ShieLD: Shielding Cross-zone Communication within Limited-resourced IoT Devices running Vulnerable Software Stack

Anum Khurshid, Sileshi Demesie Yalew, Mudassar Aslam and Shahid Raza

Abstract—Securing IoT devices is gaining attention as the security risks associated with these devices increase rapidly. TrustZone-M, a Trusted Execution Environment (TEE) for Cortex-M processors, ensures stronger security within an IoT device by allowing isolated execution of security-critical operations, without trusting the entire software stack. However, TrustZone-M does not guarantee secure cross-world communication between applications in the Normal and Secure worlds. The cryptographic protection of the communication channel is an obvious solution; however, within a low-power IoT device, it incurs high overhead if applied to each cross-world message exchange. We present ShieLD, a framework that enables a secure communication channel between the two TrustZone-M worlds by leveraging the Memory Protection Unit (MPU). ShieLD guarantees confidentiality, integrity and authentication services without requiring any cryptographic operations. We implement and evaluate ShieLD using a Musca-A test chip board with Cortex-M33 that supports TrustZone-M. Our empirical evaluation shows, among other gains, the cross-zone communication protected with ShieLD is *5 times* faster than the conventional crypto-based communication.

Index Terms—IoT, IoT Security, Trusted Execution Environments, TEE, Cortex-M, TrustZone, TrustZone-M.

1 Introduction

With the increasing popularity of IoT devices and the increasing number of security and privacy risks associated with these devices [1], IoT device security has drawn significant attention from both academia and industry. These IoT devices have limited storage, memory and processing capabilities but are used in critical infrastructure such as e-health, smart energy and industry 4.0, and make up the biggest share in the envisioned billions of IoT devices. As a result, ARM has released TrustZone into Cortex-M processors (Cortex-M23 and Cortex-M33) – TrustZone-M [2], which brings Trusted Execution Environment (TEE) also to battery-powered resource-constrained IoT devices.

TrustZone technology is a security extension incorporated into ARM processors. It provides a mechanism to partition system on chip (SoC) resources (e.g., memory, peripherals, etc.) in two worlds (or zones): the *Secure world* and the *Normal world*. The Secure world has a small code-base and runs only security-critical operations, whereas the Normal world runs all other apps. Keeping the Secure world's source code minimal is important for having a trusted and formally verified TEE code-base [3]. It is impossible to guarantee a bug-free software stack within an IoT device running multiple apps, third-party libraries, drivers, and protocols. Therefore, a TEE becomes inevitable for security-critical operations such as secure boot, crypto operations, software/firmware update, remote attestation, and handling authentication peripherals (i.e., fingerprint reader, biometric scanner, etc.).

• *A. Khurshid, S. D. Yalew, M. Aslam and S. Raza are with the Cybersecurity Unit at RISE Research Institutes of Sweden, Stockholm, Sweden. E-mail: {anum.khurshid, sileshi.demesie.yalew, mudassar.aslam, shahid.raza}@ri.se*

Manuscript revised December 30, 2021

Challenge: Although TrustZone-M provides a hardware-based TEE, which effectively isolates security-critical operations from untrusted software components, it lacks mechanisms for secure cross-world communication, i.e., between the TEE and the untrusted environment in an IoT device. TrustZone-M allows direct function calls between the two worlds, i.e., an app in the Normal world can directly call a secure function in the Secure world using an API call; however, TrustZone does not provide authentication mechanism for the incoming call from a Normal world app. This allows an attacker to run malicious code to falsify calls and repeatedly pass maliciously-crafted messages to find vulnerabilities in secure software. This weakness was exploited in certain Motorola phones with a TrustZone-based TEE [1]. Moreover, TrustZone-M does not provide a mechanism to protect messages transferred between the two worlds. Most TrustZone-based TEEs use a shared memory area allocated in the Normal world memory region, as a channel for transmission of messages. Unfortunately, this channel is vulnerable to attacks if the Normal world is compromised (Figure 1). As a result, attackers could intercept and manipulate messages transferred through this channel. Hence, malicious apps with privileged access in the Normal world can compromise the *confidentiality*, *integrity* and *authentication* of communication between the two TrustZone worlds. High-end mobile devices with TrustZone-A support may use traditional crypto-based communication approaches like using shared session key based encryption and MAC operations [4]. However, such solutions are too expensive, in terms of computation and power consumption, when used in resource-constrained IoT devices that are expected to run on batteries for months or years.

1. http://www.cvedetails.com/cve/CVE-2013-3051/

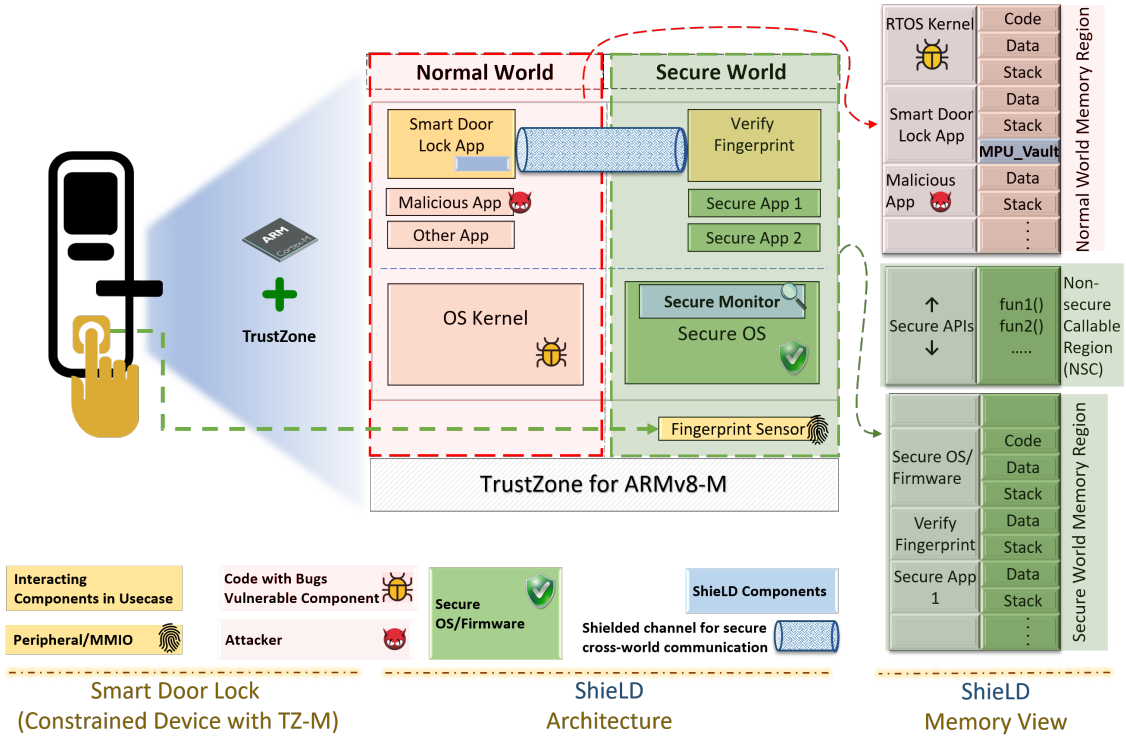


Fig. 1: Overview of ShieLD Architecture that introduces cross-world/cross-zone secure communication in a constrained IoT device for ARM TrustZone-M based TEE. While the apps in the Secure world (TEE) are safe, Normal world OS can have **vulnerabilities** allowing **malicious apps** to compromise the communication between a legitimate app (IoT App) in Normal world and its corresponding secure app in the Secure world. ShieLD's Secure Monitor uses **MPU_Vault** for shielded cross-world secure communication.

ShieLD: In this paper, we present the design, implementation and evaluation of ShieLD, a framework for securing cross-zone/cross-world communication in IoT devices that support TrustZone-M. ShieLD provides the ability for legitimate apps to use a protected shared memory zone, we call it *MPU_Vault*, as a medium to establish a secure channel for exchanging messages between a Normal world app and the secure services in the Secure world, without using cryptographic operations to protect message confidentiality. ShieLD protects the *MPU_Vault* using the Memory Protection Unit (MPU), which limits the access permissions of a Normal world memory region in terms of accessibility by a certain CPU execution context. We leverage the MPU in a novel way to establish a secure communication channel between the two worlds, which to the best of our knowledge is the first attempt in not just solving the secure communication problem for this class of devices but also in using the MPU for this purpose. To establish ShieLD communication framework; challenges like task legitimacy, task interruption and termination, task code integrity, and privileged software integrity were also identified and resolved.

Usecase Scenario: Constrained IoT devices that have security critical sensors involving biometric data, such as a fingerprint sensor can utilize TrustZone-M for trusted execution. The firmware of such sensors is placed in the Secure world to protect its functionality. On the other hand, IoT apps are added by developers in the Normal world which use the sensors by calling secure functions in the Secure world (as depicted in Figure 1). In this scenario, the security guarantees of TrustZone could be compromised due to vulnerable cross-world communication channel. An attacker could intercept and manipulate messages transferred between worlds to analyze the behavior of secure firmware [2], like was possible in the case of a TrustZone-based digital rights management solution developed by Discretix [5]. ShieLD avoids such an attack by enabling an MPU_Vault that is only accessible by the legitimate app in the Normal world. A malicious app in the Normal world can abuse the communication channel to arbitrarily send a crafted message to the secure service in the Secure world to launch different attacks, as in the recent case of Boomerang attack [6]. In order to avoid this attack, ShieLD authenticates the requesting app thereby preventing malicious apps to make a call with malicious input data.

