IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2026, No. 1, pp. 135–160. DOI:10.46586/tches.v2026.i1.135-160

It's TEEtime: Secure Interrupt Isolation for Normal-world Enclaves on Arm

Friederike Groschupp, Mark Kuhne, Moritz Schneider, Ivan Puddu, Shweta Shinde and Srdjan Capkun

ETH Zurich, Zurich, Switzerland, firstname.lastname@inf.ethz.ch

Abstract. Secure and direct peripheral access is an important building block for protecting sensitive applications on mobile and embedded devices: Otherwise, an untrusted OS or hypervisor can, for example, trivially intercept an app's secrets when it renders them or restrict an app's functionality by limiting the way it can interface with a peripheral. While Arm TrustZone supports secure peripheral access, its design choices limit the flexible deployment of apps in the secure world. Recent TEE designs address this by isolating applications outside the secure world, but at the cost of secure peripheral access. We propose TEEtime, a novel system that gives software running in normal-world isolated execution environments (domains) direct and secure peripheral access by relying on existing Arm TrustZone mechanisms and the secure monitor for enforcement. TEEtime introduces interrupt isolation as a novel key primitive; we design a fine-grained interrupt isolation framework for Armv8-A. We prototype TEEtime on Arm's FVP simulator and on the Purism Librem 5 phone, showcasing a Signal messenger app running alongside an untrusted OS.

Keywords: Trusted Execution Environments (TEEs) · Arm TrustZone · Device Isolation · Interrupt Isolation

1 Introduction

Securing sensitive applications from an untrusted OS or hypervisor on mobile or embedded platforms poses unique challenges: Apps on these platforms often need to interact with the user or the environment, e.g., through the touchscreen, sensors, or camera. Such interactions are typically controlled by the OS, meaning that even if an application's core logic is protected, e.g., inside a trusted execution environment (TEE), its device interactions can still be intercepted or tampered with. For instance, end-to-end encrypted messages received by a secure messenger application can be trivially read by the OS when rendering them. An additional concern is that the OS can restrict the way apps can interface with peripherals, inhibiting app functionality. For example, the use case of contact tracing was only possible after relevant sensor data was made available to apps [App20, Reu20].

The need for secure and direct peripheral access is reflected in the design of Arm TrustZone, which has long supported it. It is enabled by two mechanisms: (1) address space controllers (ASCs), which allow marking devices as accessible from the secure world (i.e., the TEE) only, and (2) secure interrupts, which can only be configured, observed, and handled by the secure world. A number of works rely on these mechanisms [PKS+21, PL22, GL22, DWY+22, SSWJ15, YLL+24], highlighting diverse use cases.

However, these approaches inherit TrustZone's fundamental constraint: There is only one architecturally secured "enclave", the secure world. Apps running in the secure world are generally considered significantly overprivileged and potentially endanger software in the normal world [BGJ+19, CSFP20]. As a result, it is infeasible to securely deploy

Received: 2025-07-15 Accepted: 2025-09-15 Published: 2026-01-16

sensitive but potentially untrusted apps (e.g., those with large code bases) within the secure world. Further, secure app deployment is usually severely restricted and often tightly controlled by the manufacturer (e.g., a phone manufacturer only allowing deployment of their own secure services) [PS19, HZY+24, SSW+15, And].

Consequently, several novel TEE architectures propose deploying sensitive applications outside the secure world for mobile platforms. These approaches either avoid TrustZone entirely, e.g., pKVM [Gooa], Gunyah [Cen], or Arm Realms [Arma], or fix its limitations by moving sensitive apps to the normal world [BGJ+19]. However, they do not support direct IO, making use cases such as secure messaging in a protected environment infeasible.

We close this gap with TEEtime: A system that extends such isolated execution environments deployed in the normal world (*domains*) with secure and direct access to devices—removing untrusted intermediaries (e.g., the legacy host OS/hypervisor). Drawing inspiration from TrustZone's secure device access, TEEtime provides two core capabilities to domains: (1) Exclusive and unbrokered access to the control and data regions of a peripheral to protect the confidentiality and integrity of data exchanged with the peripheral. (2) Secure configuration and handling of interrupts, addressing the asynchronous nature of peripheral operation by protecting information leakage based on interrupt patterns [DLLZ16] and ensuring the integrity of interrupt delivery.

We design TEEtime to work with two different deployment modes for domains: One where domains run interleaved on the platform [SSW+15], with at most one domain being active at a time, and one where different domains run concurrently on different cores [BGJ+19]. We observe that to enable exclusive device access, we can apply methods similar to those proposed by previous work for memory isolation: Configure peripheral-focused ASCs to allow access from non-secure code only when the owner domain is running (or from the respective core). The situation is more complex for interrupt isolation. The Arm interrupt system is designed such that the configuration of interrupts is tightly intertwined, with up to 32 interrupts sharing one configuration register [Arm24], rendering ASC-based approaches infeasible. We solve these issues by leveraging the secure monitor (SM) in EL3 to enforce fine-grained isolation of the interrupt handling system. The basic idea is to partition the interrupt handling system such that only the interrupts belonging to the domain currently running (or running on the respective core) are enabled, while all other interrupts are inactive and protected. Further, we configure the interrupt system so that interrupts are guaranteed to be routed to the correct domain. This tight handling of the interrupt system ensures that domains cannot observe or change the interrupt behavior of other domains, closing the door to interrupt-based attacks on app confidentiality and execution integrity [DLLZ16, SSK+24, SSBS24].

An additional challenge TEEtime faces is that devices are discrete and unique resources (e.g., in comparison to memory or compute time) and more complex realistic deployments would require multiple domains to have access to the functionality of a peripheral. Therefore, we discuss the different sharing modes TEEtime supports, and how they enable reasonable use case scenarios based on different functionality, security, and peripheral requirements.

To demonstrate the feasibility of TEEtime, we implement our design on a smartphone, the Purism Librem 5 with an NXP i.MX ARMv8-A CPU [Pur], and deploy a functional Signal messenger in a domain that is running alongside, but protected from, the phone's default OS. We further deploy TEEtime on the Arm FVP simulator [Armb] and develop a range of domains with varying peripheral requirements that we run aside a full OS, showcasing the versatility of TEEtime. In summary, we make the following contributions:

- Motivate and define the notion of interrupt isolation.
- Design, implement, and evaluate an interrupt isolation framework for Armv8-based systems using GICv3. This framework is independent of the underlying memory and peripheral isolation mechanisms.
- Discussion of the implications of our design on the sharing of peripherals among multiple isolated environments.

A video of our Signal messenger domain is available at https://youtu.be/FUUax5m5pqY.

2 Background

2.1 Arm Architecture Fundamentals

Armv8-A cores offer four privilege layers, from EL0 (lowest privilege) to EL3 (highest privilege). EL3 hosts the secure monitor (SM), which is part of the platform's trusted computing base (TCB) and performs low-level and sensitive tasks such as trusted boot or power management. Other ELs can invoke the SM through a secure monitor call (SMC).

Platforms with TrustZone extensions distinguish between secure and non-secure states for EL0 to EL2. Software running in secure state is more privileged than and protected from software in non-secure state: The current state of the processor is determined by the NS bit, which can only be changed by software running in EL3. Commercial off-the-shelf Arm devices are commonly equipped with address space controllers (ASCs) such as the TrustZone Address Space Controller [Arm14], which allow state- and sometimes core-based configuring of parts memory and peripherals as accessible from secure state only [CMSP22].

Traditionally, the TrustZone extensions have been used on Arm devices to protect trusted sensitive workloads: they run in secure state, isolated from the untrusted software running in the normal world. However, time has shown that this solution is insufficient for many use cases for two reasons. First, the secure world is over-privileged. As a result, any application running in the secure state increases the size of the system TCB, which is undesirable. Concerns about the impact of the secure world on the security of the normal world have been raised, as attacks that exploit vulnerabilities in the secure world have been presented [ZB22]. Second, the secure state only provides one isolated world, making it infeasible to run sensitive workloads from different stakeholders [SSW+15]. Both points have led to the secure state often being locked down, i.e., only a very limited set of applications are allowed to be run in the secure state and profit from its protections.

2.2 Normal World Isolated Execution Environments on Arm

The concerns about the TrustZone extensions mentioned above have led to a stream of work providing isolated execution environments in the normal world [SSW+15, BGJ+19, ZHN+23]. We refer to such environments as *domains*. The advantages of such domains are that (1) an arbitrary number of them can be created and run on the same platform, allowing different stakeholders to deploy their sensitive payloads and (2) these domains are 2-way isolated, i.e., the untrusted "host OS" is protected from the domains as well. This means that apps that are considered sensitive but not necessarily trusted can be deployed.

Previous work describing such domains on Arm has focused on the isolation of compute time and memory. The isolation of memory is performed by ASCs that are configured such that software can only access the memory regions belonging to the domain by marking other regions as secure. This allows for protection of conventional trusted services, such as generating cryptographic material, but not for applications requiring peripheral access.

While different works rely on different components to configure and enforce isolation, e.g., the SM in EL3 [ZHN+23] or a dedicated deployment in S-EL1 [BGJ+19], one common denominator is that the TCB does not manage resource allocation or the lifecycle of domains. Usually, tasks like assigning compute time (i.e., performing scheduling) and assigning memory regions (i.e., performing memory management) are delegated to the host OS. The TCB receives corresponding requests, checks their validity, and performs the necessary actions, such as dispatching a different domain or updating the memory isolation. This approach maintains a small TCB.

