It’s TEEtime: Bringing User Sovereignty to Smartphones

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

ETH Zurich

Abstract

The majority of smartphones either run iOS or Android operating systems. This has created two distinct ecosystems largely controlled by Apple and Google—they dictate which applications can run, how they run, and what kind of phone resources they can access. Barring some exceptions in Android where different phone manufacturers may have influence, users, developers, and governments are left with little control. Specifically, users need to entrust their security and privacy to OS vendors and accept the functionality constraints they impose. Given the wide use of Android and iOS, immediately leaving these ecosystems is not practical, except in niche application areas. In this work, we propose a new smartphone architecture that securely transfers the control over the smartphone back to the users while maintaining compatibility with the existing smartphone ecosystems. Our architecture, named TEEtime, is based on ARMv8 and implements novel, TEE-based, resource and interrupt isolation mechanisms which allow the users to flexibly choose which resources (including peripherals) to dedicate to different isolated domains, namely, to legacy OSs and to user’s proprietary software. We show the feasibility of our design by implementing a prototype of TEEtime on an ARM emulator.

1 Introduction

Smartphones are the centerpiece of most people’s digital life. However, they do not offer the same flexibility as PCs, where users can install and run arbitrary software. Smartphone manufacturers and vendors of major operating systems such as iOS and Android1 restrict which apps can be run on smartphones, type of peripheral access, and data access. While Android allows side-loading, users are still limited in the way they can run apps and the kind of access they have. There are several examples where, in order to protect the users from developers,

1In the following, Android refers to the Google-controlled variant. While there is the Android Open Source Project (AOSP), it does not come with core features such as Google Mobile Services, which are closed-source and proprietary to Google [7, 20]

Apple and Google limit apps’ access to peripherals and data even if the users would allow such access [21, 32]. Smartphone manufacturers and OS vendors can use such control to optimize performance, provide a good user experience, and protect users from malicious apps. At the same time, these companies become arbiters and gatekeepers. Recent examples show that this is not a minor issue. In the case of contact tracing apps, Apple (and to some extent Google) restricted access that government apps can have to Bluetooth beacons, citing privacy and performance concerns. This restricted the design space and performance of contact tracing apps in several countries [32, 38, 39]. After political unrests in the US, the Parler app has been removed from Apple and Google app stores [19]. Apple and Google policies required in-app purchases to use in-store payments, resulting in these companies being accused of gatekeeping in several jurisdictions [40]. Users, developers, and governments have therefore faced restrictions under the current model. Most notably, users are unable to freely use their smartphones—they are subject to several limitations that they do not face on their PCs.

This is clearly an undesirable situation. Even if the current OSs offer some leeway, like side-loading, they can and have taken it away from users and developers [41]. A few companies should not be in a position to have such control. As much as possible, control over the devices should be handed back to the users who own the phones and whose data the phones primarily process and hold. One obvious solution is to replicate the PC model for smartphones. Projects that allow this already exist [29, 30]. However, this would require the users to switch to other app stores, making it hard to replicate the functionality and protections that large OSs such as iOS and Android offer. But more importantly, if one runs legacy OSs on top of an open platform, e.g., by virtualising the OS, one removes the ability of the OS to interact directly with the phone hardware and to guarantee the security that such direct interaction currently provides. Virtualization-based solutions [3, 8] would further hand the control to the hypervisors. However, such privileged software can inspect and modify the OS and app memory, therefore, removing control from the users and
We build TEEtime on top of ARM architecture. One of the TEEs are well suited to address the challenges that we outlined above. They can, as exemplified by TEEtime, offer the flexibility in the deployment of new apps and functionalities and align the interests of users, operators, and OS vendors.

2 Motivation & Problem Statement

In this section, we describe three exemplary use cases to highlight where the incentives of current OS vendors collide with users’ desire for privacy and functionality. We then generalize these examples into our design goals and finally introduce the threat model of our design.

2.1 Motivating Examples

Data privacy concerns. Users face uncertainty about when and what kind of data is collected by their phone, for example through the phone sensors, and how it is processed [12]. While the OS allows the user to manage peripheral permissions for apps or to disable access to some resources globally, e.g., by turning off GPS or Bluetooth, they have to trust the OS. Intentional or unintentional misuse of such OS-level privileges or opaque policies can put the user data in danger without the user’s knowledge. For example, Google used to gather location information even when the location history feature was turned off [4, 34].

Network Access - Censorship Circumvention. Various approaches to avoid censorship that do not require a gateway for tunneling communication rely on modifying the headers of IP/TCP packets, e.g. by segmenting TCP packets, injecting SYN packets, or adapting sequence numbers [10, 44]. Unfortunately, apps running on a mobile OS cannot make use of such strategies [22]. IP/TCP packets and their headers are assembled by the OS, leaving only two possibilities for implementing these circumvention strategies. First, the censorship circumvention logic can be implemented by the OS. Second, the OS can provide a low-level interface to apps where they can specify IP-level properties. In both cases, the OS has to implement a functionality for the user. Crucially, as censorship is usually dictated by a country’s government, if the OS vendor was to give users ways to circumvent it, it risks being denied access to the country’s market. Thus, the OS vendor has to choose to either turn a profit or to grant the functionality to the user. To break these incentives, it would be necessary to grant an app direct and unfiltered access to the network card, without relying on the OS for this access. This bars the OS vendor from any penalty, as it would not be directly aiding users to bypass censorship. However, even if such access would be available and possible in a smartphone architecture, the app itself risks being censored: State actors might force OS providers to not let the app be executed or scheduled. Such censorship has happened before [9, 31]. Therefore, the app needs to be provided with integrity and confidentiality guarantees against the OS and needs quality of service (QoS) assurances. Particularly, it should not be possible for the OS to block other apps from communicating with the app managing the censorship circumvention or to selectively block the app from timely handling incoming and outgoing network packets.

Bluetooth - Contact Tracing. During the Covid pandemic, a number of Bluetooth-based contact tracing approaches were developed [42, 43]. Bluetooth-based contact tracing apps periodically receive and process Bluetooth Low Energy (BLE) beacons to measure the distance to other smartphones and register contact. Due to concerns about privacy and power consumption, iOS and Android did not permit apps running in the background to freely broadcast and receive BLE beacons, effectively disallowing BLE-based approaches preferred by some countries [32, 38, 39]. Instead, the introduction of such contract tracing apps almost entirely depended on Apple and Google implementing and providing an API for a particular decentralized contact tracing approach [5, 32]. Even though the companies cooperated in the end, these companies could always disable the API again.
2.2 Goals

Our objective is to enable the user to define and run different domains on their phone that can arbitrarily control any resource assigned to them. By domain, we refer to a bundle of software, data, and hardware resources that the software in the domain may utilize. For example, the user can decide to run a legacy OS such as Android or iOS in one domain, a secure network app in another, and a contact tracing app in yet another domain.

Domains need to be able to accommodate the user’s functional and security requirements. Functional requirements revolve around the user being able to specify which hardware resources are assigned to which domain, and maintaining backward compatibility with the existing ecosystems so that users can enjoy the benefits of a legacy OS, without surrendering control over their devices. The security requirements instead concern the user’s differentiated trust in the various domains running on their device. This requires that the confidentiality and integrity of one domain cannot be violated by other domains and that the access of a domain to certain peripherals can be prohibited due to trust issues.