Contributions: The main contributions of this paper are:

- *Secure communication channel design*: We propose ShieLD, a framework to build secure communication between the two worlds of TrustZone available on the resource-constrained IoT devices. We present a lightweight message protection scheme using the MPU, while providing similar security assurances as provided by typical crypto operations.
- *Implementation and Evaluation*: We demonstrate the ShieLD practicality via proof–of–concept implementation and evaluate its performance using the

2. https://sensepost.com/blog/2013/a-software-level-analysis-of-trustzone-os-and-trustlets-in-samsung-galaxy-phone/

TrustZone-M (on Musca–A2 Test Chip Board). In our experimental evaluation, ShieLD achieved nearly *5x* lower execution time as compared to message encryption approach which provides similar security guarantees.

The rest of the paper is structured as follows. Related work is discussed in Section 2. In Section 3, we discuss in detail and link the technologies and building blocks needed to understand ShieLD. Section 4 describes the threat model. Section 5 explains a usecase of ShieLD followed by the ShieLD design in Section 6. We discuss the design and architecture of related work SeCReT in comparison to ShieLD in Section 7. The implementation of a proof-of-concept of ShieLD is explained in Section 8. Section 9 follows with performance evaluation and results. We bring an important discussion about scalability of ShieLD in Section 10. The security analysis of the proposed design is discussed in Section 11 and limitations in Section 12. We conclude the paper with Section 13.

2 Related Work

We categorize the related work into three major areas: TEE, cross-world communication in TEEs and MPU-based isolation techniques.

2.1 Trusted Execution Environment (TEE)

A TEE ensures that sensitive code and data are stored, processed and protected in an isolated, trusted environment. Researchers have provided TEEs based on virtualization, e.g., Terra [7], that rely on the hypervisor to create and manage TEEs without any modification to existing hardware. Others have leveraged the Trusted Computing Group's Trusted Platform Module (TPM) to provide higher assurance, e.g., [8], [9]. More recently hardware extensions are also proposed for the creation of TEEs. For example, Intel's Software Guard Extensions (SGX) [10], [11] is an architecture designed to support enclaves that are isolated execution environments for code and data within an application space. ARM TrustZone [12] is a widely adopted TEE in embedded systems and mobile devices. vTZ [13] is designed aiming at virtualizing TrustZone in the ARM architecture. Iso-X [14] offers a higher memory allocation flexibility than SGX; however, it is not supported by current processors. Keystone [15] is another newly emerging open-source TEE framework, providing customizable software enclaves for RISC-V. These TEEs are targeting high-end devices and are not suitable for low-power IoT devices.

Since the introduction of TrustZone-M [2] in Cortex-M family, resource-constrained devices have started implementing systems and solutions utilizing TrustZone-M [16], [17]. Securing the process of firmware update is a crucial phase in IoT life-cycle. ASSURED [17] extends TUF [18], which is an update framework resilient to key compromise. SAFES [16] is an architecture for self-measuring code integrity in sand-boxed environment at each I/O event (running on embedded devices) implemented on a Cortex-M33 processor with TrustZone enabled. [19] is a Trusted-M-assisted virtualization architecture featuring the ARM Musca-A platform. CoreLockr-TZ [20] provides a platform for IoT devices that allows apps running in the Normal world to access secure services by making API calls. Such recent research shows the usefulness of TrustZone-M for constrained devices.

2.2 Cross-world Communication

The security of cross-world communication channel is of ultimate importance in ARM Cortex-A, Intel and other architectures as well. Several solutions attempt to provide a secure communication infrastructure between partitions. LTZVisor [21] and TZ-VirtIO [22] are systems based on ARM Cortex-A utilizing TrustZone to setup hardware-assisted virtualization of the system into secure and non-secure partitions. VM communication is established using VirtIO [23] and takes place such that every exchange of data between partitions is carried out through a shared memory (non-secure) which is overseen by the trusted hypervisor. InkTag [24] provides a virtualization architecture supporting Intel's VMX hardware virtualization support. It allows the applications to define access policies for its own secure files, so in the event of corruption of privileged Operating System, the files remain inaccessible. The communication between untrusted OS and trusted applications is maintained by InkTag using HAP page tables [24]. Virtual Ghost [25] uses compiler instrumentation like sandboxing to protect the code and metadata of a process from the operating system. SeCReT [4] is a recent and relevant approach aiming to solve the cross-world communication issue for Cortex-A family of microprocessors. It uses session keys protected by TrustZone to protect code and data of the communicating processes. TrustZone has been very recently introduced in the Cortex-M family and to the best of our knowledge, no work is done on secure cross-world communication in TrustZone-M. The above mentioned solutions for cross-world communication are vulnerable to man-in-the-middle attacks, malicious RTOS and hardware tampering [21], [22], [24], [25]. ShieLD eliminates the risk of former two security threats by ensuring the security of cross-world communication for data exchange between processes.

2.3 MPU-based Isolation Techniques

The ARM Memory Protection Unit (MPU) is an optional programmable unit preferably used as a memory isolation mechanism for lightweight architectures that do not require complex memory management (like low-end devices based on Cortex M3/M4/M23/M33) [26]. Several architectures utilize the MPU to provide thread isolation per task, allocate on-demand stack, establish a virtualization layer to protect critical software components. uVisor [27] utilizes the MPU to isolate groups of tasks and threads and allows them to access resources based on pre-defined access permissions. Tock OS [28] is another open source OS targeting Cortex M3/M4 processor architectures and utilizing the MPU to compartmentalize the system into three security levels. A virtualization layer setup between hardware components and application software using the MPU to prevent misuse of hardware components by third-party applications is presented in [29]. M2MON utilizes the MPU on Unmanned Vehicle (UV) peripherals to defend against several attacks by monitoring the I/O activity [30]. ACES is another proposal

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3147262, IEEE Transactions on Dependable and Secure Computing

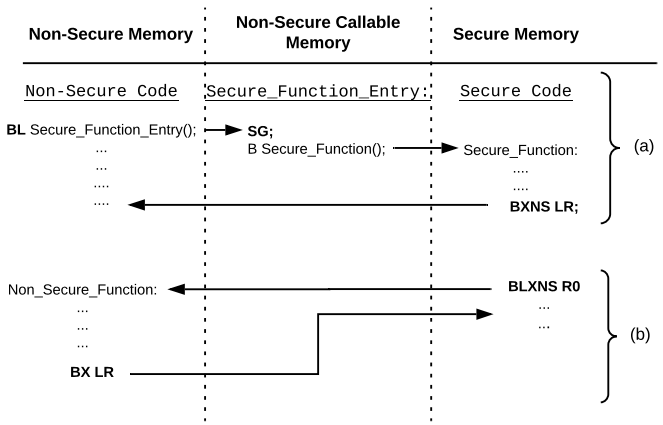


Fig. 2: Secure and Normal world switch when (a) a non-secure code calls secure function via a secure function entry in NSC memory region; (b) a secure code calls a non-secure function.

that uses the MPU to create isolation and compartmentalization of applications using developer-specified policies in bare-metal embedded systems. It uses an LLVM-based compiler to automatically create an instrumented binary inferred from the developer policies [31]. A combined utilization of off-line static analysis on the firmware to identify process memory; and the MPU to isolate the processes dynamically is demonstrated on real-time microcontrollers in [32]. μRAI is another compiler-based mitigation technique, to prevent control-flow hijacking attacks in MCUs that do not have access to special hardware like TEE. It also uses the MPU to enforce Data Execution Prevention (DEP) [33]. μXOM is a novel technique that realises eXecute-Only Memory (XOM) for Cortex-M processors using unprivileged memory instructions and MPU [34]. However, using the MPU for establishing a shielded cross-world communication channel is a novel mechanism.

3 Technologies/Building Blocks

This section provides background information about the technologies underlying the design of ShieLD: *Cortex-M architecture*, *ARM TrustZone* and *Memory Protection Unit (MPU)*.

3.1 Cortex-A and Cortex-M Architecture

The ARM Cortex-A is a series of application processors providing solutions for devices that require a rich processing environment (with commodity OS). They support A64 and A32 instruction sets. In order to undertake complex compute tasks, supporting multiple software applications and modes of operation, it supports virtual memory system architecture with a separation between OS space and space for application programs. On the other hand, the Cortex-M is a series of microcontroller processors that are programmed either bare metal (without libraries) or linked with some libraries that could provide OS-like features. Cortex-M based devices support the T32 instruction set, which is a subset of the A32 instruction set. The processor does not offer complex memory management (no MMUs), cache and often no FPUs either. Moreover, the A and M profile architectures also differ in the way TrustZone security extensions operate on them. One of the differences is the switching mechanism between secure and non-secure state. In TrustZone-A, the transition takes place with an SMC instruction which is implemented in software. In TrustZone-M the transition is implemented in hardware with SG instruction and is hence faster. The security state of a process in TrustZone-A relies on the value of the NS bit, whereas in TrustZone-M, the security state is determined by whether the code being executed resides in the secure memory or the non-secure memory as per the memory map.

3.2 ARM TrustZone-M

ARM TrustZone is a set of hardware security extensions incorporated into recent ARM processors (such as Cortex A8, Cortex A9, Cortex M33, Cortex M23) [2], [35]. TrustZone allows to compartmentalize the device hardware (Flash, SRAM, peripherals) and software resources in two security states or worlds: the Secure or the trusted world (also called TEE) that has minimal code-base and runs only security-critical operations, and the Normal world that has a rich code-base (therefore is vulnerable and unsecure) and runs most user-level applications. Malicious code in the Normal world cannot affect the integrity and confidentiality of the code and data running in the TEE.