2.3 Interrupt Handling on Arm

Arm platforms use the Generic Interrupt Controller (GIC) to deliver interrupts from interrupt sources such as devices to cores. The GICv3 (and onwards) consists of the memory-mapped *distributor* for the configuration of interrupts and the *CPU interfaces* for the handling of interrupts through system registers. Interrupts are uniquely defined by its interrupt ID (INTID). Through the distributor, software configures an INTID to be en- or disabled, its priority, whether it's level- or edge-triggered, and its affinity (i.e., its target core). This configuration is stored in GIC registers shared among INTIDs.

When a source generates an interrupt, its status becomes *pending*. The GIC forwards the interrupt to a CPU interface based on the INTID's configuration, which then redirects execution to the interrupt handler. Once the software acknowledges the interrupt, it is *active* until software signals the end of the interrupt and it becomes *inactive* again.

The GICv3 supports the TrustZone extensions and allows INTIDs to be assigned to different security groups: Group 0 secure, Group 1 secure, and Group 1 non-secure. For the purpose of this paper, we refer to them simply as secure and non-secure interrupts. The main effect of these groups is that non-secure software cannot change or observe the configuration or state of secure interrupts. Further, non-secure interrupts are signaled as interrupt requests (IRQs) to the CPU and routed to EL1/EL2 in non-secure state. In contrast, secure interrupts are signaled as fast interrupt requests (FIQs), which are not received or handled by non-secure software and are routed to EL3.

2.4 Secure IO in Trusted Computing

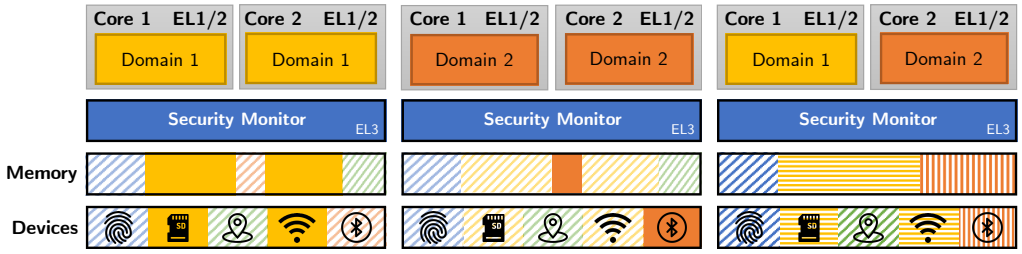
Trusted computing typically follows the paradigm that the TCB does not manage resources such as memory or compute time, but outsources this task to the untrusted OS. The TCB verifies and enforces access control information set up by the untrusted operating system. This design paradigm has led to multiple systems with extremely small TCBs that clearly remain in the realms of formal verification [KEH+09].

However, such systems usually disregard secure IO, which should follow a similar paradigm. Instead, protected applications rely on an OS or hypervisor to broker access. This is, for example, the case in traditional hypervisor setups [BDF+03], where the hypervisor fully virtualizes devices, but also in more recently proposed trusted computing architectures, such as pKVM [Gooa], which relies on virtIO and the host to provide device access, or Arm Realms [Arma], where the untrusted host manages device interrupts. Usually, the applications cannot verify that what they receive is valid data from the peripheral and that their output remains confidential and untampered. Other isolated execution frameworks do not support device access for isolated environments at all [BGJ+19, ZHN+23]. This lack of direct access to peripherals and interrupts is unsatisfactory for the developers and the users of sensitive applications that interact with peripherals, as they need to include large OS or hypervisor code bases in their TCB. In general, relying on virtualization for domain isolation is unsatisfactory, as full hypervisors are complex and large pieces of software, which pose another big attack surface [SKLR11, KSRL10].

Arm TrustZone is one of the few TEE architectures that supports direct device access for its protected secure world. However, apart from the general drawbacks of TrustZone, trusted applications still have to rely on the secure OS or hypervisor that is typically deployed by the manufacturer and cannot implement their peripheral stack themselves [PS19].

3 Design

TEEtime is designed to support multiple isolated, rich domains on Arm platforms. This includes not only execution and memory isolation but also the secure assignment of devices and their interrupts to these domains. We briefly describe how we handle the isolation of



(a) Temporal: Domain 1 active **(b)** Temporal: Domain 2 active **(c)** Spatial

Figure 1: Isolation of execution, memory, and devices. Blue resources belong to the SM, yellow to domain 1, orange to domain 2, green is unassigned. Diagonal lines: inaccessible to NS software. a) Domain 1 is active: only its resources are accessible to NS software. b) Domain 2 is active: access modifiers have been flipped accordingly. c) Two domains are executing concurrently on different cores. Access control is based on core identifiers. Horizontal lines: accessible to core 1, vertical: accessible to core 2.

execution time and memory for our base isolated execution environment before describing the design of TEEtime's new major components: device and interrupt isolation.

3.1 Threat Model

We consider a system in which multiple domains are deployed in the non-secure world EL2–EL0; the host OS or hypervisor are considered to be a domain as well. Domains might run either interleaved or concurrently on different cores. We assume mutual distrust between domains and that an attacker can control one or several domains, e.g., by exploiting domain vulnerabilities or by convincing the user to install malicious code. A domain can selectively expose interfaces to provide services to other domains. Domain developers decide if they trust the exposed interface of another domain. We make no assumptions about the general trustworthiness of such interfaces or interactions.

We assume hardware and firmware are implemented correctly and work as specified by the manufacturer: We trust the highest privileged software layer, e.g., the secure monitor (SM), and assume its code is public and verifiable. Further, we assume that the platform and its peripherals are integrated correctly. Finally, we assume that software running in the secure world is trusted or is moved to a protected TEEtime domain.

Side-channel attacks have been used to leak confidential data from protected environments [ZSS+16]. These attacks are out of the scope of this work. We refer to orthogonal work for possible mitigations of such attacks [RLT15]. We do not offer any additional protection against these attacks beyond what is available on today's platforms, but we aim not to introduce any additional weaknesses.

3.2 Base System: Isolation of Memory and Execution

Previous work has shown how to assign compute time to domains in different modes: Either domains run interleaved [SSW+15] or concurrently on different cores [BGJ+19]. We refer to these modes as *temporal* and *spatial* sharing. Our baseline system supports both modes and enforces isolation of execution by instrumenting the SM during context switches between domains: The SM saves, clears, and restores the domain state during context switches. This includes flushing caches and saving and restoring all read-and-write registers accessible to ELs 2 to 0. It further relies on commonly present hardware-based memory protection mechanisms, i.e., the ASC, to protect domain data and code in memory. In temporal sharing (Figures 1a and 1b), the SM configures memory regions belonging to the next domain as accessible by marking them as non-secure and the rest as inaccessible by marking them as secure access only. In spatial sharing (Figure 1c), the SM re-configures

the ASC during a context switch such that each core can only access memory belonging to the domain they are executing. Previous work [BGJ+19, ZHN+23, SSW+15] has detailed temporal and spatial isolation of execution, code, and data on Arm platforms.

Domain Scheduling. We designate one domain to be in charge of assigning compute time and cores to domains and call this domain the *scheduling domain*. The scheduling of domains is independent of any scheduling that might happen inside domains (e.g., if an OS is deployed, it is still in charge of scheduling its apps, while a bare-metal workload might not require its own scheduling). Using a domain to schedule other domains reduces the TCB, but puts extra trust in the scheduling domain. One option is to deploy a dedicated scheduling domain with a minimal and trusted scheduler, which could be considered an extension of the SM. Minimal trusted schedulers for TEE systems have been explored in existing work and are orthogonal to this work [ABPM21]. To invoke a different domain, the SM is called to context switch, i.e., to initialize and restore the state of the new domain and re-configure memory protection before handing control to the scheduled domain.

For temporal sharing, the SM sets up a secure timer that triggers after a time period specified by the scheduling domain and ensures that all cores are either suspended or set up for executing the scheduled domain before handing over control. If the domain does not yield execution before the specified time, an FIQ triggers that hands control to the SM, which in turn resumes the scheduling domain. As the domain cannot change the configuration of the secure timer, and the interrupt is routed to secure software, the domain cannot avoid being preempted. The scheduling domain has the privilege to invoke and preempt other domains; therefore, domains are not guaranteed availability. For spatial sharing, the scheduling domain assigns one or more cores to the scheduled domain, which the SM initializes. The scheduled domain controls the cores until it yields them.

