# An REE-independent Approach to Identify Callers of TEEs in TrustZone-enabled Cortex-M Devices

Antonio Ken Iannillo
SnT, University of Luxembourg
Luxembourg, Luxembourg
antonioken.iannillo@uni.lu

Sean Rivera
SnT, University of Luxembourg
Luxembourg, Luxembourg
sean.rivera@uni.lu

Darius Suciu
Stony Brook University
New York, USA
dsuciu@cs.stonybrook.edu

Radu Sion
Stony Brook University
New York, USA
sion@cs.stonybrook.edu

Radu State
SnT, University of Luxembourg
Luxembourg, Luxembourg
radu.state@uni.lu

## ABSTRACT

Internet of Things (IoT) devices are becoming increasingly ubiquitous in our lives, from personal health monitoring to house and factory management. Further, IoT devices are becoming increasingly complex, and ensuring their security is of paramount importance. As a result, they started to include Trusted Execution Environments (TEEs) to protect security-critical IoT operations.

This paper focuses on improving the security of the next-generation IoT devices by introducing *Secure Informer*, an identification and authentication mechanism for ARM TrustZone for Cortex-M. Under *Secure Informer*, the TEE can directly determine the Rich Execution Environment (REE) context without introducing additional communication or dependency on the REE software stack. We implement our solution for ARMV8-M architecture, showing its efficacy with no need to change the source code of the REE. Empirical results show that *Secure Informer* can help mitigate confused deputy attacks targeting TEE-running services while only incurring an average 3.2% overhead on the device performance and requiring an additional 464 lines (52 bytes in the compiled binary file) to the TEE.

## CCS CONCEPTS

• **Hardware** → *Safety critical systems*; • **Computer systems organization** → **Embedded software**; *Processors and memory architectures.*

## KEYWORDS

TEE, TrustZone for Cortex-M, confused deputy attack, caller identification

**ACM Reference Format:**
Antonio Ken Iannillo, Sean Rivera, Darius Suciu, Radu Sion, and Radu State. 2022. An REE-independent Approach to Identify Callers of TEEs in TrustZone-enabled Cortex-M Devices. In *Proceedings of the 8th ACM Cyber-Physical System Security Workshop (CPSS '22), May 30, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3494107.3522774

*CPSS '22, May 30, 2022, Nagasaki, Japan*
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9176-4/22/05.
https://doi.org/10.1145/3494107.3522774

## 1 INTRODUCTION

Software vulnerabilities can closely affect our daily lives, as Internet-of-Things (IoT) devices might monitor our health [38][28], control our homes and cities[34], or assist in our factories[32][36][40]. As the complexity of IoT devices increases, more and more developers are turning to designs that feature many different applications running in parallel, emulating more traditional computing architectures. As these types of embedded systems are becoming more connected and widely adopted, the primary security challenge is ensuring the correct functionality of the critical parts of the system. Security for IoT is a very challenging domain, as IoT devices are heavily resource-constrained. Researchers and practitioners have proposed many solutions to address this design space, with Trusted Execution Environments (TEEs) currently emerging as one of the most supported solutions[8][37][41].

TEEs are introduced due to increasingly bloated operating systems and applications (a.k.a. Rich Execution Environment or REE), which are prone to introducing software bugs due to their complexity. A TEE is an isolated environment where sensitive data are protected from malicious or compromised components in the REE, such as the OS or user applications. The TEE protects both the run-time states and stored assets of security-critical logic by running it isolated from the REE. Typical TEE use cases are

- the secure storage and usage of cryptographic keys,
- the protection of digital copyrighted information, or
- the safe processing of biometric sensor data.

.

With regards to IoT devices, ARM has introduced TrustZone technology in Cortex-M devices to implement TEEs[21]. While the names can be deceptive, it is essential to highlight that TrustZone technology in Cortex-M (TrustZone-M or TZ-M) is very different from the same technology concept implemented for Cortex-A (TrustZone-A or TZ-A)[12]. In the research communities, typically, TrustZone refers to TZ-A present in modern smartphones and other digital devices.

TZ-M consists of hardware extensions present in the processor, memory, and peripherals, and it enables embedded systems to run two different software environments, commonly known as the Normal (Non-secure) World (NW) and the Secure World (SW). The NW acts as a resource-restricted environment that runs no security-critical software. In contrast, the security-critical functionality is located in the SW and has complete control over all device resources.

The SW also represents the TEE and acts as the final arbiter for determining which device resources are accessible from the NW. In contrast, the NW can not directly access any SW resources.

The OS and user applications run inside the NW in a typical scenario. At the same time, the SW only contains a set of security-critical services exposed to the NW applications in the form of APIs. This paradigm decouples the secure and non-secure logic, allowing developers to focus on the security of the secure software without worrying about the additional code and vulnerabilities present in the non-secure logic implementations. The introduction of the NW and SW helps minimize the SW Trusted Computing Base (TCB) while retaining the more complex non-secure logic functionalities inside NW. As a result, NW applications can rely on the SW services to protect their sensitive data. For NW applications to use SW APIs, the NW and SW need to communicate. However, they are unaware of each other's layout and inner logic due to their isolation. As a result, a "semantic gap" is introduced, where SW services operate on NW received data without knowing their NW clients' explicit permission and capabilities. Attackers can exploit this lack of information and fool the SW into misusing its unrestricted capabilities. For example, a malicious NW task might break the NW's security mechanisms by making a simple request to the SW to write data in NW OS code pages, as the SW is unaware of the existing content of these pages or where the NW OS is.

This paper focuses on solving the semantic gap problem in TrustZone-M devices. It presents ***Secure Informer*, a novel approach implemented in the SW that can identify secure service calls from the NW**. *Secure Informer* works by monitoring the side-effects of intrinsic NW behavior and providing a semantic translation to the SW without directly communicating with the NW or depending on its implementation. As a proof of concept, *Secure Informer* is implemented on top of the ARM reference implementation of a secure firmware, namely trustedfirmware-m (tf-m)[23] and evaluated against several types of tf-m attacks to demonstrate its effectiveness. While current solutions need changes to the non-secure software, *Secure Informer* leverages available hardware (*i.e.,* the memory protection unit or MPU) to handle client identification in a transparent way to non-secure software. Further, our solution binds the identification mechanism to the secure service call instead of the task context switch mechanism in the NW scheduler, as currently happens.