2.3 Threat Model & Scope

We consider an ARM-based architecture, as phones nowadays are predominantly based on it. We trust the hardware provided by the manufacturer of a phone, including the peripherals’ hardware and firmware. The manufacturer of the phone typically also provides the privileged software running in the secure state (S-EL 2 to S-EL 0) and the secure monitor layer (EL3). Software in EL3 has the highest privileges in the system, and so it is also included in the TCB of every domain running on the system. We assume that this software is reasonably static: it is placed there by the manufacturer and can only be updated with consent by the user. Software in secure state offers security services to the user, but also to content providers or apps (e.g. banking applications). As it is typically provided by the manufacturer as well, we consider it trusted for compatibility reasons. However, long-term, we aim for a design that allows us to remove these services and domain TCB. The software deployed in a domain is trusted only within its own domain, i.e., for accessing domain resources and processing domain data. However, domains can expose interfaces to other domains e.g., to allow proxy access to a resource. It is up to a domain developer to decide whether to trust an exposed interface.

We consider a software-only attacker with control of one or several domains. This attacker aims to violate the integrity and confidentiality of other domains and tries to access resources not assigned to its domains. We consider software vulnerabilities, physical attacks, and side-channels out of the scope of this work.

3 Background on ARMv8-A Architecture

In this section, we give a brief summary of details of the ARM architecture that are relevant for our work. Figure 1 depicts the a generic ARM platform with two execution cores. ARMv8-A cores offer four privilege layers, from EL0 (lowest privilege) to EL3 (highest privilege). EL3 is populated by the secure monitor (SM), which is commonly considered trusted and performs low-level and highly sensitive tasks such as trusted boot, power management, and context switches between secure and non-secure state (sometimes referred to as normal state). Other privilege levels can invoke the SM and request its services through the secure monitor call (SMC) instruction. Platforms with TrustZone extensions furthermore distinguish between secure and non-secure states for EL0 to EL2. This processor state can be switched only in EL3. Software running in secure state is more privileged than and protected from software in the non-secure state. The current state of the processor is determined by the NS bit, which is propagated on buses. Commercial off-the-shelf ARM devices are commonly equipped with partition controllers, which allow state- and sometimes core-based access control to memory (MPC) and peripherals (PPC) [14]. MPCs can be used to divide memory address space in different regions, for which access policies can be defined separately. PPCs either work with address space partitioning of the (memory-mapped) peripherals as well, or offer per-peripheral access control. The controllers can be configured from within the secure state and EL3. An example of such controllers is the ARM TrustZone Address Space Controller.
Interrupt handling with ARM GICv3  On ARM platforms using the Generic Interrupt Controller v3 (GICv3), the interrupt system consists of the distributor for the configuration of interrupts that are shared among cores (SPIs), re-distributors for the configuration of interrupts that are banked per core (SGIs and PPIs), and the CPU interfaces for the handling of interrupts. The (re)distributors are memory-mapped peripherals while the CPU interfaces are accessed through system registers. Interrupts are identified by an id, called INTID. When an interrupt’s source (e.g. a peripheral) asserts the interrupt, its status in the distributor becomes pending. If the interrupt is enabled, the controller logic then forwards the interrupt to a CPU interface. With default configuration, SPIs can be routed to any core. However, software can configure the routing information of an INTID (called affinity), which designates one core as the recipient. If the chosen core does currently not mask interrupts, an interrupt will be triggered, and the software is in charge of handling it. Once software acknowledges the interrupt through the CPU interface, the interrupt’s state becomes active. When software is done with handling the interrupt, it signals an end-of-interrupt, which sets the interrupt status to inactive.

The GICv3 supports the TrustZone extensions and allows interrupts to be configured as being either secure or non-secure. Secure interrupts cannot be configured or handled by software in the non-secure state. In addition, non-secure interrupts are usually signaled as IRQ to the CPU, while secure interrupts are signaled as FIQs, which have a higher priority and can only be handled by secure software.

4  TEEtime: Overview

As described in Section 2.2, our goal is to allow users to create different trust domains on their phones and assign resources (memory, compute time, peripheral access) to those domains. The domains and their access to resources need to be natively isolated thus eliminating the need to trust a large code base or parties that have interests in controlling the way that users use the platform.

To achieve this, we leverage TEE-inspired mechanisms to create multiple isolated execution contexts in the normal execution state. In each isolated execution context, the user can deploy the code and data of one domain. Some domains might already be pre-deployed on the platform (e.g., legacy OS), and depending on the platform policy, the users would or would not be allowed to modify such software. For example, if a legacy OS such as Android is pre-deployed and running on the platform, the user might be barred from inspecting or modifying its memory or be able to otherwise influence its behavior. Other domains might be freshly installed by and under full control of the user. In either case, the user decides which domain is assigned which resources.

We build TEEtime on an ARMv8-A platform, where we reserve non-secure EL0 to EL2 for the isolated execution contexts, which are used to run domain software as defined by the user. We use EL3 to enable TEEtime; here, we extend the SM with the functionality to isolate the execution of domains and configure access control to platform peripherals. A user can define a domain by providing a software bundle to the monitor, together with a manifest that describes the resource demands of the domain. In our design, EL3 code is either immutable or can be updated only if allowed (i.e., signed) by the manufacturer and/or approved by the user. Ideally, EL3 code will be public to further increase platform transparency.

In Figure 2 we depict two different instantiations of TEEtime in which domains share the platform either temporally or spatially. With temporal sharing, only one domain can be active on the platform at a time and temporal isolation mechanisms are in place. This means that cores run either the active domain’s software or are suspended. Furthermore, any software can only access resources that are assigned to the currently executing domain. When EL3 performs a domain switch, access to these resources is withdrawn, and the resources are made available to the new active domain. This approach has the advantage that it is achievable with simpler hardware control blocks, as only a distinction between currently accessible or inaccessible resources needs to be made.

With spatial sharing of the platform, different domains execute on different cores simultaneously and spatial isolation...
mechanisms are in place. An example is shown in Figure 3. In this case, access control to memory, peripherals, and interrupts is performed based on the core performing the access and based on the core-to-domain mapping. This requires more elaborate hardware that supports per-core access control. However, spacial isolation allows greater flexibility in terms of resource utilization.

In our design of TEEtime and its implementation on ARM, we support both temporal and spacial isolation of domains. Below, we describe the details of our design. In particular, we briefly describe the isolation of a domain’s execution and data, domain scheduling, enforcement of peripheral access, and isolation of interrupt handling.

4.1 Isolation of Execution and Data

TEEtime enforces isolation of execution in the SM by facilitating context switches between domains. Such a context switch can be triggered by a few events: by a domain issuing an explicit call to either schedule another domain, a domain simply yielding their execution, or by a privileged timer interrupt arriving, signaling that the monitor should resume the domain with scheduling capability (cf. Section 4.2). The SM saves, clears, and restores the domain state during context switches. This includes flushing instruction and data caches, and saving and restoring all registers that are read-and-write accessible to ELs 2 to 0.

We use hardware-based memory protection, i.e., an MPC, to isolate domain data and code. With temporal sharing, the SM configures memory regions belonging to the next domain as accessible, and the rest as inaccessible. With spatial sharing, the SM re-configures the MPC during a context switch such that each core can only access memory regions belonging to the domain they are executing. Previous work has shown how both temporal [37] and spatial [11] isolation of execution, code, and data can be achieved on ARM platforms. In contrast to these works, we do not use software running in secure EL1 for isolation enforcement, but perform this task in EL3. This has the advantage that less state switches are required for a domain switch, as a domain switch only requires to go through EL3, and not EL3 and S-EL1.