TrustZone-M provides similar hardware-based isolation guarantees as conventional TrustZone for Cortex-A family (TrustZone-A) but, unlike TrustZone-A, the transition (or the context switch) between the two worlds depends on the setup of memory map, without the need to enter the secure monitor mode. This design feature makes TrustZone-M more energy-efficient, hence suitable for low-powered IoT devices. Saving and restoring the system context before and after the transition can be handled by the Secure world [36]. In TrustZone-M enabled MCUs, a system designer can define a particular device memory region either as secure or non-secure by configuring the Security Attribution Unit (*SAU*). An SAU is programmable in the Secure world using memory-mapped SAU registers. A system designer can also use the Implementation Defined Attribution Unit (*IDAU*) to define a fixed memory map [2]. When the processor is executing code in the secure memory region, it is in the Secure world; otherwise it is in the Normal world. The system designer can further divide the secure memory region into two sections, i.e., secure and non-secure callable (*NSC*). The secure section contains trusted code and data (including secure stack and heap, and any other secure data) and the non-secure callable (*NSC*) region contains entry functions (i.e., branch instructions to the actual secure code in the secure memory). The Secure world can access non-secure memory region, but the reverse is not possible and is protected by the TrustZone.

3.2.1 Calling Secure Function

As depicted in Figure 2 (a), a non-secure code can interact with a secure function in the Secure world using a direct function call through entry points/APIs in the *NSC*. The first instruction of the entry point in NSC should be a

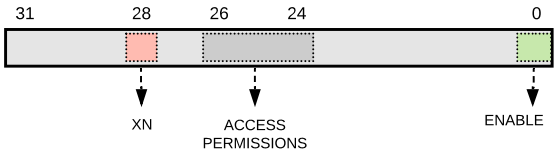


Fig. 3: Region Attribute and Size Register (RASR).

secure gateway (*SG*) instruction, which is a CPU instruction introduced to prevent non-secure code from branching into an invalid entry point in secure code. The call to a secure function completes by returning to non-secure code using a new CPU instruction, known as Branch with exchange to Non-secure state (*BXNS*). The secure function returns to non-secure code by performing `BXNS LR`, which behaves like a return through `BX LR`, but additionally switches the processor to non-secure state.

3.2.2 Calling Non-Secure Function

The secure code can make a call to non-secure function directly using a CPU instruction, known as Branch with link and exchange to Non-secure state (*BLXNS*) (as shown in Figure 2 (b)). The execution of `BLXNS R0` (`R0` contains the address of non-secure function that is being called) switches the processor from secure state to non-secure state [37]. Information leakage from secure state to non-secure state may occur through parts of the system that are not banked between the two security states. Therefore, the return address and some processor state information should be pushed into secure stack before switching to non-secure state. At the same time, the return address on `LR` is set to a special value `FNC_RETURN`. The non-secure function then completes with branch to `FNC_RETURN` address by performing `BX LR`. This unstacks the actual return address from secure stack and the control flow is transferred back to the calling function in the Secure world.

3.3 MPU-based Memory Protection

All Cortex-M processors except Cortex-M0 have an MPU, which is a programmable block inside the processor that can be used to restrict access to a memory region by dividing the entire memory space (including Flash, SRAM) into a number of MPU regions and assigning access permissions to each region. The MPU can be configured to support 8 or 16 regions by privileged software using a series of 32-bit memory mapped registers [26]. For example, the Region Attribute and Size Register (`RASR`) is used to define the region size and memory attributes of an MPU region (as shown in Figure 3). Privileged software such as an OS kernel, can change the MPU region setting at run-time based on the process being executed. The processor generates memory management (`MemManage`) fault exception or `HardFault` if any illegal access to MPU regions (including instruction fetches and data accesses) is detected [38]. The Cortex-M23 and Cortex-M33 can have up to two MPUs if the TrustZone security extension is implemented and enabled (one for Secure world and one for Normal world). The secure and non-secure MPU can be configured independently with a different number of MPU regions to protect memory for the associated security domain. Secure software running in the Secure world can access and configure a non-secure MPU using an alias address.

TABLE 1: Encoding of AP Field for various access permissions configuration.

AP	Access Privileges		Descriptions
Value	Privileged	Unprivileged	
000	No Access	No Access	Any access attempt generates a fault
001	RW	No Access	Read & write access in priv. level only
010	RW	RO	Write in unpriv. level generates fault
011	RW	RW	Read & write, by priv. & unpriv. level
100	Reserved	Reserved	Reserved
101	RO	No Access	Read access, by priv. level only
110	RO	RO	Read access by priv. & unpriv. level
111	RO	RO	Read access by priv. & unpriv. level

4 Threat Model and Goals

Our primary assumption is that an attacker with privileged access can compromise the software (both applications and the operating system) running in the Normal world to mount various attacks on secure software running in the Secure world. The attacker could modify the legitimate applications' code to repeatedly call secure functions in the Secure world with maliciously crafted parameters. We also assume the attacker could intercept and analyze messages exchanged between the two worlds of TrustZone-M via a shared memory. The goal of the attacker, for example, could be analyzing the behaviour of secure software for bug hunting. We also assume an attacker with privileged access could modify the configuration of the Normal world MPU to change the access permissions of Normal world memory regions protected by its MPU. However, we assume that the attacker cannot directly access the Secure world memory region because of hardware-enforced memory protection mechanisms such as SAU. At hardware level, we assume ShieLD runs on resource-constrained IoT devices that support ARM TrustZone-M hardware protection, which is implemented correctly and is not compromised.

Based on the above threat model and assumptions, we design ShieLD with the following security goals: *(i)* no software component running in the Normal world besides legitimate ones can access critical resources in the Secure world; *(ii)* during cross-world communication, only legitimate software component can access MPU_Vault; and *(iii)* when a legitimate Normal world application is successfully completed or unexpectedly interrupted, the MPU_Vault being used is protected against third-party accesses.

5 Smart Door Lock - A ShieLD Usecase

In this section, we present a real-world scenario where the need for our proposed mechanism would arise. A *Smart Door Lock* with an embedded fingerprint sensor has a critical function of biometric authentication which is considered *"very effective"* by 92 percent of enterprises according to a recent Ping Identity survey [39]. However, there remains a high risk of unrecoverable loss if biometric data is compromised; because, while a leaked password can be changed, biometric data is immutable and thus cannot be changed.

A TrustZone-enabled *Smart Door Lock*, considered as a usecase example (Figure 4), can protect users' biometric data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3147262, IEEE Transactions on Dependable and Secure Computing

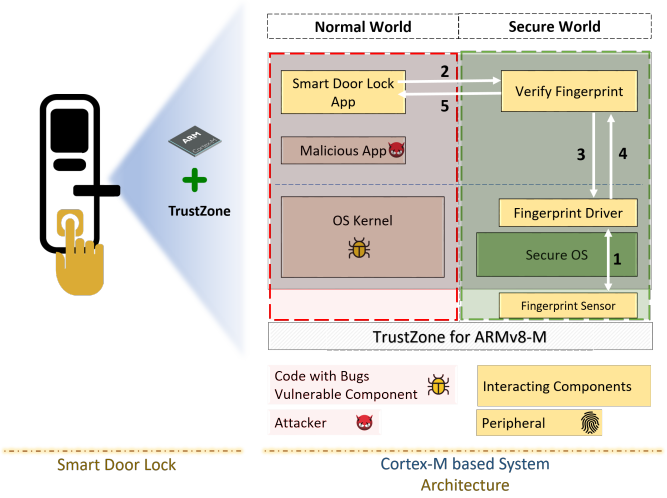


Fig. 4: A TrustZone enabled *Smart Door Lock* is considered as a usecase example. 1) The fingerprint sensor (peripheral) mounted to Secure world detects a fingerprint input signal which is handled by its firmware in the Secure world memory region, 2) a *Smart Door Lock App* hosted in the Normal world requests the fingerprint verification secure service, 3) the service invokes the fingerprint driver for the fingerprint that was received, 4) on receiving the fingerprint, the Verify Fingerprint secure service verifies it, 5) a response is returned to the Door Lock App based on whether the fingerprint was verified.

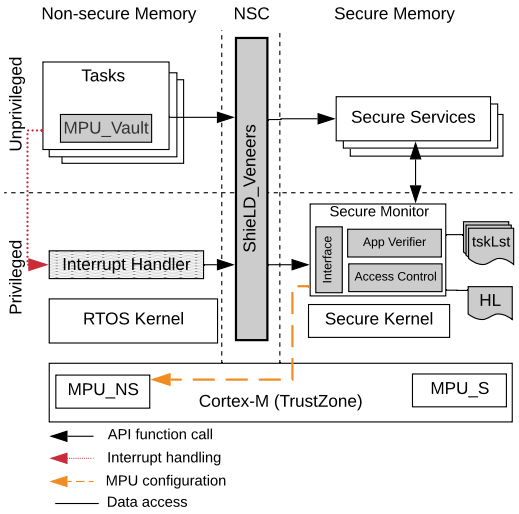


Fig. 5: High-level architecture of ShieLD. The shaded boxes are the new components that we propose and implement for ShieLD, while the pattern-filled box is an existing system component that we extend for ShieLD.

by 1) mounting the fingerprint sensor (peripheral) to Secure world, and 2) by keeping its firmware in the Secure world memory region to prevent any alterations in its behavior. The fingerprint scanning and authentication services provided by the fingerprint scanner can then be securely used by the applications hosted in the Normal world. The *Smart Door Lock App* placed in the Normal world communicates with the secure services dealing with secure peripherals (i.e. fingerprint sensor). The lifecycle of the application begins with (i) *Fingerprint Sensor* receiving a fingerprint, (ii) based on this event, the *Smart Door Lock App* requests the *Verify Fingerprint* secure service, (iii) the *Verify Fingerprint* service verifies the fingerprint received by the *Fingerprint Sensor*, (iv) based on the verification result, an appropriate response is sent either allowing access or denying it. (v) The last step of the application lifecycle is deletion of task context from the Task Control Block.

