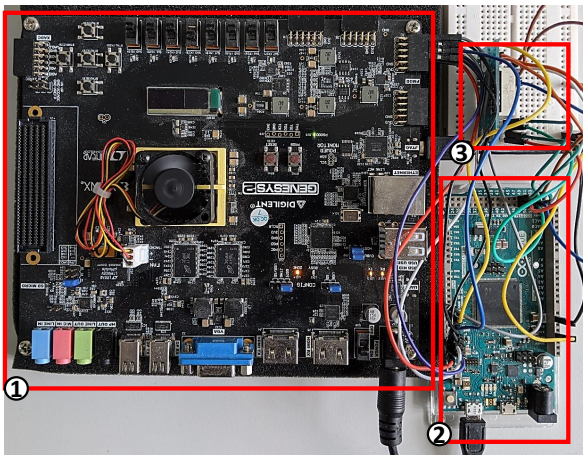

Fig. 8: **PIE prototype.** The figure shows different components of our prototype: ① Digilent Genesys 2 FPGA board, ② Arduino Due as the peripheral simulator, and ③ a seven-segment display unit as an example peripheral.

VI. Implementation and Evaluation

A. PIE *Prototype*

We implemented a prototype system for PIE that is based on the Keystone enclave framework [27]. In order to fully demonstrate such a system, we deployed an entire end-to-end example on a platform that consists of an FPGA that emulates the central processor connected to several Arduino boards that emulate peripherals, as shown in Figure 8. We also discuss the added complexity in the form of trusted computing base of our prototype. Since added complexity is hard to quantify, we fall back on the metric "lines of code" for the TCB.

1) FPGA platform: We base our system on the Ariane core [48]. Ariane is an open-source RISC-V 64-bit core that supports full blown operating systems such as Linux. It is an RC64GC 6-stage application class core that has been taped out multiple times and can operate up to 1.5 GHz. It also runs on an FPGA with reduced speed.

However, the core does not support physical memory protection which is critical to our protoype. In the process of this work, we added this capability. To perform the physical memory access checks, we designed a new PMP unit in 160 lines of SystemVerilog. The PMP unit is formally verified against a handwritten spec with yosys [45]. Two of these units are inserted into in the memory management unit (MMU) and are responsible to check data accesses and instruction fetches respectively. An additional unit is placed in the hardware page table walker to check page table accesses. Our implementation has a configurable number of PMP entries up to the maximum number of 16 mandated by the standard [41]. Our modifications have been contributed to the Ariane project and are open source [47]. Note that physical memory protection is part of the RISC-V privilege standard and as such is already available on many other cores [3], [31].

2) Keystone: We base our system on the Keystone enclave framework [27]: A low-level trusted security moni-

tor is used to run enclaves. We modified the SM to be able to connect two enclaves or an enclave and a peripheral. Specifically, we added three calls to the SM called `connect_enclaves`, `sync_disconnect_enclaves`, and `async_disconnect_enclaves`. These calls can be used to set up shared memory between two enclaves or peripherals specified by their identifier. We also modified the attestation procedure of Keystone to also include a list of identifiers for all connected enclaves. Our modifications only amount to 181 additional or modified lines of code. In total, the SM consists of around 2000 lines of code excluding SHA3 and ed25519 implementations which contribute around 4000 additional lines of code. Our extension thus amounts to 9% of added lines of code to Keystone (and 4.5% including the crypto libraries).

In Keystone, every enclave runs on top of a minimal runtime. Since the runtime is included in any enclave, its code is critical and part of the TCB. The runtime provides similar functionalities to an operating system: it handles syscalls and manages virtual memory. For our prototype, we had to add support to dynamically map shared memory regions into the virtual address space of an enclave. We modified 140 LoC out of a total of 3600 LoC for the stock runtime called eyrie.

On the OS side, there are many untrusted components to make it easier to create and run enclaves such as a software development kit and a kernel driver. These components also required numerous changes. However, they are not trusted and as such do not increase the TCB.