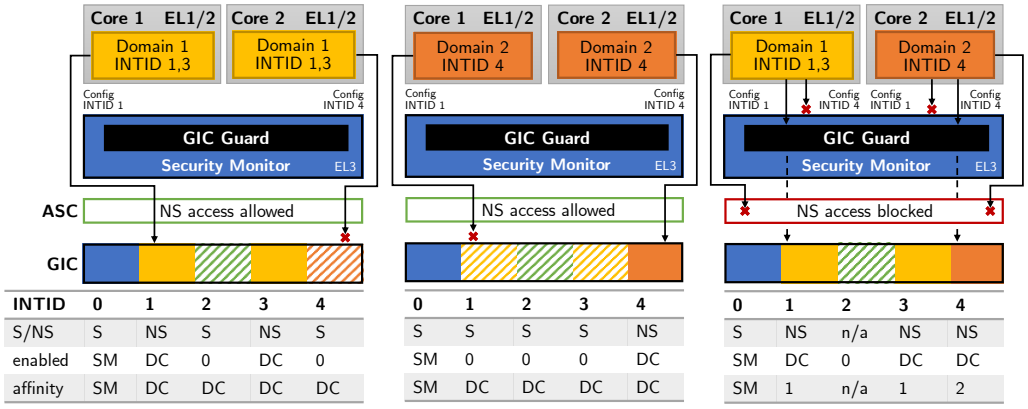
3.3 Device Isolation

In addition to the isolation of execution and memory, TEEtime supports assigning devices to domains, allowing them to access these devices directly, i.e., without the interference of intermediaries. We isolate device access using techniques similar to memory isolation: On Arm, devices are memory-mapped, and platforms are usually equipped with ASCs such as the Arm TrustZone Address Space Controller [Arm14] that also cover the device address space, or with vendor-specific components dedicated to performing access control to devices, such as the NXP Central Security Unit [NXP21]. As with memory isolation, under temporal sharing the memory-mapped address regions for device configuration and data are marked as non-secure only when the corresponding domain is executing. Under spatial sharing, access is configured such that only the core executing the owner domain can access. Figure 1 visualizes this setup.

In contrast to memory isolation, a challenge is that devices are discrete resources. While it is possible to slice memory into parts that fulfill the domain requirements, this cannot be done with devices: A device can usually not be divided into multiple parts. In TEEtime, devices can therefore only belong to one domain at a time.[1] Mapping one device to a single domain avoids issues concerning both security and functionality arising on legacy systems when two execution contexts access the same devices concurrently. As multiple domains might require access to the same device to fulfill their purpose in a realistic setup, we discuss how diverse use cases might still be achieved in Section 4 and showcase different case studies in Section 7.3.

Finally, many devices are interrupt-driven, meaning a domain needs to be able to configure, receive, and handle interrupts. This requires that the interrupt system can be shared securely between domains. We discuss our solutions for interrupt isolation below.

[1] The only exceptions are peripherals for which access is handled through system registers and interrupt handling is banked per core. An example of this is the Arm Generic Timer commonly used for scheduling.



INTID	0	1	2	3	4
S/NS	S	NS	S	NS	S
enabled	SM	DC	0	DC	0
affinity	SM	DC	DC	DC	DC

0	1	2	3	4
S	S	S	S	NS
SM	0	0	0	DC
SM	DC	DC	DC	DC

0	1	2	3	4
S	NS	n/a	NS	NS
SM	DC	0	DC	DC
SM	1	n/a	1	2

(a) Temporal: Domain 1 active **(b)** Temporal: Domain 2 active **(c)** Spatial

Figure 2: Continuation of the examples in Figure 1. (a) and (b) When a domain runs, its interrupts are marked as non-secure (NS), all other interrupts are marked as secure (S). The enabled state of its interrupts is domain-controlled (DC), while others are disabled (0) or controlled by the SM (SM). The affinity of interrupts is always controlled by their owner. (c) All interrupts owned by domains are configured as non-secure; their enabled state is domain-controlled, while others are disabled. The SM controls the affinity. In contrast to temporal, access control is not performed by the GIC. Instead, all direct accesses to the GIC by domains are blocked and domains can request GIC access through the GIC guard module in the SM, which decides whether the access is allowed.

3.4 Interrupt Isolation

Software interacting with devices usually relies on interrupts to be notified of certain events, such as when a device finishes an operation or a user inputs something. Therefore, for full device support, domains need to be able to control the interrupts of the devices they own. Furthermore, it is important that domains cannot observe or influence the interrupt state of devices of other domains. Observing interrupt state might be used to infer information about events that other domains should not be able to observe [DLLZ16]. Influencing interrupt state can be used to modify the control flow of other domains, e.g., by triggering an interrupt that the domain is currently not expecting and ready to receive, or by suppressing the routing of interrupts that a domain expects [SSK+24, SSBS24]. Therefore, we require a mechanism that partitions the GIC in a way such that a domain can configure, receive, and handle exactly the INTIDs it owns.

Requirements. To prevent domains from interacting with the interrupts of other domains while simultaneously preserving the functionality of the interrupts system, an interrupt isolation mechanism needs to fulfill the following requirements.

***R_{conf}*:** A domain can access the configuration of exactly the interrupts it owns.

***R_{state}*:** A domain can observe and change the state of exactly the interrupts it owns, i.e., set them pending, active, or retired.

***R_{route}*:** A pending and expected interrupt will trigger when its domain is executing and will only trigger within its domain.[2]

Strawman approach. One might think that the approach of using existing ASCs (as is used for isolation of memory and devices) can be applied to interrupts. However, this is not feasible because ASCs only support a coarse granularity of memory regions. While it is possible to allow or deny a domain access to all of the GIC's memory-mapped registers, this is insufficient, as it would either allow a domain to modify all INTIDs or prevent it from modifying its own. Compliance with R_{conf} and R_{state} requires bit-level granularity as GIC registers are tightly packed with up to 32 INTIDs sharing one register.

[2]Expected means that all requirements for the interrupt to trigger at the CPU interface are met. This includes that the interrupt has sufficient priority and is enabled.

Further, an approach that only enforces access control is not sufficient to satisfy R_{state} and R_{route}: The GIC cannot be aware of the domain context on the cores, and therefore, it would deliver interrupts at any time and to any core. Once an interrupt is signaled to a domain, this domain can handle it, regardless of whether it is its owner.

A different approach is to delegate the management of interrupts to a common trusted component. This is the approach taken by full-fledged hypervisor solutions. The GICv3 supports the virtualization of interrupts, which allows a hypervisor to transparently configure, emulate, and route interrupts to VMs. These mechanisms can also be used to achieve interrupt isolation among different VMs, as the hypervisor can manage the system such that each VM seems to have its own view of the interrupt system. However, such an approach would significantly increase the complexity and size of the TCB, as its tasks now do not only include enforcement of interrupt isolation but also full GIC virtualization.

Our approach. We extend the SM in EL3 with mechanisms to enforce access control to the GIC and correct routing on a per-INTID level. This has two advantages: the SM is aware of which domain is currently running on which core and software can provide bit-level access control. For temporal sharing, the SM ensures that only the INTIDs of the currently running domain are active, i.e., can be configured, routed, and handled, while all other INTIDs are inactive and protected. For spatial sharing, it ensures that INTIDs are only active for the cores that their domain is running on. Below, we describe the design of our mechanisms for temporal and spatial isolation to protect the configuration and routing of interrupts and argue that once an INTID is only delivered to the correct domain, the requirements for INTID handling are satisfied.

3.4.1 Temporal Interrupt Isolation

Under temporal sharing, exactly one domain is running on the system at a time. This means that exactly one domain's interrupts need to be active at a time. Instead of relying on mechanisms such as the TZASC, we leverage the features of the GIC that are designed for TrustZone: secure and non-secure groups of INTIDs.

Figures 2a and 2b visualize how we achieve temporal interrupt isolation through careful configuration of interrupt properties in the GIC. The SM marks the INTIDs of the running domain as non-secure and all other INTIDs as secure. Since the domain is running in non-secure state, it can access the configuration of its INTIDs but not of the INTIDs of other domains or the SM, thus fulfilling R_{conf}. To fulfill R_{route}, the GIC stores the enabled state of INTIDs of a domain that is preempted before disabling them. This prevents the domain's interrupts from triggering while another domain is running. However, our system must register the arrival of these interrupts, which are to be triggered when the domain is running again. Fortunately, if a peripheral asserts an interrupt, its interrupt becomes pending independently of its security/enabled state and stays pending until it is re-enabled, at which point it can trigger. Therefore, when a domain is resumed, the SM restores the previously recorded enabled state of its INTIDs. It is important to note that the currently active domain cannot 'steal' another domain's interrupts, as they are marked as secure, which protects their configuration. Interrupts can only be handled if they are enabled and pending. Since these conditions are only met for the interrupts of the currently active domain, no other interrupts can be handled. This property is guaranteed by the Arm specification [Arm24] and fulfills R_{state}.

3.4.2 Spatial Interrupt Isolation

Under spatial sharing TEEtime cannot achieve isolation by configuring interrupts as inactive as this is a global configuration and not applied per core. Doing so would either prevent a domain from legitimately configuring its interrupts or allow other concurrently running domains to break interrupt isolation, conflicting with R_{conf}. Further, the delivery

behavior would be that of secure interrupts—which would prevent them from being handled by a domain altogether, violating R_{state}. While one could solve this with hardware changes to the GIC, TEEtime avoids any such invasive changes and leverages the SM to do domain-based filtering as depicted in Figure 2c: Direct domain access to the GIC is blocked with an ASC. Instead, the SM offers an interface called the GIC guard that allows domains to request changes to the GIC configuration. The GIC guard performs valid requests on behalf of the domain: Reads and writes to the configuration of INTIDs owned by the domain are forwarded, while reads and writes to other INTIDs return zero or are ignored, respectively. This corresponds to the behavior when non-secure software tries to access secure interrupts [Arm22] and satisfies R_{conf}. While every configuration access now requires a more expensive SMC call (instead of one memory access), they are mostly performed during device boot. At runtime, interrupts are not frequently reconfigured, except in cases when a peripheral is hot-plugged or for load balancing.