There are two main security threats that we consider in this paper. First, the applications in the Normal world can reverse engineer the behavior of the Secure world application by exploiting the possibility to call any Secure world function without authentication [4]. Second, the large codebase of the Normal world exposes it to the vulnerabilities which could be exploited by the attackers to intercept the communication between a Normal world application and Secure world code (e.g. firmware of fingerprint sensor).

While the *Smart Door Lock* with embedded fingerprint reader is presented as an example, the threat scenario focused in this paper applies to all usecases where TrustZone is used to ensure the security of critical functions, such as, e-health devices, industrial sensors, critical infrastructures.

6 ShieLD Architecture and Design

Keeping the above mentioned scenario as motivation, we present the ShieLD architecture and the details of all the ShieLD operations for enabling and protecting MPU_Vault in this section. Figure 5 depicts the high level architecture of ShieLD, highlighting its components and basic interactions. The shaded boxes represent ShieLD's implemented components, while the pattern-filled box represents an existing system component which required modifications to fulfill ShieLD requirements. The Normal world runs the applications usually referred to as *tasks* in embedded system, and a modified version of an *interrupt handler* on top of a Real-Time Operating System (*RTOS*) kernel. The Secure world runs the ShieLD components: *secure monitor*, *app verifier* and *access control* that are designed to implement security mechanisms. A small underlying *secure kernel* provides basic OS functions for software running in that world (e.g., process management, file access, and memory management).

The *secure monitor* implements the core functionalities of ShieLD. It is comprised of three main modules: *interface*, *app verifier*, and *access control*. The *interface* module acts as a gateway between the Normal world and other *secure monitor*'s modules. It receives and handles all API function calls to the *secure monitor* modules from the Normal world components via *ShieLD _veneers*. The *app verifier* is responsible for authenticating a task accessing Secure world resources. The *access control* module manages and protects access to MPU_Vault for secure cross-world message transmission. We discuss each component in detail and the security mechanisms provided by ShieLD to enable secure communication between the worlds of TrustZone-M in the next sections. We begin with elaborating the steps of cross-zone communication process, then discuss the protection of the MPU_Vault and

finally go into details of how we maintain the integrity of ShieLD components in the system.

6.1 Cross-zone Communication Process

ShieLD uses MPU_Vault, as a secure medium for the transmission of messages in cross-world communication. In this section, we discuss how this MPU_Vault is setup/allocated in a three phased process: *allocation request*, *task authentication*, and *MPU_Vault setup*.

6.1.1 Allocation Request Phase

The MPU_Vault setup is initiated by the tasks in the Normal world. A task that needs access to a service in the Secure world, first allocates an optimal block of memory area in the Normal world, and then sends a request to the *secure monitor* for its MPU_Vault setup. This is done using the ShieLD _veneer call shown in the following listing; the arguments for this call are passed to the *secure monitor* using general purpose registers (`R0-R3`).

```
setupMPUVault(int tskID,          // in
              char *buffer,       // in
              int buffer_size,    // in
              int *success_code)  // out
```

The `tskID`, in the function, is a unique task ID (a task can be identified by a pointer to its `Task Control Block` (`TCB`) in RTOS kernel, which would always be unique to a specific task), `buffer` and `buffer_size` are the physical address and the length of the allocated memory region, respectively. If this function call is successful, it returns `success_code` with the value of zero. On an error (e.g., if the requesting task is not legitimate), it returns -1. Listing 1 shows the code snippet for MPU_Vault allocation request made using `setupMPU_Vault` function.

```
tskID = ...
size_of_inputs=compute_size(input_descriptors);
size_of_outputs=compute_size(output_descriptors);
buffer_size=max(size_of_inputs, size_of_outputs);
buffer=allocate_buffer(buffer_size);
success_code=0;
setupMPU_Vault(tskID, buffer, buffer_size,
               success_code);
if (success_code < 0) {
   process_error(); }
```

Listing 1: MPU_Vault Allocation Request

6.1.2 Task Authentication Phase

The second phase is authentication of the requesting task to verify if the task is permitted to access Secure world resources. When the *interface* module receives a call/request for MPU_Vault allocation from a task in the Normal world, it invokes *app verifier* module to verify the legitimacy of the requesting task. To this end, an access control list containing tasks and the secure services they are allowed to access is required.

We assume that ShieLD is provided by an IoT device vendor as a part of secure firmware. Moreover, we assume that the device vendor also provides a certificate called *hash list* (*HL* in Figure 5), that describes a list of tasks that are granted access to Secure world services. These assumptions are based on the fact that low-power IoT devices are designed for special applications (such as industrial control systems and automotive systems) and do not run many software modules provided by different vendors. Therefore, legitimate tasks with their access to secure services can be predefined during device manufacturing. HL contains a list of hashes/digests of the tasks and unique identifiers (ID) of corresponding secure services to which tasks are granted access. The *hash list* can be updated during firmware updates if new applications are added or existing applications are removed. Secure update mechanisms like ASSURED [17] are recently proposed for constrained platforms and further studies show that it is possible to create secure and standard-compliant update mechanisms [40] even for this class of devices. HL is stored in the Secure world, isolated from the Normal world. Furthermore, in case of a software update, we assume that a new HL, with updated hashes, will be provisioned in the secure update process along with the new images of the updated components.

When the app requests creation of the MPU_Vault, the *app verifier* module calculates the hash of the task code using a collision-resistant hash function and compares it against hash values present in the HL. To calculate a hash value of a running task, the start and end address of the task code are required. The *app verifier* can retrieve those addresses from (`TCB`) of the task, which holds all the information about execution context of the task. If the calculated hash value matches the corresponding hash value in the HL, the *app verifier* creates a task context (`tskContext`) data structure in the Secure world and maintains task context information, such as a unique identifier of the task (`tskID`), the task code memory space, etc. The *app verifier* creates the `tskContext` for each active task that is allowed to access secure services and links them to a resulting task context list (`tskContextList`) (see *tskLst* in Figure 5) which is used to control access to MPU_Vault. The integrity of the task is hence verified. To ensure that the task code is not modified between calls and interrupts, the task's code, data and stack region are also protected by the MPU by configuring them as RO. Later, when the task requests access to an MPU_Vault, the *app verifier* retrieves the `tskID` of the current task from the (`TCB`), and checks the *tskLst* for the the tasks allowed to access the MPU_Vault. Another possible solution could be to verify the integrity of the task requesting MPU_Vault access on every access request using hash comparison. The protection of task using the MPU not only ensures the integrity of the task, but is also relatively lightweight in terms of operational cost as compared to hash calculation and verification on every access.

6.1.3 MPU_Vault Setup Phase

The third and final phase to enable secure cross-world communication is to setup MPU_Vault. After authenticating the requesting task, *access control* module configures an MPU-protected memory region associated with the requested buffer address and size, and sets the access permission of that region to Read/Write (RW) by configuring *MPU_NS* registers.

In Figure 6, we illustrate an example how system memory regions could be configured using MPU to protect MPU_Vault. The figure shows two tasks (Task1 and Task2) running on top of an RTOS kernel. The system memory is organized into four MPU regions with associated access

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3147262, IEEE Transactions on Dependable and Secure Computing

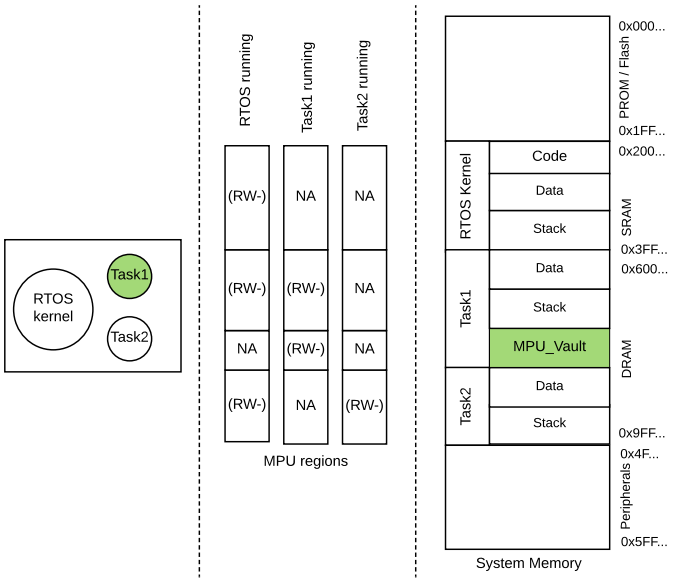


Fig. 6: Example MPU-based memory protection, defining various memory regions and access permissions for two tasks Task1 (with MPU_Vault) and Task2 as well as an RTOS kernel.

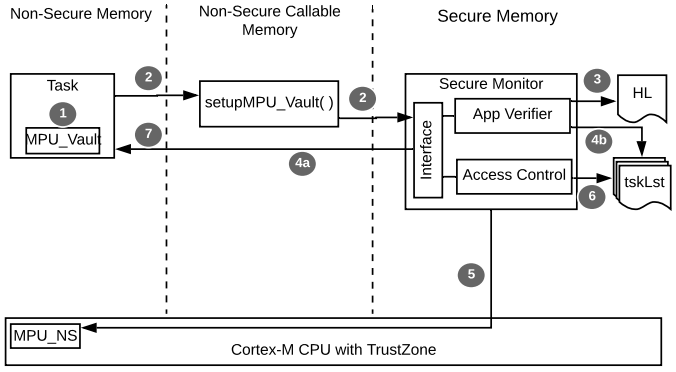


Fig. 7: MPU_Vault setup: (1) the Task in the Normal world allocates a shared MPU_Vault. (2) the Task sends a request for MPU_Vault setup to the Interface module via the *ShieLD _veneers* in the NSC memory region. (3) the *App Verifier* validates the Task if it is legitimate to get access to secure services. (4a) the *Interface* module returns an error message if the Task is not legitimate. (4b) the *App Verifier* module creates *tskContext* if the Task is legitimate. (5) the *Access Control* module sets up an MPU region for MPU_Vault by configuring MPU_NS. (6) the *Access Control* module adds the MPU_Vault address and sets the `MPUVaultEnabled` flag in *tskContext*. (7) finally, the control flow moves to the Task.

permissions (as discussed in Table 1). An MPU-protected memory region (depicted as MPU_Vault in Figure 6) is exclusively accessible to Task1. Considering an example scenario, when Task1 is running, the MPU configuration would trigger a fault if Task2 or RTOS kernel tries to access the *MPU_Vault*.