3) Peripherals: In our prototype, we emulated the peripherals using the Arduino Due microcontroller prototyping board (② in Figure 8). The board emulates a range of peripherals using the Arduino HID library. The Due is connected to the FPGA over 8 wires that run from the Arduino's GPIO pins to the FPGA's PMOD pins. We use two such 8 wire connections for upstream and downstream data, respectively. Note that in our implementation, we use a modified I^2C protocol over the GPIO as the communication channel between the peripheral and the FPGA. Due to the physical limitations of the PMOD pins, the channel's frequency is restricted to 8 MHz that yields 1 MB/s bandwidth. In the real world, the physical interfaces between the peripheral and the platform could be diverse such as USB, PCI-E, etc. For example, in this paper, we implemented a keyboard with the Arduino board and wrote a simple keyboard driver that interprets the GPIO signal from the Arduino. Additionally, we we use a PMOD based seven-segment display unit as an output peripheral (③ in Figure 8). The driver contains around 50 LoC and is incorporated into our example controller enclave. Additionally, we use the `USBHost` library that can emulate a number of USB peripheral devices on the Arduino. We use the Arduino cryptographic library for signing the challenge messages from the controller enclave during the local attestation. The Due uses 128-bit AES (CTR mode) for encryption, HMAC_SHA256 for message authentication, Curve25519 for key exchange, and SHA-256 for the hash function. We use `DueFlashStorage` library to implement the NVM flash that contains the key materials for the peripheral attestation. Our prototype implementation is approximately 2.5K lines of code.

PMP entries	logic	caches	total
0	472k GE	686k GE	1141k GE
8	497k GE	686k GE	1164k GE
16	531k GE	686k GE	1197k GE

TABLE I: Size of the default configuration of the Ariane core in gate equivalents (GE), synthesized in 22nm at 1GHz with varying number of PMP entries.

B. Prototype Performance

1) Performance of Enclave Communication: A comparison between traditional enclave communication and our shared memory model is quite pointless. Traditional enclaves communicate through the untrusted operating system. As such, they first have to perform local attestation to establish a shared key which is then used to encrypt every message. In our prototype, the enclaves can establish a shared memory region to communicate directly. This also requires some work by the SM similar to the local attestation for traditional enclaves. However, enclaves in our prototype do not need to encrypt ever message since only the two communicating enclaves can access the shared memory region. Thus, one approach requires encryption and the other does not. Since our platform does not support any kind of hardware acceleration for encryption algorithms, the performance difference between the two approaches would be huge. Therefore, we do not believe a comparison between these two systems is meaningful.

2) Performance within an Enclave: We report that the performance within an enclave in our prototype is equivalent to the performance of stock Keystone [27]. This is reasonable since we do not modify anything that would affect Keystones performance.

3) PMP Overhead: We measure the hardware overhead of PMP units within the processor pipeline for three distinct configurations: no PMP entry, 8 PMP entries, and 16 PMP entries. We instantiate the Ariane core [48] with the default configuration: including the floating point unit, 32KiB L1 data cache, 16KiB L1 instruction cache, branch history table of size 64, and a 16-entry branch target buffer. We synthesized this instantiation of the core in a 22nm technology at 1GHz. In order to provide a meaningful comparison, we provide the separate size of the logic, the caches, and the total amount in NAND2 gate equivalents in Table I.

4) Peripheral: The communication overhead between the platform and the peripheral device emulated by the Arduino due is very small. At the time of initialization, the peripheral and the platform exchanges handshake messages to perform local attestation. The initial handshake message is 60 bytes. Every message size of our modified I^2C protocol is 32 bytes. The combined initial latency introduced by signing is around 60 μs in average.

VII. Security Analysis

In this section, we informally analyze the security of PIE. We split the security analysis in three separate parts. First, we show how isolation from a malicious OS and other malicious peripherals is achieved. Then we analyze the attacker-controlled life cycle events of platform-wide enclaves, and finally, we discuss the security of platform-wide attestation.

A. Isolation

1) Malicious Operating System: The memory of platform-wide enclaves in our prototype of PIE is protected using PMP entries [41]. Recall that in stock keystone [27], PMP is based on the physical memory range and only allows the specific enclave to access its private memory. On top of this, we use additional PMP entries to protect the shared memory regions. Hence, only the participating enclaves can access the shared memory regions. Note that only the highest privilege level, i.e., the SM, can modify PMP entries. During a context switch, the SM re-configures all PMP entries such that the correct memory ranges are available again. The processor will throw an access fault exception upon any memory access into protected memory regions. The hardware page table walker also must behave according to the configured PMP rules. Therefore, miss-configured page tables cannot be used to leak any data from protected memory ranges.