To satisfy R_{route}, we can leverage the affinity feature of the GIC: The GIC guard ensures that the affinity of each INTID is set to a core that is currently running its owner domain. Further, it ensures that interrupts that are currently not owned by a domain are not delivered to any core by disabling them. E.g., in Figure 2c, the affinity of INTIDs 1 and 3, which are owned by domain 1, is set to core 1. Note that domains cannot change the affinity of their interrupts to invalid values, as the GIC guard would block this.

Once an interrupt is delivered to the correct core, software usually handles it through the CPU interface. This CPU interface is banked per core and can only be used to handle interrupts that were delivered. Therefore, no arbitration is needed to handle the interrupt. This satisfies R_{state}. Software can still decide to handle the interrupt through the GIC interface, but this would incur the GIC guard performance penalty.

3.4.3 Compatibility of Interrupt Isolation with Existing Software

Our interrupt isolation framework changes the way that software interacts with the GIC. This might break compatibility with existing software. For temporal isolation, this is not a concern: Access to the GIC is unchanged, apart from software not being able to access the configuration and state of INTIDs it does not own. For these INTIDs, the behavior is like that of secure interrupts. As long as the software is configured to only expect to control the interrupt it owns, it is compatible with our framework. This is the case for the Linux kernel, which is usually supplied with a device tree that specifies the devices and corresponding interrupts that are available, as well as for bare-metal apps that only control a subset of the devices and interrupts available. We confirm this in Section 7.

For spatial isolation, TEEtime changes are more significant as software cannot access the GIC directly anymore. This requires adapting existing software. One option is to adapt legacy software to interact with the GIC guard instead of accessing the GIC. This might incur significant engineering effort. Instead, we propose the introduction of a minimal compatibility layer that traps accesses to the GIC and forwards them to the GIC guard. We discuss this layer in more detail in Section 6.3. With such a layer, no additional changes to the OS, drivers, or apps are necessary.

4 Device Assignment

With the TEEtime device and interrupt isolation mechanisms, only one domain can have secure direct access to a peripheral at a time. Therefore, TEEtime supports different peripheral access modes to balance security, developer effort, and functionality requirements across domains. We demonstrate how different peripherals are used in appropriate modes with our prototype (cf. Section 7.3) and reason about security implications in Section 5.1.

Exclusive Access. Only one domain is allowed to access the peripheral during the platform's runtime. Every peripheral type supports exclusive access; no specific functionality apart from basic support for this peripheral needs to be implemented in software.
Multiplexing. Platforms can have multiple instances of the same peripheral type, each with its own memory and INTIDs (e.g., buttons, LEDs). Similarly, peripherals may natively support multiplexing, i.e., have different logical instances with separate memory regions and interrupt IDs (e.g., counters). TEEtime can exclusively assign individual instances to different domains, allowing them to profit from the same functionality. Furthermore, it is possible for some peripherals to split them into multiple functional groups, e.g., assigning different isolated parts of the display to domains or even the SM [PKS+21]. Support for this mode depends on the platform configuration.
Handover. For peripherals that support hot plugging, i.e., a software-based reset, TEEtime enables transferring ownership during the lifetime of domains. Such a transfer can happen with user interaction (e.g., button press) or by a domain yielding ownership and transferring it to a new domain. During handover, the SM ensures that device and interrupt isolation are updated correctly.
Read-only Access. Some devices only provide information and optionally generate interrupts when the information is updated (e.g., GPS, gyroscope). They require minimal and often one-time configuration. In such cases, TEEtime can allow one domain to perform configurations and receive interrupts. Additionally, it can selectively allow other domains to read device memory. This allows several domains to access the peripheral without corrupting its state. This approach might require adapting domains that are granted read access only, as they will not receive interrupts from the device and need to poll instead.
Proxy Access. The owner domain can decide to allow other domains to have proxy-based access to its peripherals. In this mode, TEEtime is not actively involved in enforcement. Instead, the owner offers an API to other domains via inter-domain communication. While this mode does not require specific peripheral properties, it requires software changes—in the owner domain to offer a peripheral-specific API and in the requesting domains to use the peripheral through the API instead of directly interacting with it. Different approaches can be taken to implement the API, e.g., the owner emulating the peripheral interface or offering a high-level API that abstracts the peripheral's major operations. The interrupts are still routed only to the owner domain, which can decide to expose interrupt configuration and/or information.

5 Security Analysis

This section presents a security analysis of TEEtime. We analyze memory isolation, peripheral isolation, and interrupt isolation separately. We distinguish between two distinct adversarial settings: an attacker running concurrently to a domain on a different core and an adversary running just before and after the domain on the same core. These settings correspond to our threat model, where an attacker has control over one domain, and our setups of temporal and spatial platform sharing. As the attacker is running in a domain, it is executing in non-secure state. The attacker cannot access the data or code of the SM, as they are stored in secure memory. Therefore, the attacker cannot modify the domain configuration information of the SM.

5.1 Memory and Devices

To achieve memory and device isolation, we leverage address space protection based on ASCs as described in Section 3.2. The ASC enforces access permissions for a limited set of contiguous memory regions or device address ranges according to the source of the access request, i.e., the core id, and its security state. It guarantees that no memory or device

access that violates the configured access control policy will succeed. The ASC also only accepts configuration changes from the SM.

Assuming that the configuration setup within the SM is correct, an attacker who runs before and after a victim domain cannot access the victim's memory or devices. This is because the corresponding address ranges are always marked as inaccessible from non-secure state while the attacker is running. When the attacker is running concurrently to the victim, the victim's address ranges must be accessible for non-secure software. However, as access is restricted to cores executing the rightful domain, accesses stemming from within the attacker's domain are blocked.

Different device assignment modes introduce specific security considerations. *Exclusive mode* introduces no additional risks beyond correct initial configuration and enforcement by the SM, as no other domain can access the peripheral or its interrupts at any point. In *multiplexing mode*, the same security considerations as for exclusive access hold as long as the different instances of the peripheral are properly isolated. However, shared hardware (e.g., buses or power lines) could allow for side-channel leakage. In *handover mode*, incorrect management of devices could lead to device hijacking and state leakage. We avoid this by having the SM enforce that only the specified recipient of a device can claim it and by ensuring that domains clear residual state before handing them over. Finally, it might be important for a user to know which domain currently owns a peripheral. Otherwise, a malicious domain could spoof the interface of another domain. Such dynamic ownership information can, e.g., be communicated through LEDs controlled by the SM [SSW+15]. In *read-only mode*, domains have to trust the owner not to tamper with the configuration—while it might be possible to verify it, they cannot intervene in case of misconfiguration. However, any information that is read from the peripheral can be considered authentic, as access is direct. *Proxy mode* inherently entails reduced security guarantees; either the proxying domain must be trusted (e.g., an attested dedicated network-proxying domain), or the use case must be able to tolerate untrusted intermediary access (e.g., end-to-end encrypted communication). At the same time, the owner needs proper in- and output techniques to protect itself and the peripheral from a misbehaving domain.

5.2 Interrupts

In Section 3.4, we list three guarantees that need to hold for secure interrupt isolation: Only the domain owning an INTID can access its configuration (R_{conf}) or its state (R_{state}), and interrupts are routed only to the domain owning it (R_{route}).

For an adversary running before and after the victim domain, we consider the temporal TEEtime interrupt isolation mechanism. The SM marks only interrupts that belong to the currently running domain as non-secure. Consequently, an attacker cannot modify the configuration (R_{conf}) or state (R_{state}) of the victim domain. Additionally, the SM marks all interrupts belonging to domains that are currently not active as disabled. As only one domain is active at a time, this means that an interrupt can never trigger while a domain other than its owning domain is running (R_{route}).

For a concurrently running adversary, we discuss the spatial isolation approach. When multiple domains are running on the platform at the same time, the SM marks the GIC as inaccessible. This prohibits all domains, including the attacker's, from directly accessing the GIC interface. Instead, the GIC guard in the SM brokers access to the GIC. The GIC guard allows or blocks access to interrupt configuration depending on the domain that requested the access (R_{conf}). In addition, the GIC guard does not allow a domain to set the affinity value of an interrupt to a core that is outside the domain (R_{route}). Interrupt states are managed through system registers, which allow each core to handle interrupts that are delivered to it. A core cannot access another core's system registers. Consequently, a core cannot change the state of interrupts routed to other cores (R_{state}).

6 Implementation

To implement TEEtime, we implement two modules that extend the Trusted-Firmware A, the reference implementation for the secure monitor provided by Arm [Lin]: The domain dispatcher (Section 6.1), which is the implementation of our baseline for the isolation of different execution contexts in a non-secure state and facilitates their lifecycle, and the GIC guard (Section 6.2), which is in charge of enforcing interrupt isolation.

6.1 Domain Dispatcher

We develop the domain dispatcher as the foundation for developing our device and interrupt sharing mechanism. The purpose of this dispatcher, which is running in EL3 as part of the Trusted Firmware, is to support the lifecycle of independent coexisting software stacks in non-secure EL2 to EL0. For this, the domain dispatcher implements functionality to isolate execution and to support the lifecycle of domains.