Once the MPU region for MPU_Vault is configured, the *access control* module saves the address and `MPUVault_RegionNumber` of MPU_Vault in the task's `tskContext` for future reference and sets `MPUVaultEnabled` flag, indicating an MPU_Vault is currently enabled. The *access control* module also enables MPU regions for protecting the requesting task's code memory, data and stack region and sets the access permission as read-only (RO) to prevent modification of the task code and prevent arbitrary execution of code by malicious tasks and OS. To make the requesting task entirely trustworthy and ensure that any code executing from its assigned memory region is not compromised, its data memory and stack need to be protected. If dynamic memory allocation or heap memory is supported on the target system, and is implemented as per the Platform Security Architecture (PSA) specifications [41], ShieLD design would need minor updates to incorporate heap protection for the task as well. As of now, heap memory is not supported in the target platform we have used for the design of ShieLD, hence it is not included in the design contributions.

Finally, the *access control* module transfers the control flow to the requesting task by executing `BXNS LR` instruction (see Section 3.2). Before returning to the task, the *interface* module first checks the value of Link Register (`LR`) to see if the return address is in task's memory range. The steps for the MPU_Vault setup is summarized in Figure 7.

6.2 MPU_Vault Protection Mechanisms

In this section we discuss the protection of the MPU_Vault during software and hardware interrupts, execution context switch and on task termination. An MPU_Vault is associated to a task existing in the system and should be accessible only when its associated task is executing. However, there are several circumstances when the task gets suspended or terminated. As a result, an attacker with privileged access might compromise the suspended or terminated task's MPU_Vault if it is still accessible. To avoid such an attack, ShieLD provides the following mechanisms.

6.2.1 Interrupt Control

Tasks are frequently interrupted by software and hardware interrupts (e.g., to react to events like receiving input from sensors). Whenever an interrupt occurs, the processor stops the current task and starts the execution of a predefined routine, known as *Interrupt Handler* or *Interrupt Service Routine* (ISR), which handles the interrupt. When the ISR is complete, CPU continues to execute the interrupted task from the point where it was interrupted. This interrupt handling process allows an attacker compromising the ISR or RTOS kernel to access contents of the task's MPU_Vault.

Most RTOSs implement specific entry code at the starting point of ISR and exit code at the end of ISR that are responsible for saving and restoring the task context respectively. To protect MPU_Vault, we modify the ISR entry and exit code with our *trampoline* code to redirect the control flow to *secure monitor*. When an interrupt is received, the *entry trampoline* transfers the control flow to *secure monitor* prior to execution of the ISR. The *access control* module then searches for the task in `tskContextList` using its *tskID*. If the task is found

and `MPUVaultEnabled` flag is set, the *access control* changes the access permission of the corresponding MPU_Vault to no access (not readable and writable) by re-configuring *MPU_NS*. Next, the *access control* clears `MPUVaultEnabled` flag and returns the control flow back to the ISR to continue execution of the ISR routine.

When the execution of ISR is complete, access permission of the MPU_Vault has to be set back to read/write before the control is returned to the interrupted task. In order to do so, the *return trampoline* code that is inserted in the ISR exit code redirects the control flow to *secure monitor* before the task execution is resumed. The *access control* module again looks up the task in the `tskContextList`. If the task is found and `MPUVaultEnabled` is not set, *access control* changes the access permission of MPU_Vault back to read/write and sets `MPUVaultEnabled`. Algorithm 1 depicts the steps involved in protecting the MPU_Vault when the task is interrupted.

6.2.2 Control-Flow Integrity

When a task is interrupted, the CPU saves the current execution context state (such as Stack Pointer (`SP`)–`R13`, `LR` – `R14`, Program Counter (`PC`)–`R15`) onto stack of the task or `TCB` to ensure that the task resumes correctly after execution of the interrupt handler routine is complete. The value of `PC` saved on stack represents the return address from which task execution will resume after the interrupt. If the interrupted task had been accessing an MPU_Vault, an attacker with privileged access may manipulate the `PC` value on stack and cause an effective return from the ISR to a malicious code that could compromise the contents of MPU_Vault. A common technique to mitigate this kind of attack is by use of a protected shadow stack [42]. The shadow stack stores a copy of the return address which is compared with the return address on stack before the control flow is transferred back to the interrupted task. ShieLD protects register values that can be exploited by attackers using a secure protected stack similar to the concept of shadow stack. This protected stack is maintained in the Secure world. On task interruption, control flow is redirected to *secure monitor* by the entry trampoline code and *access control* module pushes the values in `PC`, `LR`, and `SP` onto the protected stack of the interrupted task. It also replaces those values on stack with arbitrary values to hide the task context. Upon returning from ISR and before resuming the interrupted task, when control flow is redirected to *secure monitor* by return trampoline code, *access control* module pops the stored values from secure protected stack and writes them back onto stack.

6.2.3 MPU_Vault Release

When the task accessing an MPU_Vault is terminated, we assume all of resources including MPU_Vault will be deleted by *RTOS* kernel. If the task being terminated has an associated MPU_Vault, ShieLD flushes contents of MPU_Vault and removes the task context information from `tskContextList`. For this purpose, the task termination routine in the RTOS kernel is modified with trampoline code which switches control to *secure monitor* prior to deleting task's resources. The *access control* module looks for the task in `tskContextList`. If the task is found, *access control* module flushes the message in MPU_Vault,

Algorithm 1: SHIELD'S MPU_VAULT LIFECYCLE

Input: ShieLD_request ▷ requesting message to invoke ShieLD services
Output: *success_code*

success_code = −1;
Save `R0, R1, R2, R3, R12, LR, sp, PC, return_address;` ▷ Saves execution context of the task
Switch to ShieLD using `SG` entry;
if *MPU_NS exists* **then**
  if *ShieLD_request == setMPUVault* **then**
    if *`buffer` is 32-byte aligned* **then**
      Set bit 0 of `MPU_CTRL` to 0; ▷ Disables MPU
      Set bits [7:0] of `MPU_RNR` to `MPUVault_RegionNumber`; ▷ Begins configuration of MPU_Vault
      Set bits [2:1] of `MPU_RBAR` to `0b01`; ▷ Sets the permissions of MPU_Vault to RW
      Set bit 0 of `MPU_CTRL` to 1; ▷ Enables MPU
      Update `tskContextList` with `buffer` address;
      Set `MPUVaultEnabled` flag;
      success_code ← 0;
    else
      success_code ← −1;
    end
  else if *ShieLD_request == protectMPUVault* **then**
    if *Is `MPUVaultEnabled` flag set* **then**
      Set bit 0 of `MPU_CTRL` to 0;
      Set bits [7:0] of `MPU_RNR` to `MPUVault_RegionNumber`;
      Set bits [2:1] of `MPU_RBAR` to `0b10`; ▷ Sets the permissions of MPU_Vault to NA by NS context
      Set bit 0 of `MPU_CTRL` to 1;
      success_code ← 0;
    else
      success_code ← −1;
    end
  else if *ShieLD_request == unprotectMPUVault* **then**
    Set bit 0 of `MPU_CTRL` to 0;
    Set bits [7:0] of `MPU_RNR` to `MPUVault_RegionNumber`;
    Set bits [2:1] of `MPU_RBAR` to `0b10`; ▷ Sets the permissions of MPU_Vault to RW
    Set bit 0 of `MPU_CTRL` to 1;
    success_code = 0;
  else if *ShieLD_request == deleteMPUVault* **then**
    Set bit 0 of `MPU_CTRL` to 0;
    Set bits [7:0] of `MPU_RNR` to `0`;
    Set bits [2:1] of `MPU_RBAR` to `0`;
    Set bit 0 of `MPU_CTRL` to 1;
    Delete `buffer` address from `tskContextList`;
    Clear `MPUVaultEnabled` flag;
    success_code ← 0;
  else
    success_code ← −1;
  end
else
  success_code ← −1;
end
Return to task by executing `BXNS LR`;
Restore `R0, R1, R2, R3, R12, LR, sp, PC, return_address;` ▷ Restores execution context of interrupted task
return *success_code*;

disables MPU protections of the region associated with MPU_Vault, and removes the task context information from `tskContextList`. The operations in MPU_Vault lifecycle from allocation to release are shown in Algorithm 1.

6.3 Integrity of ShieLD

In this section of ShieLD design, we discuss the protection of ShieLD components and how their integrity is maintained. Components of ShieLD (*MPU_NS* and *trampolines code*) residing in the Normal world may be compromised, as the software components that run in the Normal world are untrusted and vulnerable to various attacks. Therefore, it is important to protect the integrity of these components from unauthorized modification.

6.3.1 Trampolines Integrity

Our trampoline code residing in the interrupt handler and task termination routines are part of the RTOS kernel code and vulnerable to attackers. An attacker might tamper with our trampoline code to block MPU_Vault protection mechanisms entirely. Therefore, ShieLD protects the integrity of kernel code which is part of the static region of RTOS kernel. In TrustZone-enabled systems, when the device boots up, we assume that Secure world is booted first, which later transfers control to the Normal world. Before passing the control to Normal world, ShieLD verifies the integrity of kernel and enables MPU protections for its static region. The kernel code with read-only access permissions ensures that ShieLD components in the RTOS kernel are not compromised.