The SM enforces a strict one-to-one mapping for every shared memory region, i.e. a shared memory region is strictly shared between two entities (e.g., a processor-local enclave and a peripheral). The SM also verifies that no overlap exists between the memory ranges. Note, that in the stock keystone, the SM performs a very similar validation. Thus, we only had to introduce minimal changes to add these new checks for shared memory.

2) Rogue DMA Requests: Malicious peripherals can try to access protected memory through rogue DMA requests. Mechanisms to restrict DMA requests already exist in other architectures, e.g., AMD IOMMU [2], Intel VT-d [1], and ARM SMMU [21]. These mechanisms process every DMA request and verify its validity according to some access policy. Any memory access attempt that does not fit the access policy is blocked. Currently, the RISC-V standard does not contain a mechanism to limit such DMA requests. However, an input-output variant of a PMP called IOPMP is currently getting discussed in a working group. It is supposed to enforce the configured PMP rules for non-RISC-V peripherals. Since our current prototype does not have any peripheral interface open to DMA requests, we do not need such protection. However, platforms that support DMA could implement mechanisms like IOPMP.

3) Malicious Application or Controller Enclaves: The attacker-controlled OS can spawn malicious application enclaves and controller enclaves. However, users should remotely attest before providing any secret to the application enclave. During the platform-wide attestation, the user can check the attestation report of both the application enclave and controller enclave and could abort if they are not an exact match with the intended enclave measurements. The platform-wide attestation also reveals any misconfiguration of communication links by an adversary. Note that this only verifies the static configuration of communication links. Upon any change to this setup, the external verifier might need to re-attest (c.f. Section VII-B).

We require the controller enclave to provide isolation between multiple connected application enclaves (c.f. Section V-A2). Hence an attacker-controlled application enclave can not compromise other application enclaves or controller enclaves.

Vulnerabilities within any of these application enclaves or controller enclaves could break the isolation guarantees of the data in that specific enclave. However, such an attack will not easily be expandable to connected enclaves. E.g., if a vulnerability in a controller enclave is found, only the data within that enclave is revealed. Any data that does not pass through this controller enclave remains confidential. In this way, we provide defense-in-depth and reduce the potential impact of vulnerabilities.

4) Malicious Peripheral: If an adversary manages to compromise the exact peripheral that is used by an enclave, then any data on said peripheral is forfeit. However, any data not passed to the peripheral and of other enclaves remains confidential.

We stress that certain manipulations of specific peripherals are always possible for an adversary. Consider, for example, a temperate sensor. Any local physical adversary can increase the real-world temperature and thus manipulate the sensor reading. However as we describe in out attacker model in Section III-B, the physical attacker is out-of-scope of this paper. Note that this only applies to sensors, accelerators cannot get tampered with in this manner.

5) Side-channel Attacks: In this work, we do not consider any side-channel attacks. However, various techniques could be applied to PIEs, such as cache partitioning [11], [49]. PIE adds additional components, e.g., controller enclaves, that must be protected from any leakage. We believe that all additions of PIE can be protected from side-channel attacks in a similar way as traditional TEEs [35], [**?**], [11]. However, more research is needed in this direction.

B. Lifecycle Events

As described in Section IV-B3, there are two additional events for platform-wide enclaves in PIE. `Connect` is used to connect two entities over a shared buffer. `Disconnect` facilitates a disconnect between the two enclaves. The `disconnect` is split into a synchronous and asynchronous event. Asynchronous disconnects only occur when one entity surprisingly dies. We protect the stale shared memory region by moving it to sole ownership by the enclave that is still alive. This enclave can then try to continue its execution. However, it will realize that the other entity has died as it does not react to any activity on the shared memory region. At a later point, the untrusted OS can issue a synchronous disconnect to notify the enclave and free the shared memory officially. Note that the SM mandates a synchronous disconnect before another `connect` command.

Due to this architecture, a stale shared buffer will never be made accessible to any untrusted entity until a synchronous disconnect happens, during which the enclave will officially get notified. The separate handling of synchronous and asynchronous disconnect events enforces protection for any secret data during an enclave's entire life cycle.