4.2 Domain Scheduling

We need an entity in charge of assigning compute time and cores to domains. The scheduling of domains is independent of any scheduling that might happen inside of domains (where e.g. an OS is still in charge of scheduling its apps). To keep the size of the shared TCB small, we decided to not put the scheduling functionality in the SM. Instead, the user designates one domain as the scheduling domain. The scheduling domain has the privilege to invoke and preempt other domains. For this, the domain performs a call to the SM, which initializes or restores the state of the new domain and performs other steps necessary for isolation before handing control to the scheduled domain. As the scheduling domain can violate the scheduling requirements of other domains, domains are not guaranteed availability. However, if the user trusts their chosen scheduling domain, they can assume that the compute time assignment defined by them is upheld.

For temporal sharing, the SM sets up a secure timer that triggers after a time period determined by the scheduling domain and ensures that all cores are either suspended or set up for executing the scheduled domain before handing control over. If the domain does not yield execution before the time specified by the scheduling domain, a secure timer interrupt (FIQ) triggers that hands control to the SM, which in turn resumes the scheduling domain to make the next scheduling decisions. As the domain cannot change the configuration of the secure timer and the corresponding interrupt is always routed to secure software, the domain cannot avoid being preempted. In case of spatial isolation, the scheduling domain assigns one or more cores to the scheduled domain, which are initialized by the SM. The cores remain under the control of the scheduled domain until it yields them voluntarily.

4.3 Peripheral Access

TEEtime supports not only domain isolation but also allows domains domain software to directly access peripherals. To accommodate the user’s trust assumption, the user is able to specify which peripherals a domain is allowed to access. TEEtime then enforces this decision by employing the PPC.

In TEEtime, peripherals can only belong to one domain at a time. The choice of mapping one peripheral to a single domain avoids issues that arise when two execution contexts access the same peripheral concurrently. To support such simultaneous accesses securely, such peripherals would need to be virtualized or made domain-aware. However, when needed, the ownership of any resource (peripheral) can be transferred, if allowed by the user. The domain that owns a peripheral can also act as a proxy for other domains if allowed by the trust assumptions. Finally, many peripherals are interrupt-driven peripherals, which means that 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.

4.4 Interrupt Isolation

One of the core novel aspects of TEEtime is the way that it handles interrupt delivery and isolation. We enforce interrupt isolation both for safety and security reasons. Safety, since domain software might not function correctly if it does

\footnote{The only exception are peripherals for which access is handled through system registers and interrupt handling is banked per core. An example for this is the ARM Generic Timer, which is commonly used for scheduling.}
not receive expected interrupts or receives unexpected interrupts. Security, to protect the domains from cross-domain side-channel leakage and from the violations of the integrity of their execution by other domains. This means that we need to make sure that miss-configuration and miss-routing of interrupts must not happen, neither accidentally nor maliciously.

An interrupt, therefore, must only be configurable by and be routed to the domain that owns it. We say that a domain owns an interrupt if the peripheral that is the interrupt source belongs to the domain. Each interrupt is owned by at most one domain, as one INTID strictly belongs to one peripheral and each peripheral can be owned by only one domain at a time. To achieve isolation, the following guarantees therefore need to hold:

1. A domain can configure the interrupts it owns.\(^3\) No domain may configure an interrupt that it does not own.
2. Only the owning domain can change the interrupt state, i.e. set it to pending, active, or retired.
3. Interrupts can only be routed to the owner domain.

Unfortunately, interrupt configuration is tightly packed: Up to 32 interrupts can share one memory-mapped configuration register. Therefore, address space based isolation as is used for memory and peripherals cannot be applied. In Section 5.2 we show how we achieve these guarantees in TEEtime, both for temporal and spatial domain isolation. This is one of the core innovations of TEEtime.

### 5 TEEtime: Main Architectural Components

In this section, we describe the key operations and components of TEEtime.

#### 5.1 Domain Life Cycle

**Platform Boot.** On platform boot, the SM runs after a series of trusted boot stages. The SM sets up access control to platform resources and then hands control to the designated scheduling domain. Like all domains, the scheduling domain is equally subject to user’s resource access policies. Legacy software like the Android OS might also already be pre-deployed on the platform. Their access to resources is restricted by the user, but the user is restricted in that they cannot access the memory or modify the execution of the running domains. In some cases, the user might also be restricted in that it will not be able to modify or update the software of some domains (e.g., in case of legacy OSs whose updates are signed by the OS vendors).

**Launching Domains.** Prompted by the user, the scheduling domain can load a sovereign binary and its manifest into memory and invoke the domain dispatcher, a module within the SM that handles domain switches and isolation, to set up the new domain. The user specifies the domain as either being spatially or temporally shared. If the required resources for the domain are unavailable (e.g., because they are being used by another domain), we inform the user about such overlap and stop the launch.

Once the new domain is set up, the scheduling domain can ask the dispatcher to run it. In the case of temporal sharing, it can request that the scheduled domain is preempted after a certain time and control handed back to the scheduling domain. This process can be repeated arbitrarily often until the app yields or terminates. The teardown of a domain can only be initiated by the domain itself. For this purpose, it can invoke the dispatcher, which frees resources, clears the domain’s state and memory, and then resumes the scheduling domain. We allow launching more than one domain. As long as their resource requirements do not conflict, the user can keep launching domains on demand.

#### 5.2 Interrupt Isolation

One of the main challenges of isolating domains on a platform is supporting their (isolated) access to peripherals. Here, we show how TEEtime implements such isolation without the need for peripheral or domain virtualization. In particular, we propose and implement a software-based isolation on top of ARMv8 with GICv3.

**TEEtime Temporal Interrupt Isolation**

Similarly to the temporal sharing of memory and peripherals, the isolation of interrupts can be achieved by partitioning the interrupt handling system (where one partition comprises the interrupts of one domain) and denying access to any partition that does not belong to the currently running domain. In addition, only interrupts that belong to the active partition might trigger and be routed to a core;\(^4\) otherwise, interrupts would be routed to a domain that is not its owner. The partitions are activated and deactivated by the SM during domain switches.

To allow a domain to configure its own interrupts, the SM configures the interrupts of the active partition as non-secure. The domain can then decide which of its interrupts it wants to receive and enable them. The SM protects the configuration of interrupts not belonging to the active domain by marking them as secure during the domain switch. This prevents the active domain from maliciously modifying interrupt configuration. To prevent the interrupts of the inactive partition from falsely triggering, the SM also marks them as disabled. As

\(^3\)This includes things as en-/disabling interrupts, setting its priority and activation mode, or deciding which core an interrupt should be routed to.

\(^4\)Additionally, secure interrupts that belong and are handled by the SM or secure software might trigger. These interrupts, however, are not part of domains and are protected by their configuration as secure interrupts.
software can only change the state of interrupts routed to the core it is running on, this also protects the interrupt state from modifications from outside the domain. However, the peripheral, i.e., interrupt source, can still set the interrupt to pending, as the access restriction only applies to software. The GIC preserves this pending state, at least as long as the interrupt source does not explicitly de-assert it.\(^5\) When a domain is rescheduled, its interrupt configuration is restored, and all interrupts that are enabled and pending trigger in order of decreasing priority.