6.3.2 MPU Protection

As mentioned above in Section 3.3, *MPU* is programmable by privileged software, typically by the RTOS kernel to define access permissions and attributes for memory regions. This allows modification of the *MPU_NS* configuration by an attacker in the Normal world with privilege access. So the attacker may modify the *MPU_NS* configuration which is used to protect the MPU_Vault and manipulate the messages transferred via the MPU_Vault. To avoid this, ShieLD is able to setup the *MPU_NS* to be only writable/programmable from within the Secure world. In ARMv8-M architecture, access to peripherals (such as *MPU*) is implemented in the form of read/write access to memory address space using Memory–Mapped I/O (*MMIO*) method, i.e., the registers of MPU can be mapped to a memory region. Therefore, access to *MPU* could be protected by the *MPU* itself in the same way as any other memory access. ShieLD maps the *MMIO* address space of *MPU_NS* into an MPU region with read-only access permission, thereby preventing modification of the *MPU_NS* by the RTOS or other privileged software running in the Normal world.

7 ShieLD vs SeCReT

As mentioned earlier, mobile and conventional IoT devices supporting TrustZone-A are also prone to a vulnerable cross-world communication channel since TrustZone does not guarantee message integrity and source authentication. SeCReT is a sophisticated solution to protect the communication between the Rich Execution Environment (REE) and TEE. It proposes message encryption for communication between the domains; REE and TEE. SeCReT's design is based on ARMv7-A which specifies the 32 bit ARM architecture. SeCReT proposes using a session key when the Client Application (CA) and Trusted Application (TA) communicate. Protecting the session key requires involvement of SeCReT at different levels in the system like device boot, mode switch, interrupts, page table update and the selected crypto libraries.

SeCReT running in the monitor mode with the highest privilege in the system is responsible for creating the session key, maintaining the list of pre-authorized CAs allowed to invoke the TAs to use the key and storing all information relevant to session key management. SeCReT introduces additional steps in the device's secure boot sequence to calculate the hashes of the authorized CA static region and stores them in the TEE. SeCReT also inserts trampoline code (i) at the beginning of user mode exception handlers to invoke SeCReT between the user and the kernel mode switches to protect the integrity of the session key and (ii) into kernel code that handle process creation and termination. The trampoline invokes the SMC instruction with arguments that present information necessary to maintain the session key, such as the process descriptor address and kernel stack address. The trampolines are inserted in the kernel static region protected by the kernel integrity monitor. The key value always resides in a designated memory page in the memory. The permission of the memory page for provisioning the key is configured as no-access. SeCReT saves the translation table base register (TTBR) in a structure, as an identifier to lookup the keys against the CA, the hashes of the CA static region, and the value of the session key. SeCReT configures the access permission of the memory page using the Domain Access Control Register (DACR). The key only becomes accessible when the integrity of CA is verified. The integrity monitor also restricts memory page table updates in the REE which protects the key. The usage of the session key is also defined by the libraries used and simple operations could create copies of the key resulting in key compromise. SeCReT proposes instrumentation of the crypto library used for encryption to prevent the keys from being copied out of the protected memory. This prevents the stack from being used to cache a part of the key during use.

SeCReT was designed on ARMv7-A that supports the A profile instruction set which is different from the ARMv8-M profile on which we build the design of ShieLD. Most of SeCReT's design choices are targeted at the protection of the session key in a multi-threaded environment where processes run with multiple contexts. Moreover, in order to protect the session key, registers available for the Virtual Memory System Architecture (VMSA) like the DACR are used. The ARMv8-M memory management is relatively simpler as there is no virtual memory, address translation, memory page management which are all part of SeCReT's design considerations. Implementing SeCReT on ARMv8-M will require engineering effort in the following aspects: (i) exploring key management and secure key storage mechanisms for devices based on ARMv8-M, (ii) utilizing the MPU and SAU rather than DACR to manage memory accesses, (iii) designing/instrumenting crypto libraries to prevent duplication of keys to give the same level of guarantees

provided by SeCReT, (iv) identifying relevant locations for adding trampoline code in the interrupt vector tables, and (v) optimizing the design to compensate the massive overhead induced by key provisioning and message encryption on M-profile architecture.

8 Implementation

In this section, we discuss the implementation of a prototype of ShieLD framework on a real hardware board in order to do a thorough experimental evaluation of the framework. ShieLD design was based on ARMv8-M, which is a 32-bit ARM architecture for Cortex-M processors.

8.1 Runtime Environment

The implementation of ShieLD prototype builds on TrustedFirmware-M (TF-M) [43] in the secure side with CMSIS RTOS2 as Normal world OS. Approximately 156 Lines of Code (LOC) were written to modify the Normal world OS and around 203 LOC were added to the TF-M code. TF-M provides a reference implementation of Secure world software for ARMv8-M [44]. It creates the foundations of TEE by providing a set of secure run-time services such as secure storage, cryptography, attestation etc. Additionally, secure boot in TF-M ensures integrity of run-time software and supports firmware upgrade.

8.2 ShieLD Components

As shown in Figure 5, ShieLD has several components, both in the Normal and Secure worlds. In this section, we describe the implementation of these components. We implemented the *secure monitor* as a runtime secure service on top of TF-M in the Secure world.

8.2.1 Interface

In our prototype, we implemented ShieLD services as secure functions. ShieLD provides *ShieLD _veneers* (see Figure 5) as secure function APIs which can be called by the NS tasks to use ShieLD services. The veneer gateway functions are marked with the nonsecure entry attribute (`cmse_nonsecure_entry`) as shown in the code snippet in Listing 2. The *interface* module is implemented to handle all calls via *ShieLD _veneers*.

```
__attribute__((cmse_nonsecure_entry))
    void ShieLD_veneers(){
    // ShieLD_service()
}
```

Listing 2: Veneer Gateway Function declaration

8.2.2 App Verifier

To implement `tskContext`, we extended *Known Client List* data structure in the secure side which is used to register active NS task. We used SHA-512 provided by *TF-M crypto services* to calculate cryptographic hash of the task code.

8.2.3 Access Control

TF-M provides a partially implemented API for the MPU. We extend this *mpu_armv8m_drv API* for ShieLD's *Access Control* module. The *mpu_armv8m_drv API* accesses the Non-Secure MPU using MPU_BASE_NS which is the alias address used to configure MPU_Vault in the NS side.

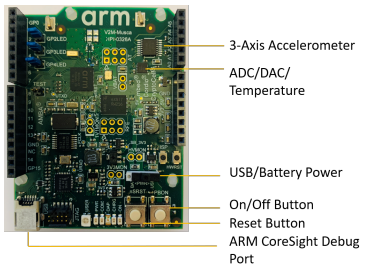


Fig. 8: TrustZone-M enabled IoT device, ARM Musca-A2 Test Chip Board, we use for implementation and evaluation.

9 Performance Evaluation

In this section, we empirically evaluate the performance of ShieLD prototype. We perform a set of microbenchmarks on our *Smart Door Lock App* replicating the usecase application scenario in Section 5. We choose *memory* occupancy of the ShieLD code, *latency* and *CPU time* as our performance metrics. Memory and power consumption are the most constrained resources in low-power IoT devices. The CPU time is directly proportional to the power consumed by the ShieLD operations. We calculate the overhead induced on the application by enabling ShieLD services and discuss the application-independent and application-dependant results.

9.1 Experimental Setup

We evaluated the performance of ShieLD on the Musca-A2 Test Chip Board by Arm (Figure 8). The Musca-A2 board implements the ARM CoreLink SSE-200 subsystem featuring dual-core Cortex-M33 with CPU0 enabled at 50MHz [45]. We use TF-M and CMSIS RTOS v2 enabled with ShieLD components for these experiments. The execution time is measured using the CoreSight debug port found on Musca-A2 Test Chip Board (see Figure 8). The CoreSight debug port contains a 32-bit free running counter that counts CPU clock cycles. The counter is part of the *Debug Watch and Trace (DWT)* module which we use to measure the execution time of our code. We repeat each calculation 10 times and our standard deviation is 2μs.

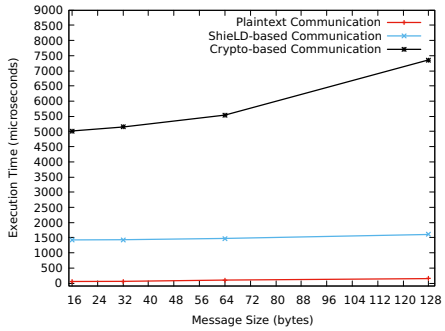
9.2 Communication overhead of ShieLD and Crypto-based Communication

Here we compare the total time (round trip time) for a single cross-zone interaction of the *(i)* ShieLD-enabled communication including the setup and release phases, *(ii)* crypto-based communication, and *(iii)* the current unsecure (plaintext) communication. This evaluation is primarily done to benchmark the overheard of ShieLD on resource-constrained IoT devices.

Prior to our work, no solution exists that provide cross-zone secure communication in resource-constrained IoT. A cross-zone secure communication solution exists for powerful conventional TrustZone device, i.e., SeCReT [4] which is discussed in Section 7. Due to platform-level differences, converting the entire mechanism of SeCReT to ARMv8-M based IoT devices supporting TrustZone-M is infeasible. ARMv8-M architecture has relatively limited resources as compared to ARMv7-A, hence the entire SeCReT solution

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3147262, IEEE Transactions on Dependable and Secure Computing

Fig. 9: Execution time (in microseconds) of ShieLD-enabled communication compared to crypto-based and plaintext cross-world communication



with heavy use of crypto, is not implementable in resource-constrained TrustZone-M devices. Therefore, we only provide a re-implementation of cryptographic components of SeCReT protocol for the sake of comparison of encrypting and decrypting messages on Cortex-M platform. We take the essential components of crypto-based communication mechanism like SHA-512 to ensure integrity and AES for encryption, and compare them with confidentiality and integrity protection mechanisms of ShieLD.