Formally, we focus on evaluating the solution based on the following research questions:

RQ1 Does *Secure Informer* provide TrustZone-enabled Cortex-M devices an effective authentication mechanism?
RQ2 How does *Secure Informer* impact the code base?
RQ2.1 Are NW code changes required?
RQ2.2 What SW software stack changes are required for introducing *Secure Informer*?
RQ3 Is device performance affected under *Secure Informer*?

The remainder of this paper is organized as follows:

- Section 2 provides the scientific and technical background;
- Section 3 describes the identification problem in TZ-M devices;
- Section 4 presents the threat model;
- Section 5 discusses our solution, namely *Secure Informer*;
- Section 6 introduces the experiments and shows their results;
- Section 7 considers the limitation and the future works;
- Section 8 brings up the related work;
- Section 9 concludes the paper.

## 2 BACKGROUND

This section describes the core concepts of TEE, TZ-M, MPU, and the communication between SW and NW.

Different standards and models present TEEs (OMTP [24], GlobalPlatform [7], ARM [22]) but they do not define TEE unequivocally. However, common property in their definitions is that TEE guarantees an isolated environment with the confidentiality of its data[33].

ARM introduced TrustZone technology to support TEE implementations [2] [4]. The design paradigm relies on a processor-enforced execution split between different security states. A running process can execute either in a secure or a non-secure state, whereas the non-secure software cannot access secure resources directly. While TZ-A relies on the secure monitor to switch states[27], TZ-M splits the memory into three different security states: non-secure, non-secure callable, and secure[42]. Figure 1 shows the different ways of communicating between the SW and NW in both TZ-A and TZ-M. A process is in the same state as the code is executing. A non-secure process can become secure by passing through non-secure callable code first, while a secure process can always regress into a non-secure one. This paper focuses primarily on the secure world (SW) and the non-secure/normal world (NW) interaction, corresponding to the secure and non-secure states. The non-secure callable state is a unique state that serves as the primary transition point between the NW and SW. When switching between NW and SW, the device purges all registers and caches to ensure the SW is not leaking any information to the NW.

Figure 2 presents the security state changes of processes in Cortex-M with TrustZone technologies graphically. From a secure state, the process can use the `BXNS` or `BLXNS` instructions to jump or call a function in a non-secure memory region and, thus, run in a non-secure state. From a non-secure state, the process can use the classic `BL` instruction to jump in the non-secure callable area only to an `SG` (Secure Gateway) instruction. Then, it can branch to a secure memory region running in a secure state. The function executed in the non-secure callable area is sometimes called the veneer function. Further, it is essential to recall that state transitions also happen because of exceptions and interruptions that can be configured to land on secure or non-secure handler code.

Embedded devices can be either with or without an OS abstraction. In the latter case, the devices are very limited in the task they perform, and the software consists of a single control loop and an interrupt handler to respond to external events. Regarding TEE, we usually refer to the first category with a logical separation between the kernel and the user programs. In ARM terminology, they are also referred to as handler and thread mode. In particular, this paper considers those devices that also include a lightweight mechanism to protect memory from unauthorized access essential to implement the TEE.

The NW typically contains an RTOS that has several programs running under it. The RTOS executes each non-secure program in user mode, *i.e.,* a permission restricted execution mode. These

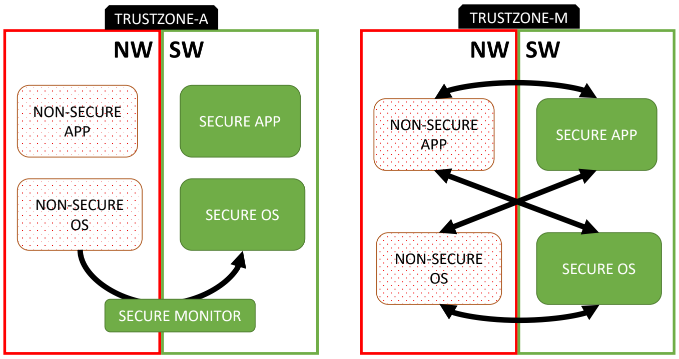

**Figure 1: Cross-world communication paths in TrustZone Technologies for Cortex-A and Cortex-M**

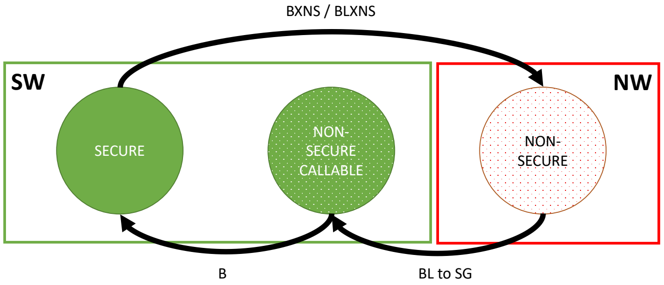

**Figure 2: Finite State Machine of Processes Life in Cortex-M with TrustZone Technologies**

programs are isolated from each other by a Memory Protection Unit (MPU), which segments the memory and distributes it between them. During execution, it is the responsibility of the RTOS to maintain separate environments for each user thread or task. In order to do so, the RTOS configures the MPU to limit what each thread can access and provides restrictions on hardware peripherals and scheduling. However, each executing thread also can access the SW, which is not aware of this separation because the SW sets up its MPU configurations and settings to protect its execution.

For Cortex-M devices, the MPU splits the memory into contiguously allocated blocks known as regions[1]. These blocks are assigned permissions of `read`, `write`, `execute`, and `privileged`. Each thread has its own corresponding MPU configuration, as does the RTOS. Each time there is a context swap, the RTOS alters the MPU configurations to the new thread configuration. In Cortex-M devices with TZ-M (a.k.a. ARMv8-M architectures with security extension), the MPU is banked between SW and NW (*i.e.,* there are two MPUs).

Inside the SW, the secure firmware provides individual isolated services to the NW. The size of these services is kept minimal due to its impact on the SW Trusted Computing Base (TCB). Developer guidelines advise that the SW contains only security-critical operations to minimize its attack surface. In contrast, all other operations should be conducted in NW [17][35]. Each service can be accessed directly by the NW executing thread. The thread must first call an NW callable gateway function that passes the request into the SW to access one of its services. Additionally, the SW halts all other threaded execution while the system is in a secure state. Secure services must complete their operations quickly to guarantee RTOS performance.

## 3 PROBLEM DESCRIPTION

This section describes the lack of information between the SW and the NW, the confused deputy problem that derives from it, and the limitations of current solutions.