6.1.1 Isolated Execution

The domain dispatcher maintains several data structures per domain: (1) The domain descriptor, which contains static information about the domain, such as its identifier, or whether it is supposed to run under spatial or temporal isolation. (2) The domain context, one per domain and core, which the general purpose registers, the processor state, and system registers such as the program counter, the exception handler base address, and translation table base addresses of EL2 through EL0. The domain dispatcher isolates the execution of different domains by storing domain state in these structs when a domain is preempted or yields and by restoring the state when the domain is dispatched again. To implement these context switches, we can reuse functionality already present in the TF-A: Processor state is saved and restored by the context management module when switches between the non-secure and secure state occur; we can reuse the corresponding struct definitions and functions.

The domain dispatcher provides an interface that can be used by the scheduling domain to request the creation and execution of a new domain. As the scheduling domain is untrusted, the domain dispatcher verifies the validity of this request and then performs the requested action. Our domain dispatcher implementation supports two modes of domain execution: Spatial and temporal. In spatial execution, the domain requesting the new domains to be run yields a core, on which the new domain is then dispatched. The new domain owns and controls this core until it yields. In temporal execution, the domain requesting the new domain to be run yields the whole system and passes a cycle counter value to the dispatcher, which specifies for how long the scheduled domain is to be run.

6.1.2 Domain Lifecycle

On device boot, the domain dispatcher initializes the data structures for managing domains; the first domain to run is the scheduling domain. The system then continues to boot normally into the non-secure world.

Creation. After loading the new domain's binary into memory, the scheduling domain can request the creation of a new domain through an SMC call directed at the domain dispatcher by specifying the entry point of the binary and the resources that are assigned to it, e.g., memory regions and interrupt IDs. The domain dispatcher assigns this domain a free ID and measures the domain setup. This measurement can later be used to, e.g., perform remote attestation or create domain-specific keys. However, this is out of the scope of this work. The domain dispatcher sets up the domain descriptor and initializes a fresh domain state. The domain is now ready to be dispatched.

Dispatchment. The scheduling domain requests the dispatchment of a domain through an SMC call. The domain dispatcher then saves the state of the currently running domain, restores the state of the domain to be dispatched, and starts execution. In the case of spatial isolation, the new domain can run on the assigned core until it yields. For temporal isolation, the domain runs for a specified number of cycles before the domain dispatcher returns control to the scheduling domain. This is achieved by the domain dispatcher setting a secure timer interrupt that will trigger after the specified number of cycles as an FIQ that redirects the control flow to the domain dispatcher.
Preemption. When a domain yields or is preempted, the domain dispatcher saves its state and restores the state of the scheduling domain. The scheduling domain can then decide which domain to run next, or not to run a domain and continue executing itself.

6.2 GIC guard

Our GIC guard consists of two logical components: The lifecycle manager invoked by the domain module on domain lifecycle events and the GIC configuration interface that domains can explicitly invoke through an SMC call to request GIC configuration changes. The GIC guard relies on information maintained by the domain dispatcher on the state of the system, i.e., which domain is currently active on a core or about to be executed. The GIC guard implementation extends the domain descriptor with an interrupt ownership field that stores which interrupts a domain owns. The domain context is extended with information about the GIC state, which includes information about the CPU interface state and which of the domain's interrupts were enabled when the domain was preempted. Further, the GIC guard stores which interrupts are owned by a domain.

On domain creation, the scheduling domain also specifies which interrupts the new domain owns. The GIC guard ensures that none of the interrupts to be assigned are already owned by a domain or are interrupts that belong to EL3/the secure state (notably, interrupts that belong to secure resources such as the secure timer). The GIC guard then populates the new domain's interrupt permission field and ensures that all interrupts of the new domain are disabled as this is the state that software expects (so that it can set up interrupt handlers and configure interrupts before enabling interrupts).

When a domain is about to be run, the GIC guard acts differently depending on whether the domain runs spatially or temporally isolated. For spatial isolation, the GIC guard ensures that the domain's INTIDs are configured as non-secure and sets their affinity to the core the domain is scheduled to run on. Further, if not already in place, it sets up the access control prohibiting non-secure accesses to the GIC. For temporal isolation, the GIC guard first sets *all* interrupts to disabled, ensuring that the old domain's interrupts do not trigger until it runs again. Then, it applies the security configuration for the new domain. This entails setting this domain's INTIDs to non-secure while all others will be set to secure. Finally, the dis-/enabled-configuration for this domain is restored, meaning that the INTIDs that were enabled when the domain stopped executing the last time are set to enabled, while all others remain disabled.

Once a domain is preempted or yields, the SM stores the current state of its GIC configuration—including its CPU interface state, and the INTID-specific state—as part of the domain state. When a domain yields, the SM disables and frees the INTIDs owned by that domain. They can now be assigned to other domains. We implement most of the checks and configuration accesses as bitwise operations for up to 32 INTIDs at a time. This reduces the number of memory accesses and makes the setup time for the GIC configuration independent from the number of INTIDs that need to be reconfigured.
Explicit GIC configuration. The GIC guard module is used by domains to configure the GIC when direct access to it is blocked. This is always the case when spatially isolated domains run on the platform. Domains can invoke this module through an SMC call. The parameters of this access are the memory address to be accessed, whether the access is a

read or write access, and, in the case of a write access, the value to be written as input. Using such an interface instead of one with semantic information (e.g., set interrupt ID 37 to enabled) allows for straightforward processing in the GIC guard and mirrors how software currently accesses the GIC. From the shared state with the domain dispatcher, the GIC guard can infer which domain made the request and validate its interrupt ownership.

Based on the access permissions and the address of the register being accessed, the GIC guard then assembles a bitmask, where bits that are allowed to be accessed are set to one. This is necessary as registers are shared between interrupts, e.g., the interrupt-enabled registers are 32-bit and hold the state of 32 interrupts. For example, when a domain owns interrupt IDs 1 and 4, its ownership mask would be 0...010010. For accessing enable registers this mask can be used directly, as these registers use 1 bit. This differs for priority registers: Here, 4 bits are used per interrupt. Accordingly, the GIC guard would extend the bitmask to 0...01111000000011110000. For write access, this mask, together with the old value of the register, is then used to compute the new value. For read access, the mask is used to set inaccessible bits to zero to not leak information. Ignoring writes and returning zero is on par with specified GIC behavior for unauthorized accesses [Arm24].

Special attention needs to be put to three details. First, the GIC guard is operating in secure state and has to perform secure accesses to the GIC, as otherwise, the access would be blocked by the ASC. This requires careful evaluation of some registers where the GIC presents different views to secure and non-secure accesses, e.g., the Distributor Control Register. Furthermore, we must ensure that domains cannot allocate INTIDs to each other, maliciously or accidentally as this would violate our requirements for interrupt isolation. Therefore, the GIC guard ignores requests to route interrupts outside the domain, e.g., when a domain wants to turn off affinity routing or set the affinity of a core that it does not own. Third, as domains can request the GIC guard service at any time, the GIC guard needs to synchronize write access across cores to avoid race conditions.

6.3 Compatibility layer

Existing OSes and services assume they have direct access to GIC and configure it via reads and writes, which is not possible under spatial sharing. An access would cause an asynchronous abort that is routed to the SM. In an ideal setup, the SM could then reconstruct the attempted access and perform it on behalf of the domain, if allowed. Unfortunately, execution may not block after the GIC access and proceed immediately after dispatching a transaction on the interconnect without waiting for confirmation. As a result, the abort arrives a few instructions later. This makes it challenging to reconstruct the faulting instruction, as the state of involved registers might have changed already and the abort might be routed to a core that it did not originate from.

Retrofitting existing code with explicit SMCs instead of accessing GIC memory might be labor-intensive or impossible for large code bases. Instead, we provide a minimal EL2 layer with TEEtime that can be placed between legacy software running in EL1 and the SM. It uses *synchronous aborts* to reconstruct memory accesses, which trigger immediately and provide precise information about the fault. This layer consists of two parts: (1) stage-2 translation tables that map the whole address space in a 1-to-1 mapping with the most permissive access attributes (effectively making the stage 1 translation tables the main decider) apart from the GIC address space, which is marked as non-accessible. (2) A minimal exception handler runs once software tries to access the GIC. This handler reconstructs the attempted GIC access from the abort syndrome registers and forwards the request to the SM. It is important to note that this EL2 module is part of a domain—it does not perform a security function and is not part of the shared TCB.

7 Evaluation

We evaluate the impact of our TEEtime implementation on TCB size and software performance. To demonstrate TEEtime's feasibility and different peripheral sharing modes, we implemented various case studies, including a Signal messenger running in a domain on a Purism Librem 5 phone.

Setup. We deploy our TEEtime implementation on the Purism Librem 5, a smartphone based on the NXP i.MX 8MQuad processor with 4 Arm Cortex-A53 cores running at 1.5 GHz with 3 GB of RAM. Notable peripherals include a touchscreen, a baseband module, and a UART port. Most importantly, the phone is completely unlocked, which allows us to deploy our version of the TF-A in EL3. The phone is equipped with an Arm CoreLink TZC-380 TrustZone Address Space Controller [Arm10] for memory protection and an NXP Central Security Unit [NXP21] (CSU) for peripheral access control. While the CSU allows enforcement based on different identifiers, the i.MX 8MQuad processor assigns the same identifier ID to all cores. On the Librem, we evaluate performance impact and demonstrate a realistic use case: A Signal messenger running in a dedicated domain.