The total time of the ShieLD-enabled communication includes the execution time of verifying integrity of the task, enabling MPU_Vault, message transfer and release of MPU_Vault on task termination. Unlike apps in traditional systems, the task termination is a rare event since the IoT devices are usually continuously active and tasks terminate only when a device is updated or rebooted.

For crypto-based communication, we only enable the confidentiality (encryption) and integrity (MAC) services. Figure 9 shows the comparison between these three communication modes for different message sizes. In the case of crypto-based communication, we see a visible escalation in total execution time with the increase in message size, this is due to the fact that time consumed by cryptographic operations is directly proportional to the size of the message being encrypted. In contrast, the ShieLD-protected communication has insignificant overhead with the increase of message size. It is an important feature of MPU-based protections that the execution time of setting up and protecting a region remains independent of the size of the message/region. The comparison between these two communication modes is also shown in Table 2, where overhead of individual message sizes are further highlighted. Note that the overhead of the crypto-based communication will be further escalated with authentication and key handling operations. This comparison aims to clarify that if the same mechanism used for Cortex-A based devices is replicated for IoT devices based on Cortex-M processors, the outcome would be same security guarantees as provided by ShieLD but much greater overhead due to limited system resources and key management overhead which will result in slowing down the entire system operation. In real-time OSes, slowing down system responses can have drastic effects on the safety and security of the device and the infrastructure.

TABLE 2: Execution overhead of ShieLD as compared to crypto-based protection for varying message sizes

Message Size	**Plaintext**	**ShieLD**		**Cryptography**	
Bytes	Time (μs)	Time (μs)	Overhead	Time (μs)	Overhead
16	55	1,427	**27x**	5,016	**91x**
32	59	1,431	**25x**	5,147	**87x**
64	101	1,473	**15x**	5,539	**54x**
128	150	1,605	**10x**	7,358	**49x**

9.3 ShieLD Overhead on Smart Door Lock Application

We measured the total overhead of using ShieLD protection mechanism in terms of *CPU Time* on a *Smart Door Lock App* execution. We considered the operations and control flow of the application as described in section 5. In a system with ShieLD services enabled, the application lifecycle includes additional steps of execution and the application lifecycle is slightly different: (i) the *Smart Door Lock App* requests ShieLD to setup an MPU_Vault in the Normal world, (ii) ShieLD sets up an MPU_Vault to be used by the *Smart Door Lock App*, (iii) when the *Fingerprint Sensor* receives input, the *Smart Door Lock App* requests the *Verify Fingerprint* service to verify the fingerprint input, based on which a response is returned to the *Smart Door Lock App* using the allocated MPU_Vault, (iv) the release of the MPU_Vault takes place during task termination.

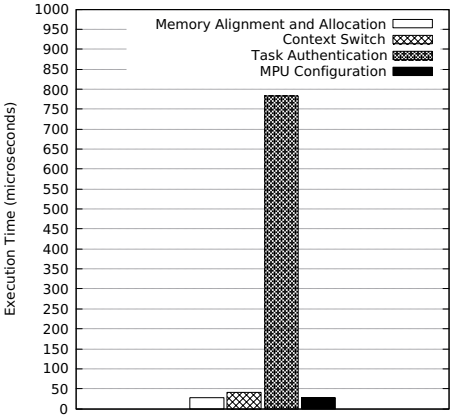
CPU Time Overhead of ShieLD: The major overhead of the proposed solution are application-independent as the overhead of the individual operations (context switch, task authentication, MPU configuration and memory alignment and allocation) remain the same irrespective of the application behaviour and execution. The evaluation results of the *Smart Door Lock App* are based on the interactions the application has with the secure services. The overhead on the system due to MPU_Vault setup and release is *946μs* which is a one-time overhead and remains independent of the application behaviour. There remains a minimal overhead of *56.4μs* on the system due to using a ShieLD protected channel for communication. An application with a more frequent communication pattern with the secure services, will have additional context switches and the overhead will vary. A detailed breakdown of the individual components of ShieLD is given in Section 9.4 and Section 9.5.

Memory Overhead of ShieLD: As resource-constrained IoT devices have very limited storage, our ShieLD code is highly optimized and increases the firmware binary (TF-M) size only by 100kb, which is 0.04% of the total TF-M binary. The memory overhead of ShieLD components is independent of the applications requesting ShieLD services.

9.4 CPU Time to Setup MPU_Vault

We measured the total *CPU Time* to setup the MPU_Vault, which involves the following sequence of operations: *(i)* A task in the Normal world allocates a 32-byte aligned memory region to be configured as MPU_Vault, and makes a call to the *secure monitor* in Secure world (MPU_Vault Allocation Request); *(ii)* the task context is saved and the control flow is transferred to *secure monitor* and the secure context is restored; *(iii)* the *app verifier* module calculates the hash of the task's code binary using *SHA-512*, gets the task

Fig. 10: Execution time (in microseconds) of operations to setup the MPU_Vault before ShieLD-based communication can begin: (i) Memory alignment and allocation for the MPU_Vault, (ii) Context switch between the secure and non-secure world, (iii) Task integrity verification and source authentication using HL and (iv) MPU configuration to set appropriate permissions



information from the tskLst and compares it against hash values in HL (*task authentication)*; *(iv)* the *access control* module configures an MPU region for MPU_Vault; and *(v)* the Secure world context is saved on secure stack, the execution context of the NS task is restored, and the control flow is transferred back to the task. The time to perform these operations individually is shown in Figure 10 in which *context switch* corresponds to the cumulative execution time of step (ii) and (v). The total time required to setup MPU_Vault is 878μs. The *task authentication* is the most time-consuming operation and takes 783μs to authenticate a task with a 64-byte code region. However, this authentication takes places once for a single task in the entire MPU_Vault's life-cycle. It is worth mentioning here that increasing the size of the MPU_Vault has no affect on performance of the system. Configuring MPU protections for a region takes *28μs* irrespective of the size of the region. The only operations incurring a significant impact on the system remain the hash calculation and verification to verify the integrity of the task, which further shows that cryptographic solutions to provide secure communication by encrypting messages in such relatively resource-constrained platforms would be impractical.

9.5 CPU Time to Protect MPU_Vault

Here we measured the *CPU Time* to protect the MPU_Vault in case of an interrupt. As mentioned in Section 6.2, every time an interrupt occurs, the ShieLD performs a set of operations to protect the MPU_Vault: *(i)* redirect control flow to the Secure world using trampoline code before execution of the interrupt handler; *(ii)* change the access permission of MPU_Vault by re-configuring the MPU_NS; *(iii)* save registers into the secure protected stack for control flow integrity; and *(iv)* return the control flow back to the interrupt handler routine. These operations are also independent of the currently executing application. The average *CPU Time* for the ShieLD protection mechanism during interrupt handling is *185μs*. This also means every ShieLD-protected interrupt handling operations will add a delay of *185μs*. This delay adds only 0.11% overhead in the total *CPU Time* required for ShieLD-based communication which has very little effect when compared with crypto-based solution (see Table 2). Also, note that this delay is only applicable if an interrupt occurs during a cross-zone communication.

10 Scalability of ShieLD in Resource-Constrained IoT

In this section, we discuss the scalability of ShieLD services with increase in number of applications from three perspectives: availability of (i) RAM, (ii) MPU regions and (iii) size of MPU region. IoT devices based on Cortex-M [3] are designed to run fewer applications as compared to high-end devices (based on Cortex-A). With kilobytes of RAM, these devices run 1-3 applications on average. If system memory supports the increase in applications, ShieLD services will be valid. Since MPU-based protections are not bound to a memory range i.e. an MPU can protect a memory region ranging from 32 bytes to 4 GB in size [26], increase in MPU_Vault size would not entail any performance overhead on ShieLD. As described with experimental evidence in section 9.4, performance overhead to enable an MPU-protected region does not increase with the increase in region size. ShieLD services are limited only by the availability of MPU regions. Based on the processor architecture, we have 8-16 memory regions that can be protected using the MPU. Considering that ShieLD protects the MPU_Vault, the requesting task's code, stack and the RTOS static region using MPU, ShieLD can provide secure communication services to 3 applications simultaneously if the system has 16 available MPU-protected memory regions.