As already presented in the previous sections, a TEE is guaranteed to be isolated from the REE, allowing communication through defined interfaces. The SW provides services to the NW threads that cannot perform such restricted operations (*e.g.,* cryptographic or secure storage functions). However, the SW has no information on the NW caller created, managed, and scheduled in an opaque way by the RTOS. This lack of information leads to the so-called confused deputy attack [11], where a restricted entity can fool a more privileged one to perform otherwise unavailable operations. In our scenario, an NW thread could exploit the SW to break the isolation between threads established by the RTOS. For example, it can access a secret key belonging to another NW application.

The identification of NW callers is the primary measure to counteract confused deputy attacks by assigning a unique and secure ID to each NW thread that interacts with the secure services. Then, the SW can distinguish different callers and avoid granting threads access to other non-secure threads' data by enforcing access policies in their services. Although this paper focuses on such an identification mechanism, it alone cannot solve all the aspects of the confused deputy problem. Section 8 will present more elaborated attacks

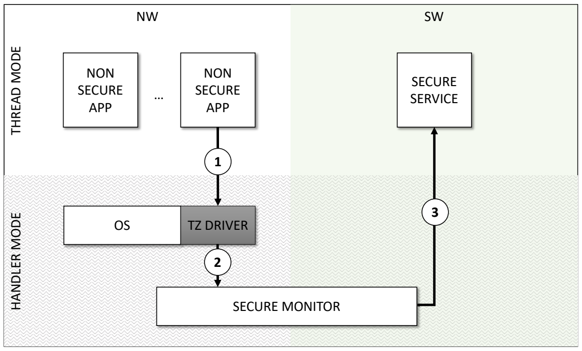

**Figure 3: Simplified View on a Non Secure Task Calling a Secure Service in TZ-A based devices (focus on identification)**

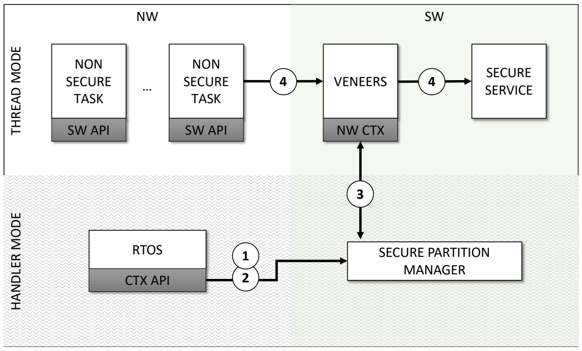

**Figure 4: Simplified View on a Non Secure Task Calling a Secure Service in Trusted Firmware M (focus on identification)**

and counter-measures. The following paragraphs will present how existing software stacks approached the identification problem.

Modern mobile devices, which incorporate TEE designs backed by TZ-A, usually implement the identification mechanism through a piggyback approach. Whenever a non-secure application calls a function in the SW, the call passes through the non-secure OS that adds the caller ID to the request before dispatching it to the secure monitor. The non-secure OS can do so because it is fully aware of the non-secure caller being the one that manages all the non-secure threads. In particular, Figure 3 shows the steps involved. First, the non-secure application ① requests a kernel module, namely the TrustZone driver, that retrieves the process ID of the running thread. Then, the request enhanced with the ID ② passes through the secure monitor for authorization, that ③ may dispatch it to the secure service. Regarding the IoT devices with TZ-M, non-secure tasks could directly jump in the SW, and they do not need to pass through the non-secure OS. Thus, such a solution may carry much overhead if adapted to the TZ-M devices.

Trustedfirmware-m (tf-m) is a reference implementation by ARM to a TEE empowered by TZ-M. It consists of a secure boot, a core in charge of isolation and communication between the two worlds (a.k.a. Secure Partition Manager or SPM), and essential secure services (*i.e.,* cryptographic, secure storage, and remote attestation). Processes in the NW can invoke these secure services through a standardized set of APIs, namely PSA Functional APIs (see Appendix A). Under the hood, the SPM takes care to dispatch the call and manage the NW context representation in the SW. Indeed, the SPM maintains a context during each call from the NW that includes at least the non-secure client ID (NSID). In tf-m, the RTOS in the NW must provide the NSID for each connection through the context APIs (see Appendix B). Otherwise, the same NSID is used for each connection. In both cases, the context is loaded and saved with the corresponding non-secure thread when the RTOS kernel performs scheduling. Figure 4 shows an example. At the moment of thread creation, ① the RTOS associates to it an NSDI to communicate to the SPM through the context API. The SPM, in turn, creates an NW context representation for that thread. When the RTOS schedules a thread, ② the scheduler invokes the SPM that ③ saves the previous context and loads the new context. Then, ④ the scheduled thread calls a secure service, while the SW has its NW context and the NSID. For the sake of simplicity, Figure 4 does not include the information exchange between the veneers and the SPM necessary for the request to be dispatched to the correct partition where the secure service runs.

Currently, the authors are not aware of other available solutions for the identification problem in TZ-M-based software stacks. However, the tf-m solution has limitations. First, the identification mechanism implies that developers need to make **changes to the code base of the RTOS**, in particular to the scheduler that is a critical component, to accommodate the calls to the SW at each context switch. Second, these changes are **dependant on the RTOS** and the different schedulers that it can include. More importantly, the whole tf-m solution depends on these modified versions of the RTOSes. Different communities need to clearly understand the management protocol of the SW, potentially leading to misinterpretation and severe bugs. Third, the reference implementation puts the solution **overhead on each context switch**. To the best of the authors' knowledge, there is no study about the workload on commercial IoT devices powered by MCUs with TZ-M[1]. However, the authors believe that not every time a task is scheduled needs to communicate with the SW, leaving room for optimization.

## 4 THREAT MODEL

This section defines the threat model under which *Secure Informer* was designed.

The focus is on protecting the NW against simple confused deputy attacks that leverage the information lack in the SW about NW threads. Such an attack is successful if a non-secure thread can access secure data that does not possess in the secure world. The proposed solution operates under the following assumptions:

- the non-secure RTOS is not compromised, and it is protected against attacks that do not leverage SW vulnerabilities (The authors believe this is a fair assumption, as once an attacker can compromise the RTOS, they can mimic any thread or system state needed to use the secure world);

[1] Further, since the authors' interest in TZ-M, the constant monitoring of the end-market for such IoT devices brought no results. If the reader is aware of any, the authors would be glad to have this piece of information shared with them. This paper leaves the workload analysis of these devices for future research.