C. Attestation

In the attestation of our programming model, the remote verifier attests to individual enclaves and receives identifiers for any connected enclave. Here we discuss what would happen

if the adversary can kill the existing enclave and launch a new different enclave with an equivalent identifier, i.e., she can kill enclave A and launch A^* with the same identifier, where the code of A must not be the same as A^*. With such adversarial capabilities, time-to-check to time-of-use (TOCTOU) attacks must be discussed. An adversary could let the remote verifier attest to an enclave A that is connected to B. Then she could kill B after the verifier has attested to A but before he attests B. The adversary may then launch his own copy of enclave B' and manipulate the system's state such that B' receives the same identifier as B. The remote verifier will then attest to B' and find that the code measurement looks fine. However, we stress that B' cannot be connected to A because then A would need to receive a synchronous disconnect and would need to be re-attested (due to the configuration of A). If the adversary goes even further, she could try to replace A with a different and malicious enclave A^* and connect A^* and B', the verifier will see that B' has the correct measurement and is connected to the identifier of A. However, the verifier will want to provide its data to A using the shared secret they have established in the previous attestation. Obviously, this cannot succeed as the new enclave A^* cannot know the secret.

This analysis shows that even when the adversary can perform reuse attacks on the enclaves identifiers, she can never successfully trick a remote verifier into interacting with another platform-wide enclave than attested. As such, TOCTOU attacks are not possible on enclaves in our prototype of PIE.

VIII. Discussion

A. Local physical attacker

Until now we have considered a remote adversary that has full control over the operating system and hypervisor. However, TEEs usually consider a local physical adversary. In the context of a platform with multiple peripherals, it is clear that a local physical adversary can alter the physical world. E.g., she can point a flashlight onto a light sensor. Such physical world manipulations are always possible is this scenario. Thus, a physical adversary cannot be tolerated. Nevertheless, some other types of peripherals could be modified so that the platform can cope with a local physical adversary, e.g., accelerators that get their input from the processor and then perform some expensive computation. Therefore, we have designed our prototype with the idea in mind that it should be easily extendable to cope with physical adversaries.

The platform would require some hardware modifications in the processor as well as in the peripherals. First, the processor needs to support memory encryption and integrity, a typical mechanism that many TEEs already employ. In addition, the communication channel between peripherals and the processor need to be secured. Specifically, the bus needs to be extended to provide confidentiality and integrity over the transmitted transactions. These two mechanisms, memory and bus confidentiality and integrity, are enough to make a PIE cope with a local physical adversary.

We note that memory encryption and integrity verification must support very high performance in order not to slow the whole system down. However, multiple examples from industry and research [18], [23], [36] have shown how to achieve such a performance. Bus encryption on the other hand might not always require such a high performance. Simple sensors, for example, only send very little data over the bus. As such, extending simple sensors with bus encryption might not require additional hardware changes as the microcontrollers that are usually present on such devices are capable to encrypt in software. However, accelerators that rely on the throughput of the bus probably require a hardware encryption engine to accelerate their encrypted communication.

B. Limitation by Number of PMP Entries

The number of PMP entries in the RISC-V privilege specification is limited to 16. This makes sense if one considers the overhead of such entries on the size of a processor. The downsides of such a low number of PMP entries lies in the number of enclaves and respective shared memory regions that may co-exist at a time on a system. At the moment this means that with one shared memory region per enclave, only $N-2/2$ enclaves may exist at a time (for 16 entries that would be 7 enclaves). However, we stress that the isolation of enclaves can also be achieved using the memory management unit in a similar fashion as Intel SGX [10] or Sanctum [11][3]. In this way, the maximum number of enclaves is not constrained. We also note that MMU based isolation can also easily be extended to shared memory ranges.

C. Untrusted controller enclave

In our proposal, we assume that the controller enclave is trusted. But we can introduce some changes in our design to enable a enhanced privacy mode into the existing controller enclave at the expense of some reduced performance. It works like the following. After the remote attestation of the application enclave and connected controller enclave and peripheral, the remote verifier receives the attestation report of the application enclave, controller enclave, and the peripheral, including their public keys. Using these keys, the application enclave and the peripheral can establish a TLS session using the controller enclave as an untrusted transport layer. But note that the developers need to enable this feature in the peripheral firmware. Moreover, as all the cryptographic operations are executed in software, lower performance must be expected. This enhanced privacy mode can work alongside the normal operation (where the controller enclave is trusted).

IX. Related Work

In this section, we discuss some of the most relevant works to PIE. We also discuss the main differences with PIE to these works.

A. Trusted Execution Environments (TEEs)

1) SGXIO: SGXIO is a proposal by Weiser et al. [42] that builds on top of Intel SGX with the goal of allowing SGX to interact with input-output devices. They achieve that by introducing a trusted hypervisor that allows enclaves to access virtualized peripherals. Similar to our approach, SGXIO also considers a remote adversary. However, SGXIO is rather static in nature, i.e., all the peripherals have to be set up at boot time.