We also implemented TEEtime for the Arm Fixed Virtual Platform (FVP), an Armv8-A simulator. This platform allows for reliable and fast development and debugging; we use it to show a TEEtime deployment with diverse interrupt and peripheral assignments under both spatial and temporal sharing. We give a short overview of the FVP case studies together with the peripherals and peripheral sharing modes in Section 7.3.

Lines of Code (LoC)in TCB. Our implementation is based on the TF-A v2.8, which comprises 427229 LoC. We add 1575 LoC to implement TEEtime: 1047 for the baseline system, 39 for memory isolation, 248 for device isolation, and 241 for interrupt isolation.

7.1 Domains

To test the functional soundness of domains running on TEEtime, we deploy a stripped-down version of the Linux kernel in the app domains, i.e., domains that are not the scheduling domain. The stripped-down versions of these kernels are supplied with a device tree that contains only the platform resources that are assigned to the domain and run with an InitramFS based on Busybox [Vla].[3]

As the scheduling domain and first domain to boot, we run the original OS of the Librem: the Debian-based PureOS (kernel version 6.4.16). We wrote two kernel modules for the scheduling domain performing the necessary SMC calls to manage domains in temporal and spatial mode (126 and 136 LoC), as well as two user space applications (170 and 176 LoC), to set up and run a domain based on a provided binary and configuration.

7.2 Librem Phone: Runtime Overhead

We evaluate the performance of TEEtime based on the following criteria:

1. *Domain Lifecycle Overhead*: We measure the overhead of memory, device, and interrupt isolation on the domain lifecycle and compare base operations to existing systems.
2. *Interrupt delivery*: We measured TEEtime interrupt delivery latency and frequency.
3. *Peripheral performance*: We benchmark the performance of the network, disk, and display under TEEtime.
4. *System benchmarks*: We measure the impact of TEEtime on system operations.
5. *Application benchmarks*: We measure the overhead of TEEtime on different user-space applications and compare it to existing work.

[3] Note that it is not a requirement to deploy a kernel in app domains, it is also possible to deploy a minimal runtime or a bare-metal app, depending on preference about functionality and domain TCB.

Table 1: Domain lifecycle operations. Average runtime reported in µs. We stepwise add memory (M), device (D), and interrupt (I) protection to the base system. Fields with – mark operations that do not apply for the respective setup.

	Spatial				Temporal			
Config:	B	BM	BMD	BMDI	B	BM	BMD	BMDI
setup	3016	3017	3018	3017	3104	3102	3107	3108
device	–	–	–	0.36	–	–	–	0.36
interrupt	–	–	–	0.60	–	–	–	0.48
run	487	490	493	500	2	8	30	39
preempt	–	–	–	–	2	8	30	38
yield	0.12	1	6	12	1	4	28	44

7.2.1 Domain Lifecycle Overhead

We benchmark the operations of TEEtime: setting up a domain, assigning devices and interrupts to a domain, running a domain, a domain yielding execution, and preemption for temporal domains. The baseline is without any configured isolation; we subsequently add the configuration of memory, device, and interrupt isolation to understand the relative contribution of each. For memory and devices, we configure the ASC, but do not enable the enforcement, as the Librem 5 does not support per-core enforcement. The results are summarized in Table 1.

We observe no significant differences in domain setup times and the assignment of interrupts and devices, as these operations are agnostic of the configuration.[4] Running a spatial domain takes longer due to the domain target core being rebooted, while for temporal domains, only a new CPU state is populated. Note that running a spatial domain is a much rarer operation, as the domain runs until it yields, while a temporal domain is continuously preempted. We observe that the different isolation configurations increasingly impact the run time of the run, preempt, and yield operations due to the memory/bus operations to configure the TZASC, CSU, and GIC.

We compare TEEtime's operation runtimes against TrustICE [SSW+15] and Shelter [ZHN+23] for temporal and against Sanctuary [BGJ+19] for spatial domains. Although this comparison can only offer intuition (due to varying platforms and slightly different lifecycle operations), it aids in interpreting the numbers we report for TEEtime. For consistency, we exclude the time for domain code verification from all reported numbers.

TrustICE, measured on a 1 GHz Cortex-A8 processor, reports approximately 32 µs for configuring and entering ("run") and 40 µs for exiting ("preempt") its protected environment; both operations include access to the GIC to disable interrupts. These numbers are very similar to our measurements for a temporal setup in full isolation configuration (39 and 38 µs). Considering our slightly higher CPU frequency, TEEtime seems to introduce slightly more delay, which we attribute to our more complex logic (keeping track of interrupt ownership, protecting interrupts of all non-active domains, setting up device protection). Shelter, measured on a Juno R2 board (Cortex-A72/A54, 1.2 GHz/950 MHz) retrofitted to approximate CCA-enabled hardware, reports a 3359 µs creation time (excluding verification) for a protected app (Sapp) and 3 µs for switches between an Sapp and the Host OS. These operations include setting up the configuration of memory isolation (no configuration for devices or interrupts) and are in the same order of magnitude as our operations in the configuration with memory isolation. For spatial domains, Sanctuary's performance was measured on the HiKey 960 (Cortex-A73, 3.2 GHz) and reported, among others, in a "lock and verify" (13 ms) and an "early core initialization" (37 ms) step. These steps correspond roughly to our "setup" and "run" operations. The time it takes to verify the domain code is not reported separately; however, it seems that their numbers reported are larger than our measurements, which are roughly 3.5 ms for all spatial configurations.

[4] The domain setup numbers presented were achieved without measurement of the domain binary, as this would add runtime depending on the binary size.

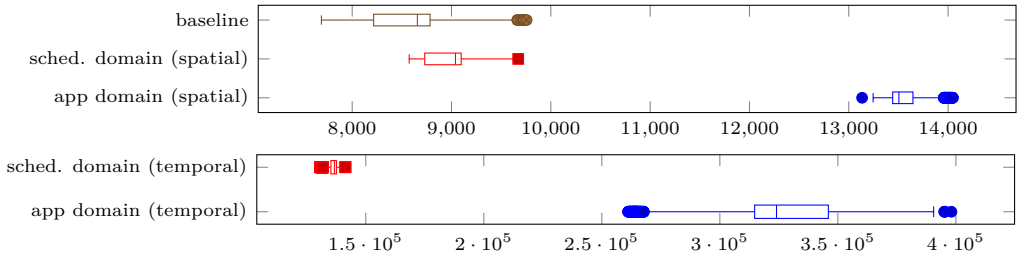


Figure 3: Distribution of interrupt delay for scheduling ticks in counter cycles (counter at 8.33 MHz). The same baseline applies to temporal and spatial. Note the different scaling for temporal and spatial.

In summary, TEEtime provides device and interrupt isolation without disproportionately increasing the runtime of lifecycle operations, also compared to related work.

7.2.2 Interrupt delivery

We use timer interrupts to evaluate the impact of TEEtime on interrupt delivery: When software handles a timer interrupt, it can know exactly when the interrupt was asserted, as this happens exactly at the moment when the comparator value (deadline previously set by software) is equal to the counter value. Other peripherals assert interrupts asynchronously and no exact information about when the interrupt was asserted is available.

We evaluate the timer interrupt behavior in two ways: First, we instrument the kernel scheduler to compare the comparator value and the current counter value at every scheduler tick. As scheduling ticks occur at very high frequency, we record the cumulative difference over 1000 scheduling ticks, a tradeoff between granularity and interference with normal kernel operations. The results are visualized in Figure 3. The baseline (when only the scheduling domain is running) is a median delay of 8 counter ticks from interrupt assertion to handling of the interrupt. When running a spatial domain, the behavior remains very similar, both in the scheduling and in the app domain. This is because both domains are running and able to service interrupts at any time. Running a temporal domain has a more significant effect: The median delay for the scheduling domain is 136, which is a result of the time required for the domain context switch if the timer triggered during the execution of the app domain. In the app domain, this value is 324 and shows higher variability, a result of the domain being scheduled by the scheduling domain. In scenarios where very low interrupt latency is crucial, we suggest spatial sharing.

Second, we use the Linux high-resolution timer to fire 10000 interrupts in intervals of 0.5 ms and measure their arrival variance. In the baseline, the mean difference is 6437 ns, with a standard deviation of 88391 ns, showing good accuracy in most cases, but hints that there are longer phases in which the core is in a non-interruptable state. In a spatial domain, it is 5119 ns and 3346 ns, for the temporal 0.3 ms and 1.2 ms, respectively.