- the RTOS will correctly configure security peripherals (*e.g.,* the MPU) to correctly match the security requirements of the NW (in particular, isolation of the non-secure tasks);
- the secure services are also not compromised and free of privilege-escalation vulnerabilities (Attacks that provide direct SW access to adversaries and hardware attacks are beyond the scope of this paper);
- any data received from the NW is completely under the attacker's control, and the SW must validate it in order to ensure the confidentiality and integrity of the secure data
- secure data can be stored in the SW by a non-secure thread;
- non-secure applications are isolated from each other in the NW by the non-secure RTOS;
- to perform the attack, a malicious thread can mimic requests sent from other non-secure threads (However, they can not perfectly mimic the system states, *e.g.,* the memory layout, of other running threads);

A successful attack is where an NW thread can mislead the SW about its capabilities or permissions. It may either impersonate another thread or successfully circumvents an SW check. An example of a successful attack would be leveraging the SW to access NW memory that the thread usually would not have permission to access.

There is no assumption about how long an attack can take or about the existence of any other defensive mechanisms in the NW (*i.e.,* mechanisms that would stop the attacker from sending arbitrary data to the SW). Additionally, there is no assumption about other security functionality in the SW. Successfully deceiving the SW is treated as a full compromise given the vast permissions of the SW.

This last paragraph will outline how such an attack can be performed in tf-m. As already mentioned, the NW access the secure service through APIs. Some of these APIs specify a resource ID that identifies the resource on which the NW thread wants the secure service to use. For example, it can be a value in secure storage or a private key. A correct thread owns some of these resources, and it expects the SW to protect them from other threads. If a running thread uses the API, the SPM reads the NW context and passes the request to the secure service together with the NSID. If the non-secure identification mechanism is not in place, the tf-m assigns the same NSID to all requests. Then, a malicious thread can guess the resource ID and make the SW operate on the correct thread's resource. Similarly, if the RTOS does not implement the necessary changes correctly, the identification mechanism can behave erratically, nullifying the efforts of the SW. In both cases, for example, the malicious thread can use the private key of a correct thread and impersonate it in network communication.

## 5 *SECURE INFORMER*

This section gives an overview of the design of the proposed solution, namely *Secure Informer*.

*Secure Informer* provides a mechanism to assign unique NSIDs and to bridge the "semantic gap" between the SW and the NW. Such a solution leverages the MPU to track the context of the NW directly, but no changes in the NW codebase are required. However, this approach needs to be extended and may touch in the future such a codebase as explained in Section 7.

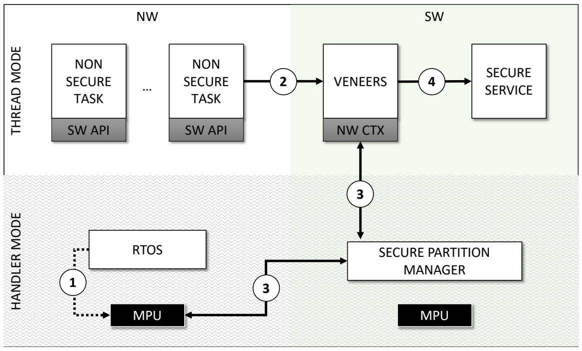

**Figure 5: View on the Approach Implemented by *Secure Informer* to Identify a Non Secure Task Calling a Secure Service**

*Secure Informer* depends on the hypothesis that the RTOS implements isolations correctly in the NW. The RTOS processes non-secure applications in executable threads of instructions. Threads are isolated from each other and the kernel. The MPU prevents threads from accessing data, code, and peripherals that the RTOS has not explicitly granted. Whenever the RTOS needs to create a user thread, it also creates an MPU configuration that grants access to a dedicated memory area that it will use as a stack buffer. Thus, each user thread has a unique MPU configuration that an external observer can use to distinguish among them. Also, the RTOS handler threads have a unique MPU configuration different from all the user threads. On a context switch, the RTOS loads the MPU configuration of the thread to be scheduled.

TZ-M splits the processing environment in two while duplicating registers and the MPU. NW and SW can operate independently and keep their MPU configurations on hold while changing the security state. The non-secure MPU is configured with the caller thread's configuration when the security state is changed to execute the secure functions in the SW. *Secure Informer* is implemented in the SPM and, upon reception of a request from the NW, computes the NSID by reading the non-secure MPU state.

Figure 5 shows *Secure Informer* graphically. At context switch, ① the RTOS changes the MPU state loading the configuration associated with the scheduled task. The non-secure task can then ② call a secure service using the SW API, that triggers the SPM to ③ read the non-secure MPU state, compute the NSID, and load the NW context. Finally, the secure service ④ receives the request and serves the non-secure task.

We implemented *Secure Informer*'s mechanism in tf-m[2] through a function named `set_client_id`. Algorithm 1 presents the pseudocode of *Secure Informer*'s implementation. Tf-m includes two different models for the communication between the SW and the NW: the IPC model and the library model. In the IPC model, the interactions are implemented through interrupts and `set_client_id` is called from the `SVC_Handler_IPC` function[3]. In the library model, the interactions are implemented through simple function calls and

[2] version 1.3, commit id `9d93424a12b8cc2ba56518f76edadeaabc29e143`

[3] `secure_fw/spm/cmsis_psa/tfm_core_svcalls_ipc.c`

**Algorithm 1** `set_client_id`

```
Global Variable: map, client_id
 1: ns_config = read_ns_mpu()
 2: for each region in ns_config do
 3:     if region.enabled then
 4:         str += region.RBAR
 5:     end if
 6: end for
 7: client_mpu_hash = hash( str )
 8: if map.has( client_mpu_hash ) then
 9:     client_id = map.get( client_mpu_hash )
10: else
11:     client_id = get_next_ns_client_id()
12:     map.put( client_mpu_hash, client_id)
13: end if
```

the `set_client_id` is called from the `tfm_core_partition_request` function[4]. In both cases, `set_client_id` is defined in tfm_nspm.h. The function makes use of two global variables: `map` and `client_id`. While `client_id` is already used by the SPM as the NSID, *Secure Informer* introduces `map` to store the mapping between the MPU configurations and the NSIDs. At the very beginning, `set_client_id` reads the non-secure MPU configuration, *i.e.,* the MPU configuration of the non-secure caller (line 1). Then it concatenates the base addresses of the enabled regions (lines 2-6). This byte array is unique to the client since it contains at least a read/write (R/W) region for its stack that no other clients can access in the NW. The function computes the hash of this array to retrieve the client id assigned to it (lines 7-9). If none is assigned, it creates a new NSID and puts it on the map for later usage (lines 10-13).