[3]There are efforts towards the hypervisor extension in RISC-V that would allow MMU based isolation without non-standard modifications.

After the system setup phase, no changes are allowed (connect new peripherals, etc.). It is not clear how enclaves are created and get access to a peripheral while preserving the confidentiality of previous enclaves that used said peripheral. Essentially, SGXIO allows for a static extension of the hardware TCB.

2) Graviton and systems based on it: Graviton [40] is a TEE that runs on an accelerator such as a graphics card. It can provide isolation between the data of multiple stakeholders that run tasks on the GPU concurrently. It also provides remote attestation of an enclave on the accelerator. Graviton was evaluated on a modern graphics card and shows that the predominant overhead stems from encryption of the communication to the processor. However, they also demonstrate that the overhead of around 20% is tolerable. Graviton would fit very well within a PIE as it is an excellent example of an enclave on a peripheral. It provides isolation for secret data and attestation reports. In addition, it shows that even some of the most powerful accelerators can be extended with a local TEE. Visor [35] is a system built upon Graviton [40] that proposes a hybrid TEE that spans over both CPU and GPU. Visor is aimed towards privacy-preserving video analytics where the computation pipeline is shared between the CPU (non-CNN workloads) and the GPU (CNN workloads) to increase efficiency. Visor addresses micro-architectural based side-channel attacks where a local physical attacker can use the data-dependent memory access patterns (e.g., branch-prediction, cache-timing, or controlled page fault attacks) to reveal elements on the video analysis (e.g., leaks pixel patterns). The communication between the CPU and GPU enclaves are encrypted. Additionally, Visor ensures that the traffic pattern between the CPU and GPU enclaves is independent of the video content.

3) ARM TrustZone and systems based on it: TrustZone is a system TEE provided by ARM for their system-on-chips (SoC) [44]. TrustZone applications run on top of a secure OS that is trusted and isolated from the standard operating system (also known as the rich OS). Isolation between the two worlds is achieved by an extra bit on the bus. However, there is no isolation between different TrustZone applications. Due to this limitation, mobile phone manufacturers usually only allow TrustZone applications that are signed by them. TrustZone only provides the lower level isolation property between the rich OS and the secure OS. Everything else, i.e., isolation between TrustZone applications, remote attestation, etc., has to be added to the secure OS [34]. There have been many proposals that try to improve on the capabilities of TrustZone [7], [22]. Sanctuary [7] enables user-space TrustZone enclaves. Sanctuary achieves isolation by running enclaves in their own address space in the normal world. However, Sanctuary is very similar to Intel SGX, and thus, it does not extend to peripherals. There exists proposals that enable additional security properties such as a trusted path by enabling direct pairing of peripherals (e.g., the touchscreen) to the TrustZone application. Such proposals include TruZ-Droid [46], TrustUI [30], SeCloak [28], VButton [29]. All of these solutions do not provide any form of peripheral attestation similar to PIE. They also do not consider other peripherals or how to dynamically allow an enclave to access a peripheral previously used by another enclave. Moreover, as mentioned earlier, the absence of isolation between the TrustZone applications makes the proposal mentioned above weak in inter enclave isolation guarantee.

B. Other Isolation Methods

Minimal hypervisors or operating systems [19], [24] can also achieve isolation, and some are even formally verified [24]. Usually, such hypervisors do not include any attestation primitives, but the cost of adding them is low. A PIE could also be based on a microkernel such as seL4. One would have to provide an interface for the malicious OS running in a virtual machine to interact with enclaves that run directly on top of seL4. It might even be possible to formally prove such modifications to provide an even stronger assurance of isolation. Other pure hypervisor-based isolation system include Virtual Ghost [12], Overshadow [9], InkTag [20], TrustVisor [33], Splitting Interface [38], Terra [16] etc. In all of these proposals, the hypervisors also take care of the scheduling. Hence the TCB is a lot bigger than our approach. Moreover, in none of the hypervisor-based proposals, peripheral awareness, and platform-wide attestation are considered. Isolation is the sole objective of these works.