7.2.3 Peripheral Performance

We benchmark three peripherals in the sharing mode used for our Signal messenger domain: To proxy the network, the scheduling domain forwards network messages from and to the app domain through a shared memory interface using two ringbuffers. Each ringbuffer consists of 25 entries that can contain up to 2000 bytes, amounting to 100 KB in total. To benchmark throughput and latency, we run a local server serving a 4 KB file once on the phone itself and once in a local wireless network, and run ApacheBench [Apa] with 100 requests. With the server on the phone, we observe a median time of 2.8 ms per request served and an overall throughput of 1469 KB/s in the scheduling domain; for spatial, these values are 3 ms and 1366 KB/s and for temporal, 34 ms and 122 KB/s. With the server

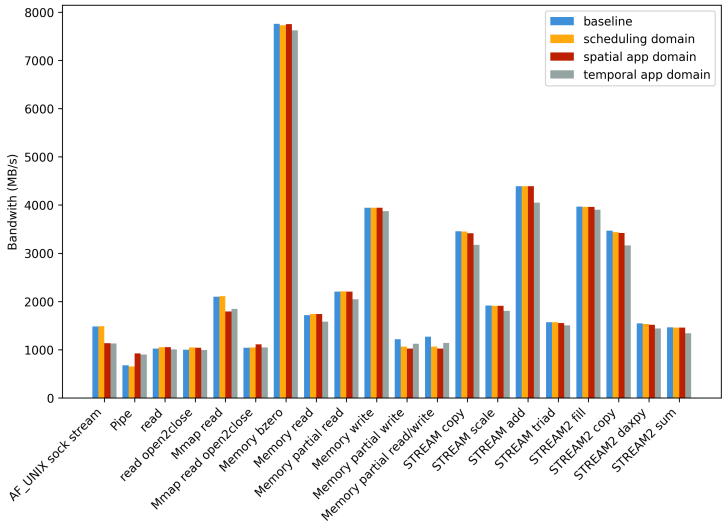


Figure 4: LMbench results for TEEtime.

Table 2: Comparison of the overhead of TEEtime temporal and spatial domains application in comparison to Shelter enclaves.

Application	Temporal	Spatial	Shelter
AES	12 %	2.1 %	5.2 %
OTP	21 %	5.6 %	0 %
LeNet	15 %	1.8 %	0 %
Memcached	2.8 %	1.9 %	8.3 %
Apache	384 %	37 %	15 %
Nginx	373 %	29 %	11.8 %

on the local network, these values are 19 ms and 24 KB/s for the scheduling domain, 21 ms and 21 KB/s for spatial, and 37 ms and 12 KB/s for temporal, respectively. To proxy disk access, we use an NFS-based implementation via the network tunnel and measure the throughput achieved by dd [dd]. As a baseline, we first measure throughput for direct disk access in the owner domain, which reaches up to 30 MB/s for read accesses and up to 18 MB/s for write accesses. In the spatial domain, we achieve 11 MB/s and 7.5 MB/s, and in the temporal domain 622 KB/s and 713 KB/s, respectively.

Finally, we measure the time it takes to handover the display, which is achieved by registering the hotplug mechanism for the touchscreen as an action for pressing the lower volume button. This removes the display gracefully from the current owner domain and attaches it to the next domain. For spatial, handing over the display from scheduling to app domain takes 1.25 s (roughly 10e6 counter cycles) and 0.14 s (1.2e6) for the other way. For temporal, it is 1.48 s (12e6) and 0.30s (2.5e6), respectively. As we did implement handover for network on the FVP, we report the instruction counts for these handovers: From scheduling to spatial app domain, we count roughly 36e6 instructions, and 31e6 back, for temporal 49e6 and 33e6, respectively.

7.2.4 System Benchmarks

We use LMbench [MS96] to measure the overhead of TEEtime on kernel-level operations, such as memory accesses and inter-process communication. This benchmark helps us to evaluate the overhead of our isolation mechanisms on system operations independent of any specific application. We ran four configurations: in a baseline system without any of our changes, in the scheduling domain, in a spatially isolated app domain, and in a temporally

Table 3: FVP Case Study Summary. Column 2 shows the peripherals and their access mode. Column 3 shows the lines of code of apps running within the domain. Column 4 depicts the number of peripheral INTIDs the domains are assigned ownership of at some point. H: Handover, M: Multiplexing, P_L: Proxy in scheduling domain, P_A: Proxy in app domain for storage (Stor), UART, network (NW), display (Disp), and mouse and keyboard (M&K).

Case-study	Peripheral Modes	LoC	INTID#
Disk	Stor (P_S), UART (M)	873	1
VPN	NW (P_A), UART (M)	2,142	2
Browser	NW (P_S), Disp (H), M&K (H)	709,921	2
Chat	Stor (P_S), NW (P_S), Disp (H), M&K (H)	637,036	2

isolated app domain. Note that the scheduling domain runtime is coreutils [MEB+], while other domains run with Busybox [Vla]. The average performance of LMbench in the scheduling domain compared to the baseline is 98.8%. Spatial and temporal app domains perform with 98.6% and 95.4%, respectively. Detailed results of the LMBench benchmarks can be found in Figure 4. We attribute the better performance of pipe in the temporal and spatial domain to the differences between coreutils and Busybox.

7.2.5 Application Benchmarks

In addition to the system benchmarks, we measured the end-to-end runtime overhead of TEEtime domains on various user-level applications to estimate the impact of TEEtime on real workloads. Each application was run in a baseline (Rich OS) environment and within a TEEtime domain. To provide a direct comparison to related work, we selected memory-bound, CPU-bound, and IO-heavy workloads evaluated by Shelter. TrustICE and Sanctuary benchmarked custom-built applications that are unavailable to us.

The results are detailed in Table 2. For compute-bound applications like OTP, AES, and LeNet, TEEtime introduces relatively modest overheads. Spatial domains, in particular, show minimal impact, partly outperforming Shelter's reported overhead. Memcached, a memory-intensive application, also shows low overhead. However, network applications like Apache and Nginx show significantly higher overhead compared to Shelter. This elevated overhead is due to our Librem implementation relying on proxy mode for network, which represents a worst-case scenario for performance. For temporal domains, this is further exacerbated by frequent preemptions, inhibiting communication between the domains. Overall, spatial domains show low overhead, while temporal domains show higher overhead, which can be attributed to the periodic preemption of these domains.

7.3 Domain Case Studies

To demonstrate the feasibility and usability of TEEtime, we implemented a working Signal client running in a domain on the Purism Librem 5 phone. The application allows users to sync their contacts and send and receive messages. The display is handed over to the domain, while network and storage are being proxied by the host OS. We provide a video demonstration at `https://youtu.be/FUUax5m5pqY`.

Additionally, we developed several app domains for the FVP to showcase the combination of different interrupt and device assignment scenarios and peripheral access modes. Table 3 shows the app LoCs, along with a summary of peripherals, modes, and the number of distinct INTIDs used in each case study.

For our **disk** domain, the scheduling domain provides the disk as a proxy to the app domain, which encrypts and MACs any data before sending it to the scheduling domain. This domain also showcases multiplexing: Multiple UARTs are available, so we assign one to the app domain, while one is used for the scheduling domain. Our **VPN** domain

allows other domains (including the scheduler) to access the network in proxy mode. This can, e.g., be useful if a trusted network domain is required, or all network traffic on the platform needs processing, e.g, tunnelling it through a VPN. We port ProxyLite [Jes] for transparent tunnelling, requiring no changes to the existing kernel and apps. Our **browser** domain enables secure web browsing. Although the scheduling domain functions as network proxy, sensitive information remains protected because connections are terminated and session keys and cryptographic operations are handled directly within the app domain. We run the GUI-based web browser Dillo [Cid]. Display, mouse, and keyboard are handed over from the scheduling domain, ensuring confidentiality of any direct user IO with the domain. An LED controlled by the SM indicates current peripheral ownership. Our custom GUI-based **chat** domain combines disk and network proxy access with display, mouse, and keyboard handover: It persists encrypted message history and contacts from disk and exchanges encrypted messages with our local server. Again, direct user IO is protected, as the respective devices are handed over with an LED indicating current ownership. A video demonstration is available at https://youtu.be/z6h8vdsYV8E.

8 Related Work

Isolated Execution on Arm. Traditional isolated execution on Arm leverages the secure world. One stream of work focuses on restricting the privilege of trusted apps by relying on hypervisors in the secure world [Goob], or by deploying ASCs and tightly locking down the configuration [CMSP22]. However, deployment in the secure state is rather tricky as the manufacturers usually restrict the secure state to only their software. Therefore, a growing body of work focuses on deploying sensitive workloads in a protected way in the non-secure state instead. TrustICE [SSW+15] proposes a model where sensitive apps can be executed in non-secure state, their memory protected from access by the host OS by processor watermarking mechanisms. However, their design ignores the issue of overpriviligisation and does not protect the host OS from access by the sensitive app. Further, apps can only be run in a temporal isolation setup. While they provide a case study where a user can interact with the display, no protection mechanism for the display and its interrupts is mentioned. The closest ARM-based design to ours in terms of memory isolation is Sanctuary [BGJ+19], which uses similar techniques to run isolated enclaves in non-secure state and does support spatial isolation. However, it does not support peripherals or their interrupts for sensitive applications. Recently, Shelter [ZHN+23] proposes to use the granule protection tables introduced with Arm Realms to isolate workloads in non-secure states. In contrast to our and the aforementioned work, access control is not enforced memory-side but at the core itself. Shelter does not discuss the isolation of devices, but their general design is compatible with TEEtime.

Virtualization-based approaches rely on a hypervisor in EL2 that virtualizes the hardware resources for virtual machines running in EL1. pKVM [Gooa], Hafnium [Goob], and Gunyah [Cen] are some of the hypervisors targeting the Arm architecture. Virtualizing peripherals is delegated to the host OS in all three hypervisors, which means that while the host OS cannot inspect the memory of other virtual machines, it observes and may manipulate any communication with peripherals, including interrupt handling.