Our approach for temporal interrupt isolation works with existing GIC hardware and does not require modifying a domain’s GIC access and interrupt handling logic. However, for the software, the observed interrupt behavior might slightly change. First, the order of interrupt arrivals is not preserved. Second, if a peripheral activates an interrupt periodically and does not wait for software confirmation, only the first occurrence of the interrupt is noted. Third, the interrupt handling latency increases by the time it takes until the domain is invoked again. However, as interrupt behavior is usually not deterministic, software should not rely on such assumptions for its correct functioning anyways. If such assumptions exist, the domain could instead be deployed under spatial sharing, as these properties are upheld as long as the domain is executing on a core.

### TEEtime Spatial Interrupt Isolation

GIC partitions need to be available simultaneously for spatial isolation, requiring domain-based access control and routing of interrupts. We describe how an ideal spatial interrupt isolation system would function to fulfill the requirements listed in Section 4.4 and what additions to existing GIC functionality would be required to approximate this functionality. As this would require hardware changes, we propose an alternate solution that uses existing hardware and pushes the domain-based access control to software instead.

First, to be able to enforce spatial isolation, the interrupt controller must be able to distinguish from which core an access originates. Second, the controller must know for each interrupt ID which domain is allowed to access it and on which core it is running on. On an access, the controller combines this information to decide whether the access is allowed. When an interrupt is pending, the interrupt system must route the interrupt to a valid recipient. This is similar to how interrupts can be configured as secure or non-secure on ARM GICv3, but unfortunately, this is insufficient as there is no further distinction (e.g., based on core id).

To enable the ideal functionality, we discuss one option for a hardware design change. We want to ensure that each domain has access to the configuration of its partition of the GIC, and that interrupts belonging to its partition are routed to cores the domain is executing on. Existing ARM interrupt controllers provide some of the required functionality: The GIC supports the concept of affinity, allowing normal world software to specify to which target core an interrupt should be routed.\(^6\) This feature allows us to specify that an interrupt should be routed to a specific core. However, the configuration of the GIC is not protected; any currently running software may change the routing and other parameters. We cannot protect the configuration by marking the interrupt secure, as this would prevent the legit domain from configuring and the routing behaviour of the interrupt would change to that of FIQs. Instead, it would be ideal if the GIC supported access control based on affinity: Only the core which is the target of a specific interrupt can change its routing behavior and other configuration. This ensures that once an interrupt id is allocated to a core, only that core can configure it. Given such a GIC, the initial assignment can be done by EL3 software, and then at runtime, the GIC can distinguish and enforce checks. This proposal for hardware changes to the GIC is only one out of many more potential options to support the requirements from Section 4.4.

Since we aim to stay compatible with existing hardware, we design a solution that leverages the SM to do domain-based filtering. Realizing access control on the GIC’s configuration with current hardware relies on one major observation: On ARM platforms, interrupts are configured through memory-mapped registers of the GIC. Usually, the configuration of interrupts is mostly performed during the boot of a device. During runtime, interrupt are not frequently reconfigured, but it sometimes occurs, for example, when a peripheral is hot-plugged, or load balancing is adapted. In contrast, the majority of the runtime workload is the handling of interrupts. Each core has a CPU interface that performs the handling. Thus, the handling does not affect shared state across cores and thus can be spatially isolated easily.

This observation allows us to use an approach that incurs a performance penalty for configuration accesses: We block all accesses from non-secure software to the GIC using the PPC. To configure interrupts, an execution context needs to perform an SMC to EL3 instead of accessing the GIC directly. The SM combines information about which core the request comes from, which domain is running on that core, and IN-TID ownership to decide whether an access is allowed. If it is allowed, EL3 performs the access. Figure 4 depicts the overview of our access-control placements with an example.

Our proposed solutions for spatial interrupt isolation—both for hardware supporting the ideal functionality and the approach based on software-enforcement—require changes to legacy software to make it aware of the new configuration regime for the GIC. In Section 6.2 we describe a non-invasive solution to this issue.

---

\(^5\) Usually, a peripheral only de-asserts the signal once it has received confirmation of the software that the interrupt is being handled.

\(^6\) Traditionally, this is used to balance the interrupt load of a system, increase overall system performance, or provide low-latency for certain tasks, e.g., network handling.
We describe the behavior of the dispatcher for both use cases. The domain dispatcher supports temporal or spatial modes. It can configure the GIC for the current state of registers and puts the saved state of the next domain to the configuration of MPC and PPC in place. Additionally, the dispatcher also updates the GIC configuration. For temporal sharing, this task is performed by the GIC as actions that must be performed on the SM need only be performed during domain switches. In contrast, the GIC-guard can be called by domains throughout their runtime.

### 5.3.1 Domain dispatcher

The domain dispatcher supports temporal or spatial modes of platform sharing. The scheduling domain informs the dispatcher about the sharing type of the domain being dispatched. We describe the behavior of the dispatcher for both use cases.

#### Temporal Sharing

When a new sovereign domain is set up, the dispatcher sets up a clean state for it (meaning setting registers and interrupt states to reset values). It creates configurations for MPC and PPC that reflect the state when the new domain is running, i.e., where only resources belonging to the new domain are accessible. Furthermore, it creates a new GIC configuration for this domain. For enabling preemption, the dispatcher enables the EL3 timer interrupt and configures it to be routed as FIQ. We do not change any other secure interrupts in the dispatcher. The dispatcher configures the target domains interrupts as non-secure and disabled, such that the domain can decide to enable them as necessary. For all the remaining interrupts belonging to other domains, the dispatcher sets them as secure and disabled.

When a domain switch takes place, the dispatcher saves the current state of registers and puts the saved state of the next domain together with the configuration of MPC and PPC in place. Additionally, the dispatcher also updates the GIC configuration. For temporal sharing, this task is performed by the GIC as actions that must be performed on the SM need only be performed during domain switches. In contrast, the GIC-guard can be called by domains throughout their runtime.

### 5.3.2 GIC-guard

The GIC-guard module is used by domains to configure the GIC when direct access to it is blocked. For this purpose, domains can invoke the GIC through an SMC call. The GIC expects the memory address to be accessed, whether the access is a read or write access, and, in 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 more straightforward processing in the GIC-guard and mirrors how software currently accesses the GIC. From the shared state with the dispatcher, the GIC-guard can infer which domain made the request and which access permissions it has.

The teardown of an app, including the adaptation of access control, is performed either when the domain signals that it wants to be stopped, or when a user action requests it. If the teardown of an app ends the spatial sharing of the platform, GIC access restrictions are lifted.

### 5.3 TEEtime Secure Monitor Extensions

In TEEtime, we extend the SM with two modules: the domain dispatcher and GIC-guard. The domain dispatcher module is the center of the isolated domain framework. It is in charge of providing clean states for domains, facilitating domain switches, and enforcing isolation between domains. The GIC-guard module provides the service of GIC configuration to domains. The dispatcher and the GIC-guard have shared state concerning which domain is running on which core and which INTIDs belong to each domain.

Figure 4: Spatial Interrupt Isolation. Domain 1 running on core 1, can configure the interrupt with the INTID 1 that it owns only by calling the EL3 GIC-guard. A direct access to the corresponding memory region of the GIC is blocked by the hardware-based access control, as the domain is running in non-secure state. Instead, it must send a request to the SM running in EL3. If the domain requests access to an INTID it does not own, such as INTID 5, the SM will not perform the access to the GIC.