C. External Trusted Intermediary Hardware-based Solutions

Fidelius [15], ProtectIOn [13], FPGA-based overlays [6], IntegriKey [14] are some of the trusted path solutions that use external trusted hardware devices that work as intermediaries between the platform and IO devices. These external trusted devices create a trusted path between a remote user and the peripheral and enable the user to send/retrieve sensitive data securely to and from the peripheral in the presence of an attacker-controlled OS. Such solutions provide a loose notion of peripheral awareness. However, platform-wide attestation and strong isolation guarantee is out-of-scope of such proposals.

X. Conclusion

In this paper, we investigate TEEs that span multiple hardware components, including the processor and external peripherals. We propose the first TEE architecture, PIE, which allows a dynamic reconfiguration of the hardware TCB and that can be used for general-purpose platforms. We identify key security properties needed to make TEEs extend to the whole platform, namely platform-wide attestation and platform awareness, and show how these properties can be achieved feasibly in a prototype. We implement a PIE system on top of Keystone, introducing only 350 LoC to the stock Keystone codebase. Finally, we present a hardware prototype based on an open-source RISC-V processor and emulated peripherals.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o." *Intel technology journal*, vol. 10, no. 3, 2006.

[2] I. AMD and O. Virtualization, "Technology (iommu) specification," 2007.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[6] A. Brandon and M. Trimarchi, "Trusted display and input using screen overlays," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–6.

[7] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[8] S. Checkoway and H. Shacham, "Iago attacks: why the system call api is a bad untrusted rpc interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, p. 2–13, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1353535.1346284

[10] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[11] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[12] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*, 2014.

[13] A. Dhar, E. Ulqinaku, K. Kostiainen, and S. Capkun, "Protection: Root-of-trust for IO in compromised platforms," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/protection-root-of-trust-for-io-in-compromised-platforms/

[14] A. Dhar, D. Yu, K. Kostiainen, and S. Capkun, "Integrikey: End-to-end integrity protection of user input," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 1245, 2017. [Online]. Available: http://eprint.iacr.org/2017/1245

[15] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh, "Fidelius: Protecting user secrets from compromised browsers," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 264–280. [Online]. Available: https://doi.org/10.1109/SP.2019.00036

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 193–206. [Online]. Available: https://doi.org/10.1145/945445.945464

[17] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[18] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[19] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[20] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, vol. 48, no. 4, p. 265–278, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2499368.2451146

[21] A. Holdings, "Arm system memory management unit architecture specification—smmu architecture version 2.0," 2013.

[22] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing {ARM} trustzone," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 541–556.

[23] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[25] I. Lebedev, K. Hogan, J. Drean, D. Kohlbrenner, D. Lee, K. Asanović, D. Song, and S. Devadas, "Sanctorum: A lightweight security monitor for secure enclaves," *arXiv preprint arXiv:1812.10605*, 2018.

[26] ——, "Sanctorum: A lightweight security monitor for secure enclaves," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1142–1147.

[27] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: A framework for architecting tees," *arXiv preprint arXiv:1907.10119*, 2019.

[28] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–13. [Online]. Available: https://doi.org/10.1145/3210240.3210334

[29] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "Vbutton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 28–40. [Online]. Available: https://doi.org/10.1145/3210240.3210330

[30] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2637166.2637225

[31] Lowrisc, "Ibex RISC-V core," https://github.com/lowRISC/ibex, 2020.

[32] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "{ROTE}: Rollback protection for trusted execution," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1289–1306.

[33] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 143–158.

[34] P. Ning, "Samsung knox and enterprise mobile security," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014, pp. 1–1.

[35] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor: Privacy-preserving video analytics as a cloud service," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poddar

[36] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2003, pp. 357–368.

[37] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, 2011, pp. 1–6.

[38] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 279–292.

[39] L. Torvalds *et al.*, "Linux kernel source tree," *Git Respository*, 2020.

[40] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 681–696.

[41] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual volume ii: Privileged architecture," *EECS Department, University of California, Berkeley*, 2019.

[42] S. Weiser and M. Werner, "Sgxio: Generic trusted i/o path for intel sgx," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 261–268.

[43] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and

A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v." in *NDSS*, 2019.

[44] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, 2008, pp. 21–30.

[45] C. Wolf, "Yosys open synthesis suite," 2016.

[46] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 14–27. [Online]. Available: https://doi.org/10.1145/3210240.3210338

[47] F. Zaruba, "Ariane RISC-V CPU," https://github.com/openhwgroup/cva6, 2020.

[48] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[49] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 89–102.

[50] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 616–630.