Arm Realms. Arm CCA [Arm22] is a new Arm feature that allows launching VMs in a separate realm mode, which is not accessible from other modes. However, while direct and secure device access of the realm state has been added in recent iterations of the specification, interrupts are still received and handled by the untrusted host OS. Recent works [SBS+24, WZD+24, SLZK25] show how peripherals can be securely connected to realms, but do not secure interrupt handling. Devlore [BSG+24] proposes a framework in which the SM monitors interrupt arrivals to ensure that interrupts are not injected incorrectly by the host into VMs. It does not support the isolation of normal world

enclaves, and VMs need to rely on the realm mode manager for interrupt management.
Peripherals in Secure World and Minimization of Device Drivers. Previous work [PKS+21, PL22, GL22, DWY+22, SSWJ15, HJ23] proposes systems in which trusted apps running in the secure world have direct access to various peripherals. However, these approaches do not generalize to isolated environments in the non-secure state. A further hurdle for deployments with device access is the lack of device drivers for secure state software, as simply shifting existing drivers is impossible due to the missing operating system and inopportune due to their big code footprint; significant engineering effort is required to port existing drivers to the restricted TEE environment. There has been research on automating this process [GL22] or sandboxing reused drivers that incur a large TCB [YLL+24]. This is a complementary problem to our work, as one can reuse such approaches for porting device drivers to TEEtime domains if minimal TCB with a domain is required. On the other hand, TEEtime offers a viable alternative to such approaches if minimizing the TCB within one domain is not a priority, as shifting device drivers is possible. SeCloak [LSDB18] proposes a mechanism to restrict non-secure software from accessing certain peripherals. It leverages an ASC and deploys a minimal secure kernel that configures access policies, traps and emulates peripheral access, and forwards interrupts. However, this enforcement is the same for all non-secure software. TEEtime enforces peripheral access for multiple domains in the non-secure state.
Interrupt-based attacks. [DLLZ16] shows how sensitive information can be derived based on the arrival of dexvice interrupts. TEEtime closes this attack vector, as an untrusted domain cannot observe the arrival of another domain's interrupts. [SSBS24, SSK+24, LML21] show how malicious interrupt injections can compromise confidential VMs. TEEtime mitigates this as interrupts can only be asserted by the corresponding peripheral or from within the domain.

9 Conclusion

We design, implement, and evaluate TEEtime, a framework that extends isolated execution environments with direct and secure access to devices and their interrupts. We leverage address space controllers to isolate memory and device access and develop a novel approach based on existing primitives for isolating interrupt access. We develop a prototype of TEEtime, confirming that it does not inhibit the functionality of existing software stacks.

Acknowledgments

We would like to thank the reviewers for their valuable feedback and comments. This work was partially supported by the Zurich Information Security and Privacy Center (ZISC).

References

[ABPM21] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1357–1372. ACM, 2021.

[And] Android Open Source Project. Trusty TEE. https://source.android.com/docs/security/features/trusty.

[Apa] Apache. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[App20] Apple. Apple and Google partner on COVID-19 contact tracing technology, Apr 2020. https://www.apple.com/newsroom/2020/04/apple-and-google-partner-on-covid-19-contact-tracing-technology.

[Arma] Arm. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture/.

[Armb] Arm. Fixed Virtual Platforms. https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms.

[Arm10] Arm. CoreLink TrustZone Address Space Controller TZC-380 technical reference manual. https://developer.arm.com/documentation/ddi0431/c/, 2010.

[Arm14] Arm. ARM CoreLink TZC-400 TrustZone Address Space Controller technical reference manual. https://developer.arm.com/documentation/ddi0504/c/, 2014.

[Arm22] Arm. Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A. https://developer.arm.com/documentation/ddi0615, 2022.

[Arm24] Arm. Arm Generic Interrupt Controller Architecture Specification. https://developer.arm.com/documentation/ihi0069/hb, 2024.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[BGJ+19] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with user-space enclaves. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[BSG+24] Andrin Bertschi, Supraja Sridhara, Friederike Groschupp, Mark Kuhne, Benedict Schlüter, Clément Thorens, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. Devlore: Extending Arm CCA to integrated devices a journey beyond memory to interrupt isolation. *arXiv preprint arXiv:2408.05835*, 2024.

[Cen] Qualcomm Innovation Center. Gunyah Hypervisor. https://github.com/quic/gunyah-hypervisor.

[Cid] Jorge Arellano Cid. Dillo. https://www.dillo.org/.

[CMSP22] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ReZone: Disarming TrustZone with TEE privilege reduction. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2261–2279. USENIX Association, 2022.

[CSFP20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1416–1432. IEEE, 2020.

[dd] dd(1) — linux manual page. https://man7.org/linux/man-pages/man1/dd.1.html.

[DLLZ16] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No pardon for the interruption: New inference attacks on Android through interrupt timing analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 414–432. IEEE Computer Society, 2016.

[DWY+22] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, and Fengwei Zhang. Strongbox: A GPU TEE on Arm endpoints. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 769–783. ACM, 2022.

[GL22] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM TrustZone. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 300–316. ACM, 2022.

[Gooa] Google. AVF architecture. https://source.android.com/docs/core/virtualization/architecture.

[Goob] Google. Hafnium architecture. https://hafnium.googlesource.com/hafnium/+/HEAD/docs/Architecture.md.

[HJ23] Seung-Kyun Han and Jinsoo Jang. MyTEE: Own the trusted execution environment on embedded devices. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[HZY+24] Haoyang Huang, Fengwei Zhang, Shoumeng Yan, Tao Wei, and Zhengyu He. SoK: A comparison study of Arm TrustZone and CCA. In *International Symposium on Secure and Private Execution Environment Design, SEED 2024, Orlando, FL, USA, May 16-17, 2024*, pages 107–118. IEEE, 2024.

[Jes] Bruno Jesus. ProxyLite. https://github.com/00cpxxx/proxylite.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In André Seznec, Uri C. Weiser, and Ronny Ronen, editors, *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 350–361. ACM, 2010.

[Lin] Linaro. TrustedFirmware-A. https://www.trustedfirmware.org/projects/tf-a/.

[LML21] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2363–2380. USENIX Association, 2021.

[LSDB18] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. SeCloak: ARM trustzone-based mobile peripheral control. In Jörg Ott, Falko Dressler, Stefan Saroiu, and Prabal Dutta, editors, *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2018, Munich, Germany, June 10-15, 2018*, pages 1–13. ACM, 2018.

[MEB+] Jim Meyering, Paul Eggert, Padraig Brady, Bernhard Voelker, and Collin Funk. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/.

[MS96] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 279–294. USENIX Association, 1996.

[NXP21] NXP. i.MX 8M Dual/8M QuadLite/8M Quad Applications Processors Reference Manual. https://www.nxp.com/webapp/Download?colCode=IMX8MDQLQRM, 2021.

[PKS+21] Chang Min Park, Donghwi Kim, Deepesh Veersen Sidhwani, Andrew Fuchs, Arnob Paul, Sung-Ju Lee, Karthik Dantu, and Steven Y. Ko. Rushmore: securely displaying static and animated images using TrustZone. In Suman Banerjee, Luca Mottola, and Xia Zhou, editors, *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021*, pages 122–135. ACM, 2021.

[PL22] Heejin Park and Felix Xiaozhu Lin. GPUReplay: a 50-KB GPU stack for client ML. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 157–170. ACM, 2022.

[PS19] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM Comput. Surv.*, 51(6):130:1–130:36, 2019.

[Pur] Purism. Librem 5. https://puri.sm/products/librem-5/.

[Reu20] Reuters. Germany at odds with Apple on smartphone coronavirus contact tracing, Apr 2020. https://www.reuters.com/article/us-health-coronavirus-europe-tech/germany-at-odds-with-apple-on-smartphone-coronavirus-contact-tracing-idUSKCN2251MR.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 431–446. USENIX Association, 2015.

[SBS+24] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. ACAI: Protecting accelerator execution with Arm Confidential Computing Architecture. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[SKLR11] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 401–412. ACM, 2011.

[SLZK25] Fan Sang, Jaehyuk Lee, Xiaokuan Zhang, and Taesoo Kim. PORTAL: Fast and secure device access with Arm CCA for modern Arm Mobile System-on-Chips (SoCs). In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, pages 4099–4116. IEEE, 2025.

[SSBS24] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using malicious #VC interrupts to break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4220–4238. IEEE, 2024.

[SSK+24] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: breaking confidential vms with malicious interrupts. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[SSW+15] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. TrustICE: Hardware-assisted isolated computing environments on mobile devices. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 367–378. IEEE Computer Society, 2015.

[SSWJ15] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming smartphones into secure one-time password tokens. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 976–988. ACM, 2015.

[Vla] Denys Vlasenko. Busybox. https://busybox.net/.

[WZD+24] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. CAGE: Complementing Arm CCA with GPU extensions. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[YLL+24] Huaiyu Yan, Zhen Ling, Haobo Li, Lan Luo, Xinhui Shao, Kai Dong, Ping Jiang, Ming Yang, Junzhou Luo, and Xinwen Fu. LDR: secure and efficient linux driver runtime for embedded TEE systems. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[ZB22] Tamir Zahavi-Brunner. Attacking the Android kernel using the Qualcomm TrustZone, 2022. https://tamirzb.com/attacking-android-kernel-using-qualcomm-trustzone.

[ZHN+23] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. SHELTER: Extending Arm CCA with isolation in user space. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 6257–6274. USENIX Association, 2023.

[ZSS+16] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptol. ePrint Arch.*, page 980, 2016.