5.3.1 Domain dispatcher

The domain dispatcher supports temporal or spatial modes of platform sharing. The scheduling domain informs the dispatcher about the sharing type of the domain being dispatched. We describe the behavior of the dispatcher for both use cases.

#### Temporal Sharing

When a new sovereign domain is set up, the dispatcher sets up a clean state for it (meaning setting registers and interrupt states to reset values). It creates configurations for MPC and PPC that reflect the state when the new domain is running, i.e., where only resources belonging to the new domain are accessible. Furthermore, it creates a new GIC configuration for this domain. For enabling preemption, the dispatcher enables the EL3 timer interrupt and configures it to be routed as FIQ. We do not change any other secure interrupts in the dispatcher. The dispatcher configures the target domains interrupts as non-secure and disabled, such that the domain can decide to enable them as necessary. For all the remaining interrupts belonging to other domains, the dispatcher sets them as secure and disabled.

When a domain switch takes place, the dispatcher saves the current state of registers and puts the saved state of the next domain together with the configuration of MPC and PPC in place. Additionally, the dispatcher also updates the GIC configuration. For temporal sharing, this task is performed by the GIC as actions that must be performed on the SM need only be performed during domain switches. In contrast, the GIC-guard can be called by domains throughout their runtime.

#### Spatial Sharing

When a new sovereign domain is set up, the dispatcher sets up a clean state for it as for temporal sharing. Since in spatial sharing, we have to keep track of the state of all domains that are being spatially isolated, we have to maintain dedicated data structures—simple save restore as in temporal sharing is insufficient. To do this, the dispatcher updates the data structures that store domain access permissions to memory, peripherals, and interrupts. Additionally, if this is the first spatially isolated domain being launched after boot up, we need to protect the GIC’s configuration registers that are memory-mapped. We use the dispatcher to configure the PPC such that the GIC is non-accessible from non-secure software. Once the the domain is scheduled, the dispatcher puts isolation mechanisms in place and ensures that all cores assigned to it are shut down before starting the target domain. The domain can then wake up other cores that it owns.

The teardown of an app, including the adaptation of access control, is performed either when the domain signals that it wants to be stopped, or when a user action requests it. If the teardown of an app ends the spatial sharing of the platform, GIC access restrictions are lifted.

5.3.2 GIC-guard

The GIC-guard module is used by domains to configure the GIC when direct access to it is blocked. For this purpose, domains can invoke the GIC through an SMC call. The GIC expects the memory address to be accessed, whether the access is a read or write access, and, in 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 more straightforward processing in the GIC-guard and mirrors how software currently accesses the GIC. From the shared state with the dispatcher, the GIC-guard can infer which domain made the request and which access permissions it has.

Based on the access permissions and the address of register that is being accessed, the GIC-guard then assembles a bit mask, where bits that are allowed to be accessed are set to
one. This is necessary because registers are shared between interrupts, e.g. the interrupt-enabled registers are 32-bit and hold the state of 32 interrupts. For a write access, this mask together with the old value of the register is then used to compute the new value. For a write access, the mask is used to set inaccessible bits to zero. Ignoring writes and returning zero is on par with specified GIC behavior for unauthorized accesses.

Special attention needs to be put on 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 PPC. This requires careful evaluation of some registers where the GIC presents different views to to secure and non-secure accesses. Furthermore, we must assure 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, e.g. by using locks, to avoid race conditions.

6 Maintaining Software Compatibility

For a functioning system, the domains deployed on the system need to fulfill some requirements. First, one domain must have the functionality to be the scheduling domain. The user could decide to install an app specifically for this task, or re-use an existing operating system. Below, we discuss how a kernel can be retrofitted for this purpose without invasive changes. Second, any domain that runs under spatial sharing of the platform, needs to be aware that it cannot access the GIC directly but must use the GIC-guard. We describe how a minimal EL2 runtime can be used to retrofit domains with this knowledge. Finally, domains must be able to run with the platform resources they are assigned to. We assume that the user ensures that a domain has sufficient access to such resources. For some software, for example the Linux kernel, it is possible to describe the hardware they are running on by providing a device tree during the domain’s boot. If this is supported, a device tree describing only the resources that the domain can accessed can be passed.

6.1 Legacy Software as Scheduling Domain

For our proof-of-concept implementation, we decide to designate a domain running a full Linux Kernel as the scheduling domain. This allows us to reuse the built-in scheduling and power management functions. We implement two loadable kernel modules for temporal and spatial sharing that perform SMC calls to the domain dispatcher. Furthermore, we provide applications that can be invoked by a user and that prepare the platform by loading the sovereign binary into memory and, in the case of temporal sharing, shutdown cores currently not needed. Then, the applications invoke the respective kernel module. In the case of spatial sharing, the application is done. In the case of temporal sharing, the application waits for a response from the kernel module, which in turn relays information from the SM detailing whether the scheduled domain was preempted by an FIQ or yielded execution. If the domain was preempted, the domain reinvokes the kernel module. In this way, the application acts as a stand-in for the domain for the purpose of scheduling (when the kernel scheduler schedules the application, it implicitly schedules the domain). If the domain yielded, it is considered to be done with execution, and will only be resumed when the user requests it. Domain switches are only from the scheduling domain to another domain and back, never between two non-scheduling domains.

6.2 Compatibility with Gatekeeper

Legacy software that has not been retrofitted will not invoke the GIC-guard, but try to access the GIC directly. As a result, an asynchronous abort that informs about an illegal non-secure access to an address will be routed to the GIC-guard. Unfortunately, the information provided by an asynchronous abort are insufficient to reconstruct the attempted memory access. Software usually proceeds with execution immediately after dispatching a write transaction on the interconnect without waiting for a success confirmation. As a result, the abort does arrive a few instructions afterwards, making it hard to impossible to figure which instruction caused the abort. Even if it is possible to infer which instruction faulted and from which register the value was to be written, the state of this register might have changed already, making it impossible to reconstruct the attempted write with certainty.

Consequently, it is essential that domains do use the GIC-guard when accessing the GIC. Unfortunately, retrofitting large code bases such as the GIC driver in the Linux kernel is, while possible, tedious work that does not scale. Instead, we propose putting a minimal layer between legacy software running in EL2 and the GIC-guard that uses synchronous aborts to reconstruct a memory access. In contrast to asynchronous aborts, synchronous aborts trigger in the core immediately and provide precise information about the fault. To trigger the fault, stage 2 translation tables are set up that deny access to the GIC memory regions, but leave the configuration of the rest of the address space untouched. A minimal exception handler parses the relevant information from the synchronous abort syndrome register and perform the SMC call to the GIC-guard. It is important to note that the EL2 runtime does not perform a security function and does not become part of the system TCB: Hardware still enforces that software in non-secure state cannot access the GIC. The small runtime in EL2 simply provides compatibility for legacy EL1 software.
7 Security Analysis

This section presents the results of a security analysis of the isolation between domains. 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.

7.1 Memory

To achieve memory isolation, we leverage memory protection based on an MPC as described in Section 4.1. The MPC allows configuring access permissions for a limited set of contiguous memory regions according to the source of the access request, i.e., the core id. The MPC is on the hot path to memory and checks every transaction against its configuration. Therefore, no memory access that violates the configured access control policy will succeed. The MPC also only accepts configuration changes from EL3 or the secure world, both of which are considered trusted.