The way *Secure Informer* computes the byte array in lines 2-6 can be optimized by considering only the MPU regions with the R/W attribute. Further, some systems may have different threads executing the same application. It may be necessary to uniquely identify the application as a group of threads. The MPU configures Read-Only (R/O) regions (*i.e.,* code) that different threads can share from the same application to implement code isolation. Thus, an implementation would be to consider only R/O regions in line 3. Similarly, there can be cases where there is a need for a thread NSID and an application/group NSID. In this case, both of the previous cases can be implemented side by side. For the rest of this paper, we only consider the thread NSID as outlined in Algorithm 1.

The main challenge of cross-world identification is that the SW should know the context of the NW without being in regular communication with it. We already presented other approaches, such as the one implemented by tf-m (*i.e.,* through context management APIs) or the mechanism used by TZ-A based devices (*i.e.,* piggyback). Each carries its potential benefits and drawbacks, which we have summarized in Table 1. The choice between *Secure Informer* and the tf-m implementation comes down to the workload of the system. Systems with a large number of context switches and few calls into the SW will have better performance using *Secure Informer*. On the contrary, systems with few context switches and many calls will perform better with tf-m solution. Given that calls to the SW tend to be rarer than context switches, the authors believe that *Secure Informer* is a more appropriate and relevant solution for the majority of systems. Analyzing the specific workload of future commercialized devices with TZ-M will allow the developer to decide which is best for their system, which should be considered for future work.

# 6 EVALUATION

This section answer the research question presented in the introduction (Section 1) through the evaluation of *Secure Informer*.

## 6.1 Testing environment

The experiments run on qemu (v5.2.0) emulating mps2-an521. The baseline configuration consists of an unmodified trustedfirmware-m (version 1.3) running on the secure side and zephyr OS[31] (version 2.6) running on the non-secure side. Zephyr OS runs application threads at a reduced privilege level (user mode). The authors chose to leverage qemu over the physical hardware to benefit from virtualization's better tracing and analysis functionality. The enhanced configuration adds the *Secure Informer* changes in the trustedfirmware-m code.

The authors used three test suites. An ad-hoc test suite implements basic attack scenarios where a malicious thread tries and impersonates a correct thread to access otherwise private resources in the SW. The ARM PSA test suite checks whether the functionalities offered by the PSA APIs are delivered correctly. The tf-m regression test suite verifies that the code changes in the firmware do not impact the existing functionality.

## 6.2 Experiments

The authors conducted two series of experiments to assess the effectiveness and the computing overhead of *Secure Informer*.

In the first set of experiments, the objective is to assess whether *Secure Informer* can provide NSID correctly without disrupting the firmware functionalities.

In the second set of experiments, the objective is to compare the tf-m vanilla version (without changes) with the version enhanced with *Secure Informer*.

### 6.2.1 *Effectiveness Experiments.*

First, the authors run the ARM PSA and the tf-m regression test suites on the modified version of tf-m with *Secure Informer*. The tests should pass without any error to demonstrate that *Secure Informer* does not interfere with the functionalities of the secure framework.

Then, the authors designed an ad-hoc test suite to validate the solution in different usage scenarios of the two primary secure services already implemented in tf-m, namely the secure storage and cryptographic services. The test suite consists of a minimal environment with several different context-based attacks. For example, let A and B be two application threads running in their own space in zephyr OS. These threads are spawned by the RTOS and maintain a similar MPU configuration throughout the system life-cycle. Both A and B are isolated in the NW; both can access the SW but do not have permission to affect each other's SW resources. The goal is to ascertain security on two main tasks: protecting stored values and protecting key usage.

[4] `secure_fw/spm/cmsis_func/tfm_secure_api.c`

**Table 1: Advantages and Disadvantages for Identification Mechanisms in Systems with TrustZone Technologies**

| Approach | Advantages | Disadvantages |
|---|---|---|
| piggy-back (TZ-A) | straighforward solution | to implement changes to NW<br>SW has to trust completely the request from the NW |
| context management API | available reference implementation | to implement changes to NW<br>call to the SW at each NW context switch |
| *Secure Informer* | no changes to NW (ready-to-use)<br>lighter NW thread context switch | overhead upon requests to the SW |

**Table 2: Attack Scenarios in the Ad-hoc Test Suite**

| ID | Secure Service | Application A | Application B |
|---|---|---|---|
| 1 | Secure Storage | write x in SW | read x |
| 2 | | | read info of x |
| 3 | | | overwrite x |
| 4 | | | delete x |
| 5 | Crypto | create k in SW | read k |
| 6 | | | read info of k |
| 7 | | | create a copy of k |
| 8 | | | encrypt data with k |
| 9 | | | delete k |

For protecting stored values, the goal is to ensure the confidentiality of secure memory, *i.e.,* to guarantee that no other actors may tamper with secured values. The goal is to analyze the ability to read, write, delete, or utilize a secured value where the isolation forbids such actions. Application A initially stores value $X$ in the SW thanks to the secure storage service. Application B attempts to act on that value. Since they are isolated in the NW, B should not be able to interact with value $X$ as it belongs to application A.

For the security of key stores, both A and B use the same key store but should not have access to each other's keys. Application A generates a key $k$ through the cryptographic service in the SW. Application B tries and invokes different operations (*e.g.,* sign a message). Once again, if they are properly isolated in the SW, the key $k$ should not be accessible to application B.

Table 2 presents the ad-hoc test suite. In all cases, a test succeeds if application B is forbidden to interact with value $X$ or key $k$. These experiments aim not to evaluate the standard protections available in the non-secure RTOS but to evaluate how attackers can analyze the lack of shared context between the SW and NW to circumvent security protections. To validate this test suite, the authors run it against tf-m without any identification mechanism (same NSID for all the non-secure callers). All the tests failed as expected, confirming the validity of the attacks.

*6.2.2 Overhead Experiments.* The overhead experiments consisted of the executions of the ARM PSA and the regression test suites on tf-m with and without the changes of *Secure Informer*, intending to compute its overhead. While Qemu does not accurately trace the number of cycles an experiment needs to complete, it can deterministically generate instruction traces for the tests. In this way, we were able to measure the number of cycles by leveraging the instruction traces of Qemu.

## 6.3 Results

In the effectiveness experiments, *Secure Informer* passed the three test suites. This result showed that *Secure Informer* is able to effectively provide the SW client identification of non-secure callers (*cfr.* RQ1).

It is important to note that no change was made to the kernel of Zephyr OS to make *Secure Informer*. Thus, the author can confirm that an identification mechanism such as *Secure Informer* is effective without requiring any code modification to the NW (*cfr.* RQ2.1).

Further, the implementation of *Secure Informer*, for both IPC and library mode of tf-m, touched only five source files in the SPM codebase and one source file in the platform utilities. It consists of only 464 lines of code. It increases the tf-m binary size by 52 bytes (<0.01%) (*cfr.* RQ2.2).

During the overhead experiments, adding *Secure Informer* incurred 3.5% and 2.9% overhead while running the ARM PSA and regression test suites, respectively. Table 3 presents the results of these last experiments. Thus, the authors state that *Secure Informer* slightly affects the performance of the device (*cfr.* RQ3).

# 7 LIMITATIONS

The presented experiments indicate *Secure Informer* is effective in providing an identification mechanism without involving the NW, resulting in security against simple confused deputy attacks without significantly impacting system performance. However, some limitations should be addressed in future work.

A first limitation is the persistence of NSID through device reboots or updates. While some RTOS systems hard-code the memory areas to assign threads as a stack, some randomly assign them. Even if this latter case is rare in lightweight devices, memory allocation for processes is not fixed and not known a priori. The same problem can appear during an update process of the device firmware. The application threads can have a different MPU configuration after a reboot or an update, thus a different NSID. Keeping the same NSID over reboot is essential for those applications that need to store and use persistent data in the SW. A potential solution is to have informed reboots and updates, *i.e.,* adding more logic to the SW. The SPM could compute the hash of the code area (*i.e.,* R/O region extracted by the MPU configuration) the first time it saw an NSID. It can then compare this hash value with other hashes previously-stored and mapped to an NSID. The SPM can always present the same NSID to the secure services from the same client even after a reboot. A similar approach can be used to address the reboots, where the secure update service is taking care of the transformation of the NSID. However, the update process should

Table 3: Cycles Measurements for the Overhead Experiments

| Test Suite | Vanilla tf-m | tf-m with *Secure Informer* | Overhead |
|---|---|---|---|
| ARM PSA | 91730 cycles | 95011 cycles | 3.5% |
| Regression Test | 381725 cycles | 392857 cycles | 2.9% |

be enhanced to include these maps between the code hash of the previous and subsequent versions of the applications.

An open challenge is that some systems, especially those not restrained with more complex memory management, constantly create and destroy applications, including their executing threads. Two different applications could have the same memory allocation, with the same MPU configuration and NSID. However, the authors do not believe that this scenario will be used in IoT devices implemented with the Cortex-M family, addressing more manageable scenarios.

## 8 RELATED WORK

This section presents the work related to the presented solution. It is essential to highlight that the state-of-the-art regards mainly the devices based on TZ-A (TrustZone for Cortex-A family). TZ-M is still in its early stage of deployment, and to the best of the authors' knowledge, there is no published research on confused deputy problems for such an architecture.

Jiang *et al.*[13] was the first to publish a research paper about the identification problem of non-secure applications in TZ-A. The main objective is to authenticate the caller's identity (self-provided by the caller) so that the NW does not leak private information to malicious non-secure callers. They propose a novel identification schema to detect the legitimacy of non-secure callers who initiate a secure service request. The approach is implemented in the NW kernel. Specifically, the TrustZone driver is used as a bridge between the user application and the secure monitor. Figure 6 presents it graphically. A non-secure application① initially invokes a secure service through the TrustZone driver, which ② computes the hash of its code segment (R/O memory region) and stores it. As already highlighted in this paper, the hash of the code area is unique to each application. Then, the driver ③ sends the request to the secure service. When a malicious non-secure application ④ sends another request using the correct application ID, the driver detects the attack by comparing its code hash with the stored one and denies the request.

Almost at the same time, Zhao *et al.*[43] faced the same problem and proposed a different identity authentication mechanism. Similar to the previously presented approach, the authors proposed to move the hash comparison function into an SW service, as shown in Figure 7. Whenever an application requests a secure service, the identity-obtaining module ① computes the hash of the code segment and uses it as an ID. The TZ driver ② receives the request with the caller's ID, and it ③ sends it to the secure monitor. The secure monitor ④ invokes an identity authentication module that checks the ID (*i.e.,* hash). If the authentication succeeds, the request ⑤ is dispatched to the secure service.

Both solutions stem from the necessity to authenticate the identity of the non-secure callers that in TZ-A, solutions are usually provided by the NW OS without any guarantee. Several attack to TZ-A based implementations of TEE were reported during the years[5][14][15][16]. All these exploits have in common the first step of fooling the TZ driver for a malicious application to communicate with the SW. In TZ-M, such a driver does not exist, and every NW thread can initiate a request to the SW, making the correct implementation of the identification mechanism of paramount importance.

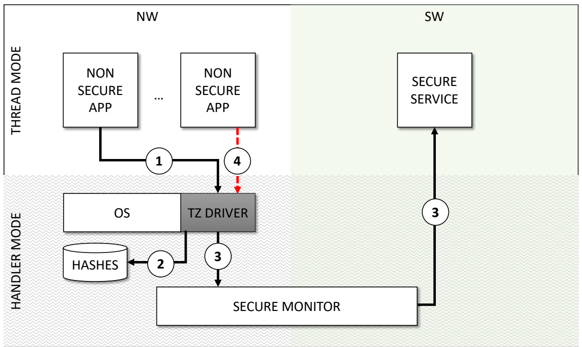

**Figure 6: Simplified view on the Identity Authentication Mechanism proposed by Jiang *et al.*[13]**

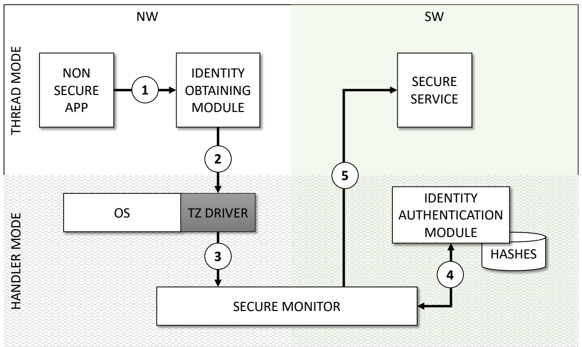

**Figure 7: Simplified view on the Identity Authentication Mechanism proposed by Zhao *et al.*[43]**

A more complex confused deputy attack is presented by Machiry *et al.*[26]. In this case, the attacker's target is not the private values of clients stored in the SW but their isolated memory in the NW. The boomerang attack alters these locations exploiting the SW without tampering with either it or the NW. Whenever an NW application passes memory pointers to a secure service, the secure monitor sanitizes these pointers to avoid requesting a secure service to change another application data. However, this check is faulty because it does not consider "inner pointers", *i.e.,* pointers hidden deep in data structures passed as parameters. This attack is mainly possible because SW developers paid little attention to creating complex API for their custom services by enlarging excessively the TCB that should be kept minimal and simple. TZ-M based software can have the same design flaws. The authors analyzed all the exposed APIs in the reference implementation tf-m, without finding any complex parameter that may allow “inner pointers”. Once TZ-M is widely used by manufacturers, future research may look for such vulnerability in custom implementations of the SW.