Assuming that the configuration setup within the trusted software is correct, an attacker that runs before and after a victim domain cannot access the victim’s memory. This is because the memory is always marked as inaccessible from non-secure state while the attacker is running. This can be enforced because software in EL3 handles all domain switches. When the attacker is running concurrently to the victim, the victim’s memory must be accessible for non-secure software as this required by the victim. However, as access is restricted to cores executing the rightful domain software, accesses stemming from the attacker domain are blocked.

7.2 Peripherals

We make the design decision to only assign peripherals explicitly to an individual domain. Our prototype does not allow to share a peripheral between two domains at the same time. Thus, we handle the peripheral the same way as a memory region belonging to a domain: we leverage a hardware mechanisms that can provide access control for every access to peripherals, i.e., the PPC. The PPC only accepts configuration changes from EL3 or secure world. Therefore, a software adversary cannot modify the access policies. The details correspond to those described for memory isolation in Section 7.1.

7.3 Interrupts

In Section 4.4, we list these three guarantees that need to hold for secure interrupt isolation:

1. No domain may configure an interrupt that it does not own.
2. Only the owning domain can change the interrupt state.
3. Interrupts can only be routed to the owning domain.

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 (1) or state (2) 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 (3).

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 distributor interface and therefore the configuration of interrupts. Instead, software in the SM brokers the access. A domain can only invoke the SM on a core that it is running on as cross-core SMC calls are not possible, and in its role of domain dispatcher, the SM is always aware which domain runs on which core. Based on this information, it allows or blocks access to interrupt configuration (1). In addition, the SM does not allow a domain to set the affinity value of an interrupt to a core that is outside the domain (3). Interrupt states are managed through system registers, which allow each core to handle interrupts that are routed to it. A core cannot access another core’s system registers. Consequently, a core cannot change the state of interrupts routed to other cores. Therefore, together with guarantee 3, we achieve guarantee 2.

8 Implementation

We implement a prototype of the domain framework on ARM Fixed Virtual Platform (FVP). The FVP emulates a reference implementation of an ARMv8 platform, including essential peripherals such as timers, UART, and network stacks. The FVP comes with the GICv3 interrupt controller and an TZC-400 Address Space Controller for memory isolation. We use a standard configuration of 8 cores. As ARM provides the FVP as a reference processor implementation, we can ensure that the developed software is as generic as possible. We plan to make our implementation publicly available.
8.1 Proof-of-Concept

We extend the Trusted Firmware A (TF-A) v.2.4, an open-source reference implementation for software in EL3 provided by ARM, to add support for the domain dispatcher and the GIC-guard (cf. Section 5.3.1, 5.3.2). Both modules can be invoked from EL1 and EL2 through an SMC call. We implemented kernel modules for Linux v.5.10.107 that perform these SMC calls to invoke domain setup and run domains under temporal and spatial isolation. For prototyping, we used a generic Linux OS based on the Yocto project, which can easily be replaced by other Linux-based systems such as AOSP.

8.2 Sovereign Applications

To demonstrate the platform, we implemented three proof of concept applications for the sovereign domain: a bare-metal UART driver, a Linux kernel running busybox on an initial ram file system (initramfs), and a networking app executing on the kernel. They showcase increasing complexity from performing simple computations to using peripherals and I/O.

**Bare-Metal UART driver** We showcase that a sovereign domain application can independently perform I/O using dedicated peripherals. It demonstrates that TEEtime can receive and handle interrupts of peripherals within the sovereign domain, which is essential for efficiently communicating with devices. This application provides a console in the sovereign domain which mirrors a string, entered through UART and terminated through a newline (“Enter”). We implement it without any underlying Kernel or OS to show a sovereign domain’s basic functionality and usage capabilities to execute bare metal applications.

During first execution, the application executing in EL1 configures the UART device. For this, it sets the baud rate, buffer locations, and other communication options. To receive interrupt-based input, the application sets up an interrupt handler and stores the address to the VBAR register. The application changes the configuration in the corresponding GIC register to activate the input interrupt. After finishing the initialization, the application enters an idle loop to wait for inputs. When the UART module receives a character, it triggers an interrupt to notify the application. The application reads the input buffer and stores the character in a dedicated buffer. Furthermore, it checks the input for newline characters, indicating that the user pressed the “Enter” button. When it detects such a case, the interrupt handler signals the main loop to print the buffer contents and flush it afterwards.

**Kernel boot** We show that the sovereign domain is capable of booting and executing the Linux kernel and a miniature execution runtime, together serving as an OS.

To boot the Linux kernel in a sovereign domain, we reduced the device tree to contain only basic functionalities such as an interrupt controller, system timers, UART I/O, and dedicated DRAM memory, thus exposing only essential platform components to the kernel. To produce a more lightweight kernel image, we change the kernel compiling configuration to disable support for non-essential devices and to include an initramfs image. The initramfs image contains Busybox, which provides a Linux bash-like environment to emulate a minimal operating system.

For booting, we place the compiled device tree blob and the Kernel-initramfs package into the dedicated memory area of the sovereign domain. In the SM, we set the first register to the address of the device tree blob and start the execution at the first instruction of the kernel binary. After booting finishes, the execution yields the Busybox prompt where we can execute busybox utilities.

**Network Access** We consider a scenario where an application in the scheduling domain lacks the network access required to communicate with a remote server. Therefore, it has a companion application in another domain that receives the legacy app’s data, creates a network connection with the server, and sends the request. When the network domain receives a response, it forwards it back to the legacy domain for the application to handle. Figure 5 depicts the message flow between scheduling domain, sovereign domain and a remote server.

We implement a message channel via a shared buffer between the scheduling and network domains. The scheduling domain application can write any string-encoded data to a size-adjustable buffer of the sovereign companion to read. The network companion application consists of a Linux kernel package, along with an additional application capable of opening a socket connection to a remote host. We used the Linux PoC as a base to get better support for operating the integrated network controller without requiring a standalone driver. The network device is assigned (using the manifest) to
the network domain.

For this PoC we implement the remote party as a simple TCP socket server. It receives the legacy domain’s message from the sovereign companion, calculates a hash, and sends it back. The sovereign companion app then forwards the hash to the legacy application using the shared buffer.

9 Evaluation

In this section, we evaluate our proof-of-concept implementation for functionality, performance, and complexity.

Complexity We add 741 LoC to the SM to implement the GIC-guard and dispatcher and change 321 LoC of the existing Trusted Firmware. These are the only changes we make to the TCB. We change 748 LoC in the u-boot to set up EL2 runtime and to set up new exception handlers for trapping into EL2 necessary for GIC-guard. Our only change to the scheduling kernel is to reserve memory regions for domains in the device tree. We add kernel modules with a complexity of 138 LoC and 149 LoC to the scheduling domain for invoking temporal and spatial isolation.

Executing a Kernel in the Sovereign Domain We show that TEEtime domains are capable of executing existing code, including a kernel boot. To demonstrate this we start with an existing kernel that boots as scheduling domain. It takes 974M and 980M cycles for booting up on 1 and 8 cores, respectively. A boot with the GIC-guard and EL2 runtime activated takes 982M and 1001M cycles for booting up on 1 and 8 cores. As expected, we see a higher cycle count to initialize 8 cores in both cases.