*Secure Informer* may be correctly associated to Virtual Machine Introspection (VMI)[39] mechanisms. VMI aims to fill the lack of higher-level knowledge of Guest OS within the Virtual Machine Monitor (VMM). Indeed, TrustZone isolation capabilities have also been exploited as a virtualization technology[6][29][30][25]. Indeed, there is also a VMI library for TZ-A, namely ITZ[9][10]. The main challenge addressed by VMI mechanisms is the translation of the virtual addresses. However, Cortex-M devices make no use of virtual addresses, and *Secure Informer* does not focus on memory content but on the MPU, a duplicated hardware resource of the processor that allows gathering information about the activity of the NW.

## ACKNOWLEDGMENTS

This work is an output of the project STARTS (SecuriTy Assessment of tRusTzone-m based Software), supported by the Luxembourg National Research Fund (FNR) in the framework of the Junior CORE programme under grant number 13624693.

## REFERENCES

[1] 2010. *ARMv7-M Architecture Reference Manual*. Vol. 3. ARM Limited. https://developer.arm.com/documentation/ddi0403/latest

[2] Tiago Alves. 2004. Trustzone: Integrated hardware and software security. *White paper* (2004).

[3] ARM. 2008. *ARM Introduces Software Interface Standard for Cortex Processor-Based Microcontroller*. http://www.arm.com/about/newsroom/23722.php

[4] A ARM. 2009. Security technology building a secure system using trustzone technology (white paper). *ARM Limited* (2009).

[5] Marcel Busch, Johannes Westphal, and Tilo Mueller. 2020. Unearthing the TrustedCore: A Critical Review on Huawei’s Trusted Execution Environment. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[6] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. 2010. Arm trustzone as a virtualization technique in embedded systems. In *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*. 29–42.

[7] Inc. GlobalPlatform. 2018. *TEE System Architecture*. Retrieved December 1, 2021 from https://globalplatform.org/wp-content/uploads/2017/01/GPD_TEE_SystemArch_v1.2_PublicRelease.pdf

[8] Christian Göttel, Pascal Felber, and Valerio Schiavoni. 2019. Developing secure services for IoT with OP-TEE: a first look at performance and usability. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 170–178.

[9] Miguel Guerra, Miguel Correia, Benjamin Taubmann, and Hans P Reiser. 2017. ITZ: an introspection library for ARM TrustZone. In *Proceedings of INFORUM*.

[10] Miguel Guerra, Benjamin Taubmann, Hans P Reiser, Sileshi Yalew, and Miguel Correia. 2018. Introspection for ARM TrustZone with the ITZ Library. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 123–134.

[11] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[12] Antonio Ken Iannillo and Radu State. 2019. A proposal for security assessment of trustzone-m based software. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 126–127.

[13] Hang Jiang, Rui Chang, Lu Ren, Weiyu Dong, Liehui Jiang, and Shuiqiao Yang. 2017. An Effective Authentication for Client Application Using ARM TrustZone. In *International Conference on Information Security Practice and Experience*.

[14] Daniel Komaromy. 2018. *Unbox Your Phone*. Retrieved December 1, 2021 from https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c

[15] laginimaineb. 2016. *Extracting Qualcomm’s KeyMaster Keys - Breaking Android Full Disk Encryption*. Retrieved December 1, 2021 from https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html

[16] laginimaineb. 2016. *TrustZone Kernel Privilege Escalation (CVE-2016-2431)*. Retrieved December 1, 2021 from https://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html

[17] Wenhao Li, Yubin Xia, and Haibo Chen. 2019. Research on arm trustzone. *GetMobile: Mobile Computing and Communications* 22, 3 (2019), 17–22.

[18] Arm Limited. [n. d.]. *PSA Certified*. Retrieved December 1, 2021 from https://www.psacertified.org/

[19] Arm Limited. [n. d.]. *PSA Cryptography API 1.0*. Retrieved December 1, 2021 from https://documentation-service.arm.com/static/5fae7799ca04df4095c1cab0

[20] Arm Limited. [n. d.]. *PSA Storage API 1.0*. Retrieved December 1, 2021 from https://developer.arm.com/-/media/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0087-PSA_Storage_API-1.0.0.pdf

[21] Arm Limited. [n. d.]. *TrustZone for Cortex-M*. Retrieved December 1, 2021 from https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m

[22] Arm Limited. 2021. *Platform Security Model*. Retrieved December 1, 2021 from https://developer.arm.com/documentation/den0128/0100/

[23] Linaro Limited. [n. d.]. *Trusted Firmware M (TF-M)*. Retrieved December 1, 2021 from https://www.trustedfirmware.org/projects/tf-m/

[24] OMTP Limited. 2009. *Advanced Trusted Environment: OMTP TR1*. Retrieved December 1, 2021 from http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

[25] Pierre Lucas, Kevin Chappuis, Michele Paolino, Nicolas Dagieu, and Daniel Raho. 2017. Vosysmonitor, a low latency monitor layer for mixed-criticality systems on armv8-a. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.. In *NDSS*.

[27] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 445–451.

[28] Adeniyi Onasanya and Maher Elshakankiri. 2021. Smart integrated IoT healthcare system for cancer care. *Wireless Networks* 27, 6 (2021), 4297–4312.

[29] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Nuno Cardoso, Mongkol Ekpanyapong, Jorge Cabral, and Adriano Tavares. 2014. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, 1–4.

[30] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. 2016. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE computer architecture letters* 16, 2 (2016), 158–161.

[31] Zephyr Project. 2021. *Zephyr Project Documentation*. Online. https://docs.zephyrproject.org/latest

[32] Juan E Rubio, Rodrigo Roman, and Javier Lopez. 2020. Integration of a threat traceability solution in the industrial internet of things. *IEEE Transactions on Industrial Informatics* 16, 10 (2020), 6575–6583.

[33] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64.

[34] Tohid Shekari, Celine Irvene, Alvaro A Cardenas, and Raheem Beyah. 2021. MaMIoT: Manipulation of Energy Market Leveraging High Wattage IoT Botnets. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1338–1356.

[35] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. 2020. Horizontal Privilege Escalation in Trusted Applications. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[36] Muhammad Azmi Umer, Chuadhry Mujeeb Ahmed, Muhammad Taha Jilani, and Aditya P Mathur. 2021. Attack rules: an adversarial approach to generate attacks for Industrial Control Systems using machine learning. In *Proceedings of the 2th Workshop on CPS&IoT Security and Privacy*. 35–40.

[37] Dalton Cézane Gomes Valadares, Newton Carlos Will, Marco Aurélio Spohn, Danilo Freire de Souza Santos, Angelo Perkusich, and Kyller Costa Gorgonio. 2021. Trusted Execution Environments for Cloud/Fog-based Internet of Things Applications.. In *CLOSER*. 111–121.

[38] Min Woo Woo, JongWhi Lee, and KeeHyun Park. 2018. A reliable IoT system for personal healthcare devices. *Future Generation Computer Systems* 78 (2018), 626–640.

[39] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. 2012. Libvmi: a library for bridging the semantic gap between guest OS and VMM. In *2012 IEEE 12th International Conference on Computer and Information Technology*. IEEE, 549–556.

[40] Joy Qiping Yang, Siyuan Zhou, Duc Van Le, Daren Ho, and Rui Tan. 2021. Improving Quality Control with Industrial AIoT at HP Factories: Experiences and Learned Lessons. In *2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 1–9.

[41] Quanqi Ye, Heng Chuan Tan, Daisuke Mashima, Binbin Chen, and Zbigniew Kalbarczyk. 2021. Position Paper: On Using Trusted Execution Environment to

Secure COTS Devices for Accessing Industrial Control Systems. (2021).

[42] Joseph Yiu. 2015. ARMv8-M architecture technical overview. *ARM white paper* (2015).

[43] Bo Zhao, Yu Xiao, Yuqing Huang, and Xiaoyu Cui. 2017. A private user data protection mechanism in TrustZone architecture based on identity authentication. *Tsinghua Science and Technology* 22, 2 (2017), 218–225.

# A PSA FUNCTIONAL APIS

ARM created Platform Security Architecture (PSA) specification as a standard for IoT security supported by a certification scheme, namely PSA certified[18]. PSA certified is guided by a consortium that includes ARM, Brightsight, CAICT, Prove&Run, Riscure, and UL. It consists of a four-stage framework that IoT designers can use to guarantee security in their products. Among other resources, it freely provides an interface specification to secure services, such as cryptography, secure storage, and attestation. This paper considers the first two services since they manage resources in the SW for non-secure threads.

The PSA storage API [20] provides a key/value storage interface for use with device-protected storage. It includes APIs for both the internal and external protected storage. We present only the protected storage API for simplicity since the internal storage API has identical parameters. The PSA protected storage API consists of 5 functions:

- `psa_ps_set(uid, lenght, p_data, create_flags` creates or modifies a key/value pair with `uid` as a key. The `create_flags` set the properties of the data (*e.g.,* write once);
- `psa_ps_get(uid, offset, size, p_data, p_length)` retrieves the data associated with the provided `uid`, including its length;
- `psa_ps_get_info(uid, p_info)` retrieves the metadata of the data associated with the provided `uid` (*i.e.,* allocated capacity, size, and properties);
- `psa_ps_remove(uid)` removes the key/value pair from the storage;
- `psa_ps_create(uid, capacity, create_flags)` reserves storage for the associated `uid`;

Similarly, the PSA cryptography API [19] defines more complex interfaces for the management of keys and both symmetric and asymmetric cryptography. The functions to manage a cryptographic key are:

- `psa_generate_key(attributes, uid)` creates a new key with the provided attributes (*e.g.,* type, size) in the SW returning its identifier to use in other functions;
- `psa_export_key(uid, data, size, length` reads a key identified by `uid` from the SW;
- `psa_get_key_attributes(uid, attributes)` retrieves the attributed of the key identified by `uid`;
- `psa_copy_key(source_uid, attributes, target_uid` makes a copy of the key identified by `source_uid` returning a new key identified by `target_uid`;
- `psa_destroy_key(uid)` delete the key identified by `uid`.

All the other functions that need a key to perform a cryptographic operation includes the `uid` of a key as first parameter.

In both services, the standard defines `uid` as a 64-bit value. This ID consists of two parts: the first half is the identifier of the requesting entity, while the second half is the resource identifier specified by the caller. The SPM retrieves the caller id and concatenates the two values. This approach allows to easily manage isolation between the identifier namespaces of the services' various clients.

# B CONTEXT MANAGEMENT APIS

ARM developed and open-sourced the Common Microcontroller Software Interface Standard (CMSIS) to help the industry in standardization [3]. CMSIS is a set of tools, APIs, frameworks, and workflows that focuses on software re-use and support developers to bring new applications to market quicker. In particular, CMSIS-Core(M) creates a vendor-independent hardware abstraction layer for Cortex-M processor-based devices and supports the TrustZone-M extension. It defines APIs to consistently manage thread contexts when switching between the SW and NW. The OS in the NW notifies the SW each time there is a thread context switch to manage the secure stack space and identify the non-secure caller. This API requires that the SW must implement a secure context management system that is accessed through five secure functions:

- `TZ_InitContextSystem_S`: prepares the context management system;
- `TZ_AllocModuleContext_S`: informs the SW that a new thread has been created in the NW that will eventually make a secure call;
- `TZ_FreeModuleContext_S`: frees the secure resources allocated for a thread that has been terminated in the NW;
- `TZ_LoadContext_S`: notifies the SW of the start of the execution of a specific thread in the NW;
- `TZ_StoreContext_S`: notifies the SW of the end of the execution of a specific thread in the NW;

By modifying the scheduler in the NW, whenever the NW makes a call to a secure function, the SW knows which non-secure thread is actually the caller and can differentiate its policy accordingly. However, this mechanism requires changes to the NW code and introduces two calls to the SW for each context switch. These calls are required even if the running non-secure thread will never call the SW before its next context switch.