We run the stripped-down Kernel described in Section 8.2. To ensure that its functional, we first boot up this version in the scheduling domain. It takes 1160M and 1173M cycles for booting up on 1 and 8 cores. We then boot it again in the scheduling domain with GIC-guard enabled. It takes 1162M and 1177M cycles for booting up on 1 and 8 cores. This slowdown is expected because the interrupt configurations are routed via EL2. Further, we see that booting 8 cores is slower that 1 core as expected.

Finally, we boot the kernel in the spawned domain. It takes 1081M cycles for spacial boot and 5076M cycles for temporal boot. We observe that spatial boot is faster than the temporal boot because it does not incur any context switches and enjoys dedicated execution on the core. Second, booting in a spawned domain is faster than booting in the scheduling domain. This is explained by the overhead of setting up devices in the scheduling domain which has access to a large fraction of devices. Whereas in the sovereign domain, the bootup only setups 1-2 devices (UART and network card).

More importantly, the functionality of the Kernel is not perturbed. For all the above runs, the kernels are reactive in their UART console, interrupts as reported in /proc/interrupts behave as expected and the kernels can access disk and the network within the bounds of their domains.

Sovereign Apps We run the two bare-metal apps under both spatial and temporal sharing on the FVP. First, we run the UART app such that it has control over a UART terminal for the duration of its lifetime. It takes 2.456 seconds and 2.204 seconds to launch the app in temporal and spatial mode respectively. In both modes, the app functions correctly and prints the strings that we input on the terminal. Next, we run the network access app with ownership of the network card and UART terminal. It takes 25.965 seconds to launch the app (including underlying kernel layer) in spatial mode. We interact with the app and provide inputs repeatedly. We can observe the processing in the network domains, the communication with our TCP server, and the response in the application. Interrupts are routed reliably to the owning domain in all cases. This validates that TEEtime achieves the desired functionality where domains have unbrokered and functional access to peripherals.

Impact of GIC-guard First, we confirm our assumption that GIC configuration accesses occur predominantly during the kernel boot, and later only under certain events. For this purpose, we track the accesses in the GIC-guard. During boot, the kernel attempts to access and setup the configuration of all available interrupt IDs on the system, sets them as appropriate for interrupts that are actually used and to default values for others. After boot, GIC accesses only occur on certain events, for example when cores are suspended, peripherals are (un-)plugged, or the user requests the rerouting of an interrupt. We observe a total of 37 reads and 19 writes when we boot the unmodified kernel. For the stripped down kernel running in the additional domain, we see 25 reads and 329 writes. The number of writes increases in the additional domain because it attempts to re-write the values blocked configuration memory. After boot we do not see any reads or writes to the GIC configuration memory, as the handling of interrupts is performed through system registers as expected.

Context Setup and Switching Costs We measure the time required to setup the sovereign world context. For spatial sharing, we pay a one time cost of booting the domain and jumping to it. After the initialization, there are no further context switches. For temporal sharing, we measure the time required for context switches. Switching from scheduling to sovereign domain takes 9231 cycles, the other way around takes 1570 cycles. We record a total of 8 context switches when booting the kernel in the temporally isolated sovereign domain. The number of context switches for the app is dependent on how long we keep the app running.
10 Discussion

TEEtme’s paradigm of isolated, equally privileged domains with access to hardware is necessary for a sovereign phone. Next, we outline remaining challenges and alternatives.

Involvement of the user. For TEEtime, we assume that the user has sufficient technical knowledge to define domain manifests and has a secure way to install domains and communicate their manifests to the SM. In a real deployment, users would probably not define a domain’s manifest themselves, but would rely on one provided by the domain software developer and then adapt it according to their privacy and security needs. Furthermore, the installation of domains would probably be facilitated by a pre-installed domain such as the scheduling domain. We imagine a system with a secure one-way-path to the SM, e.g., through a directly wired button or status LEDs, and that makes use of attestation, possibly using a second device for verification during domain installation.

Availability TEEtime currently offers confidentiality and integrity for domains, and peripheral control for the user. However, it does not provide a strong availability guarantee: Apps are not guaranteed to be assigned execution time and to be scheduled. The user only gets the guarantee that their scheduling requirements are met under the assumption that they trust the scheduling domain. We can address this by deploying a dedicated scheduling domain instead of relying on a legacy OS’s scheduler. Alternatively, one could equip the dispatcher with compliance functionality: as it is involved in every context switch, it can observe how often and for what time a domain is scheduled. It can match this information to the scheduling requirements defined by the user in the domain’s manifest. If the scheduling domain violates the scheduling requirements, the SM could then notify the user.

Legacy Secure Software One of our goals is compatibility with existing systems including secure world software. Often, these are manufactured software for not only users but also to content providers (e.g. banking application). In TEEtime, such services should also be deployed in a separate, protected domain. However, it may not be straightforwardly, as these services might assume secure world privileges. For mid-term deployments, it might be acceptable to leave this software in the secure world.

ARMv9 Realms ARM Confidential Compute Architecture in ARMv9 introduced the Realm Management Extensions (RME). We considered it as a potential option for our design. For example, we can deploy the legacy OS in the normal state, and then deploy sovereign software stacks in realms. However, this approach does not meet our requirements. First, the non-secure state is not protected from realm state. Similar to our design, restricting realm’s access would require using memory protections blocks via the monitor. Second, RME does not support native peripheral access from realms. Most importantly, domains cannot configure and receive physical interrupts. Instead, interrupts are virtualized by the normal world hypervisor (or OS) and then forwarded to realms. This could lead to the hypervisor spoofing, dropping, or delaying interrupts. Third, the isolation between realms is enforced by page tables, which are maintained by the realm manager. We argue that the complexity of setting up page tables and managing them is significantly higher than isolation based on physical memory protection mechanisms.

11 Related Work

Mobile phone design follows the ARM processor design quite closely. The manufacturer software runs in the ARM TrustZone’s secure world, the OS and user the apps run in the normal world. Since the secure world can access the normal world, the OS must trust the manufacturer’s software running in secure world. Due to its criticality, the secure world is usually locked down; only the manufacturer can deploy software on it. Examples of manufacturer software running in the secure world include Samsung Knox [33], Qualcomm’s Secure Execution Environment [13], Google’s Trusty [2], and Huawei’s iTrustee [23]. Finally, the closest ARM-based design to ours is Sanctuary [11]. Sanctuary uses similar techniques to run isolated but restricted enclaves in the normal world as we do. However, we run entire operating systems with exclusive access to peripherals concurrently in the normal world.

Beyond ARM TrustZone, TEEs on other architectures provide process-level [15, 27] or VM-level isolation [1, 6, 24]. They protect the so-called enclaves from a malicious operating system or hypervisor. In contrast, we provide equal protection for all domains from each other. TEEs typically do not support external devices out of the box, they need additional capabilities [35, 45]. Secloak restricts untrusted software from accessing peripherals and interrupts [28]. However, it can only allow or block access to all software running in non-secure space. In contrast, TEEtime allows enforcement based on trust assumptions about domains.

Hypervisors such as KVM [25], XEN [8], VMware, and HyperV concurrently run multiple OSes. Most of them have large and complex code-bases while security-focused minimal hypervisors such as seL4 [26] and NOVA [36] are smaller. We consciously decide against using a hypervisor for enforcing domain isolation. First, employing a hypervisor for this purpose takes the ability to run a hypervisor away from domains. For example, this would prohibit Android from using the latest KVM feature [17]. Second, making a hypervisor part of the shared TCB pushes the issue of software maintenance and control from the OS layer to the hypervisor.
12 Conclusion

We present TEEtime, a new smartphone architecture that allows users to control with which resources and software runs on their phones. We leverage mechanisms from the TEE space to enable the user to install isolated domains with differing resource requirements and trust assumptions. The novelty of our approach is that these domains are equally privileged, and may have direct access to peripherals and the interrupt handling system. This requires that the interrupt handling system can be securely shared by multiple domains. For this purpose, we develop two mechanisms that allow the partition of the ARM interrupt controller, one for a setup where domains run on the platform simultaneously and one for a setup where they run alternately. We implement a prototype of TEEtime and through it demonstrate the feasibility of our architecture.

Acknowledgments

This work was supported (in part) by the ETH4D Humanitarian Action Challenge Grant, Zurich Information Security and Privacy Center (ZISC), and ETH Future Computing Lab (EFCL).

References

[1] AMD. AMD memory encryption. Technical report, AMD, 2016.

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

[3] Jeremy Andrus, Christoffer Dall, Alexander Van’t Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 173–187, 2011.

[4] AP News. Google tracks your movements, like it or not, Aug 2018. https://apnews.com/article/828aefab64d4411bac257a07c1af0ecb.

[5] 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.

[6] ARM. Arm Confidential Compute Architecture.

[7] Ars Technica. Googles iron grip on Android: Controlling open source by any means necessary, Jul 2018. https://arstechnica.com/gadgets/2018/07/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/.

[8] 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. ACM SIGOPS operating systems review, 37(5):164–177, 2003.

[9] BBC News. Apple bans Hong Kong protest location app, Oct 2019. https://www.bbc.com/news/technology-49919459.

[10] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 2199–2214, 2019.

[11] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: ARMing TrustZone with user-space enclaves. In NDSS, 2019.

[12] Business Insider. A security expert found that Apple’s latest iPhone can still track your location data, even if you toggle it off for every app, Dec 2019. https://www.businessinsider.com/apple-iphone-11-pro-collects-location-data-krebs-report-2019-12?r=US&IR=T.

[13] Liang Cai. Guard your data with the qualcomm snapdragon mobile platform, 2019.

[14] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. Rezone: Disarming trustzone with tee privilege reduction. arXiv preprint arXiv:2203.01025, 2022.

[15] Victor Costan and Srínavas Devadas. Intel SGX explained. IACR Cryptol. ePrint Arch., 2016(86):1–118, 2016.

[16] Fiona Scott Morton Cristina Caffarra. The european commission digital markets act: A translation | vox, cepr policy portal. https://voxeu.org/article/european-commission-digital-markets-act-translation, Jan 2021.

[17] Jake Edge. KVM for Android. https://lwn.net/Articles/836693/.

[18] European Commission. The digital markets act: ensuring fair and open digital markets. https://ec.europa.eu/info/strategy/priorities-2019-2024/europe-fit-digital-age/digital-markets-act-ensuring-fair-and-open-digital-markets_en, Jan 2019.

[19] Forbes. Parler at risk of going offline after bans from Amazon, Apple and Google, Jan 2021. https://www.forbes.com/sites/jemimamcevoy/2021/01/
[20] Google. Google mobile services. https://www.android.com/gms/.

[21] Google. MediaRecorder overview, January 2021. https://developer.android.com/guide/topics/media/mediarecorder.

[22] Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin. GET /out: Automated discovery of Application-Layer censorship evasion strategies. In 31st USENIX Security Symposium (USENIX Security 22), pages 465–483, Boston, MA, August 2022. USENIX Association.

[23] Ltd Huawei Technologies Co. Huawei iitrustee v3.0 on kirin 980, 2020. https://globalplatform.org/certified-products/huawei-iitrustee-v3-0-on-kirin-980/.

[24] Intel. Intel trust domain extensions. Technical report, Intel, 2020.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In Proceedings of the Linux symposium, volume 1, pages 225–230. Ottawa, Ontario, Canada, 2007.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkeduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 207–220, 2009.

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

[28] Matthew Lentz, Rijuureka Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: ARM TrustZone-based mobile peripheral control. In Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, pages 1–13, 2018.

[29] Pine64. PinePhone. https://www.pine64.org/pinephone/.

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

[31] Reuters. Google, Apple remove Navalny app from stores as russian elections begin, Sep 2017. https://www.reuters.com/world/europe/google-apple-remove-navalny-app-stores-russian-elections-begin-2021-09-17/.

[32] 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.

[33] Samsung. Samsung knox. https://www.samsungknox.com/.

[34] Douglas Schmidt. Google data collection. Technical report, Vanderbilt University, 2018. http://www.dre.vanderbilt.edu/~schmidt/PDF/google-data-collection.pdf.

[35] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Ćapkun. Composite enclaves: Towards disaggregated trusted execution. IACR Transactions on Cryptographic Hardware and Embedded Systems, pages 630–656, nov 2021.

[36] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In Proceedings of the 5th European conference on Computer systems, pages 209–222, 2010.

[37] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 367–378. IEEE, 2015.

[38] The Guardian. France urges Apple and Google to ease privacy rules on contact tracing, Apr 2020. https://www.theguardian.com/world/2020/apr/21/france-apple-google-privacy-contact-tracing-coronavirus.

[39] The Guardian. NHS in standoff with Apple and Google over coronavirus tracing, Apr 2020. https://www.theguardian.com/technology/2020/apr/16/nhs-in-standoff-with-apple-and-google-over-coronavirus-tracing.

[40] The New York Times. Fortnite creator sues Apple after ban from app stores, Aug 2020. https://www.nytimes.com/2020/08/13/technology/apple-fortnite-ban.html.

[41] The Verge. Apple is blocking Apple silicon mac users from sideloading iPhone apps, Jan 2021. https://www.theverge.com/2021/1/15/22233754/apple-blocking-m1-iphone-app-sidelodaing.

[42] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. arXiv preprint arXiv:2004.13293, 2020.
[43] Carmela Troncoso, Mathias Payer, Jean-Pierre Hubaux, Marcel Salathé, James R. Larus, Edouard Bugnion, Wouter Lueks, Theresa Stadler, Apostolos Pyrgelis, Daniele Antonioli, Ludovic Barman, Sylvain Chatel, Kenneth G. Paterson, Srdjan Capkun, David A. Basin, Jan Beutel, Dennis Jackson, Marc Roeschlin, Patrick Leu, Bart Preneel, Nigel P. Smart, Aysajan Abidin, Seda F. Gürses, Michael Veale, Cas Cremers, Michael Backes, Nils Ole Tippenhauer, Reuben Binns, Ciro Cattuto, Alain Barrat, Dario Fiore, Manuel Barbosa, Rui Oliveira, and José Pereira. Decentralized privacy-preserving proximity tracing. arXiv preprint arXiv:2005.12273, 2020.

[44] Zhongjie Wang and Shitong Zhu. Symtcp: eluding stateful deep packet inspection with automated discrepancy discovery. In Network and Distributed System Security Symposium (NDSS), 2020.

[45] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In Proceedings of the seventh ACM on conference on data and application security and privacy, pages 261–268, 2017.