11 Security Analysis

We assume that an attacker with privileged access can compromise any component running in Normal world including ShieLD components to launch various attacks on the MPU_Vault which resides in the Normal world. The legitimate application (i.e., the owner of the MPU_Vault) is not malicious itself, but can be compromised. We also assume that an unprivileged task can interrupt the ShieLD operations and try to access the MPU_Vault contents. Our assumptions regarding the Secure World being intact and not accessible to any attacker remain applicable. We conduct a security analysis of the system under the above attacker assumptions using an attack tree (Figure 11). The attack tree is used to itemize the attack vectors and provide a higher level of abstraction for the security analysis of ShieLD. The root of the tree represents the final goal of an attacker, which is to compromise the MPU_Vault by gaining access and modifying its contents (**G**). The child nodes represent the possible attacks or sub-goals. In order for a parent attack to be successful, at least one of the child attacks need to be successful. Figure 11 also enlists a few (out of many) instances of vulnerabilities (**V**) that can be exploited to gain a bare minimum level of control within the Normal world

3. https://www.mbed.com/built-with-mbed/smart-door-locks

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3147262, IEEE Transactions on Dependable and Secure Computing

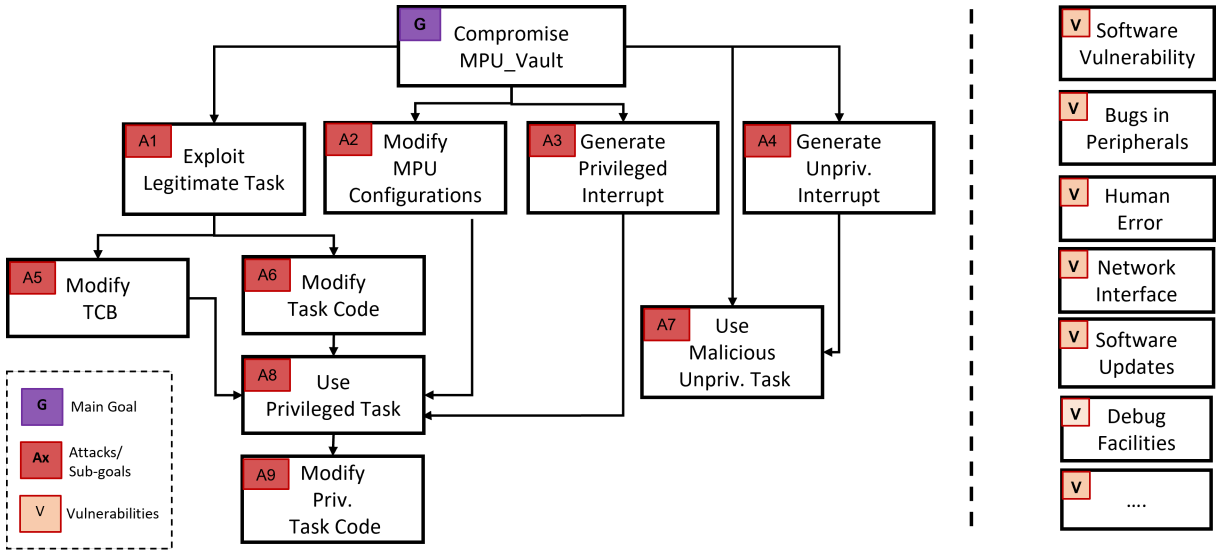


Fig. 11: ShieLD's threat analysis using an attack tree to itemize the attack vectors. The root of the tree represents the main goal of the attacker, which is to access and modify/overwrite the contents of MPU_Vault. The child nodes represent attacks or sub-goals that are needed to achieve the main goal. In this attack tree, if one of the child attack is successful, the parent attack is successful (OR Condition). Each of the attacks is possible by exploiting some vulnerability in the Normal world and a brief list of vulnerabilities is represented on the right side of the figure.

e.g., by including malicious software in the software stack of the device. The list of vulnerabilities represented here is not absolute, but is a subset of the all the vulnerabilities that the Normal world software might have. Some vulnerabilities are specific to attacks, like using the debug facilities to interrupt execution, while others are generic and equally applicable to most of the attacks.

Now, we discuss the attacks one by one (A1 to A9 in Figure 11) and how ShieLD mechanisms are designed to defend against each of them. We argue for the sufficiency of the defense mechanisms against those attacks and how the attacks fail in order to obtain the expected security guarantees of ShieLD. The attacks **A1, A2, A3, A5** and **A6** are possible if an attacker has privileged access in the Normal world (e.g., by compromising the Normal world OS). The privileged attacker may then *exploit a legitimate task code* (**A1**) by injecting carefully crafted code sequence (**A6**) or modify the `TCB` (**A5**) and replace addresses of the legitimate task to alter the control flow of execution and access the contents of the MPU_Vault. ShieLD avoids **A6** by setting the code memory region of the task to read-only using MPU region after verifying the integrity of the task code when a new request for MPU_Vault allocation is received. The attack **A5** can be prevented by configuring the RTOS in a way that it stores the `TCB` in the Secure world [46]. The privileged attacker may also attempt to modify the configuration of the Normal world MPU to *extract the contents of MPU_Vault* (**A2**). ShieLD defends against it by ensuring that the Normal world MPU must be only programmable within the Secure world. This configuration also protects the MPU_Vault from an attacker running in a different CPU context. If an untrusted privileged application runs in CPU0, and the application that communicates with the TEE runs in CPU1, the MPU_Vault would not be accessible to the untrusted application. The TrustZone logic is implemented in the hardware design including the interface between an M33 core and the system bus. Hence, each memory access request (irrespective of the CPU core) propagates through the IDAU, SAU and the MPU and the privileged application's access to the MPU_Vault is efficiently blocked. However, a privileged software with "read-write" permission to MPU_Vault (as shown in Table 1) might be able to corrupt the contents of MPU_Vault while it is being updated (**A3**). Although the attacker cannot reliably control the MPU_Vault due to a race condition, interrupting the execution of MPU_Vault update in an attempt to write/overwrite the contents might be possible. To this end, we have proposed MPU_Vault protection mechanisms in Section 6.2. Whenever a task is interrupted during MPU_Vault update (in this case, by untrusted privileged application running in CPU0), the entry trampoline transfers the control to the *secure monitor* of ShieLD, which changes the access permissions of the MPU_Vault to "no access" (neither readable nor writable), hence preventing the privileged application from making any changes to the MPU_Vault contents. An attacker could *intervene* the MPU_Vault allocation process by interrupting the execution (**A3, A4**) and modify the buffer address that is supposed to be transferred through the general-purpose registers. As explained in Section 3.2, TrustZone for ARMv8-M supports a direct function call between the non-secure and secure software without an intervention of any software in the Normal world. Therefore, tasks running in the Normal world can send an MPU_Vault allocation request to the *secure monitor* without the intervention of the attacker. This allocation request can be configured to be uninterruptible. Moreover, ShieLD also verifies if the return address is in the memory range of the task code region when the control flow is returned to the task after the MPU_Vault allocation is completed. The *debug facilities* can also be used to interpose the execution (**A4**) and intervene the allocation

of MPU_Vault or when the task is interrupted. To this end, TrustZone implements debug-authentication-signals, which can disable debug facilities when processor runs in secure state [2] hence preventing this attack. When a task accessing an MPU_Vault is interrupted (**A3**), the privileged attacker could attempt to *tamper with our trampoline code* in the RTOS kernel to bypass the MPU_Vault protection (**A8**). ShieLD prevents this attack by setting the access permission of the code region of the RTOS kernel where our trampolines are inserted to "read-only" using MPU. This configuration also minimizes the attack surface that an unprivileged malicious task could misuse to escalate privileges by exploiting a vulnerability in the RTOS code (**A9**). Moreover, protecting the task's code and stack as described in 6.1.3 makes the attackers attempt at a *Return-oriented Programming* and *Jump-oriented Programming Attack* futile. Finally, a privileged and unprivileged attacker could perform *memory dump attack* by taking a snapshot of the MPU_Vault physical memory area to access the data it contains (**A7**). Since the MPU_Vault memory region is always protected by the MPU with "no access" configuration (as shown in Table 1), the attacker cannot dump the memory region of MPU_Vault and read its content.

11.1 Security Analysis of ShieLD vs SeCReT [4]:

This section presents a comparative discussion on ShieLD's and SeCReT's [4] respective approaches to ensure (i) confidentiality of the transferred message (ii) integrity of the system components and (iii) availability of the secure communication service. ShieLD protects the message confidentiality by using the MPU. Since MPU-protected regions are neither accessible nor modifiable, the message confidentiality is preserved. On the other hand, SeCReT uses cryptography for message confidentiality. ShieLD protects the integrity of its system components like the MPU, trampoline code placed in the Normal world, privileged Normal world software and the `TCB` using mechanisms discussed in Section 6.3 to ensure the overall integrity of ShieLD. SeCReT uses TrustZone-A's Active Monitoring to protect the Active Process Context and page tables to prevent an attacker from tampering with system components (like the ISR trampolines, TCB). A Denial-of-Service Attack targeted at disrupting the services provided by TrustZone is viable and hard to defend. Such an attack on availability of the secure communication service is equally likely on ShieLD as well as SeCReT. In the case of SeCReT, an attacker can block requests to TrustZone altogether, hence preventing any application from calling secure services. With ShieLD, a malicious application could transfer excessive data repeatedly, thus depleting the available number of MPU regions to cause a Denial-of-Service. A behavioural analysis on the applications could help identify malicious application patterns and deny repeated requests for resource access. This is an interesting problem and is the focus of our upcoming work. Table 3 gives a list of attack vectors specified by SeCReT [4] and stating the existence of defense mechanisms (if addressed) by ShieLD and SeCReT.

TABLE 3: The table shows the attack vectors addressed by SeCReT [4] and ShieLD (details discussed in Section 11)

Attack Vector	Defense	
	SeCReT [4]	**ShieLD**
Shared data modification	Yes	Yes
Task code modification	Yes	Yes
ISR trampoline code modification	Yes	Yes
`TCB` modification to redirect control flow/modification of page tables	Yes	Yes
Memory dump attack on data	Yes	Yes
Using debug facilities/breakpoints to interpose secure execution	Yes	Yes
ROP Attack	No	Yes
JOP Attack	No	Yes

12 Limitations

In this section, we discuss some of the limitations of ShieLD which also lay the ground for future work and improvements. The security guarantees of ShieLD rely on the correct implementation of TrustZone-M specifications. There exist precise guidelines to implement TrustZone-M capabilities as incorrect implementations could collapse the TEE. As long as TrustZone-M remains resilient against all classes of physical and side-channel attacks, ShieLD guarantees also remain intact. Secure application design and implementation plays a crucial role in the overall system security. Applications that are to be placed in the TEEs should be free of bugs that could be exploited. ShieLD relies on the application developers in this regard. This is another limitation of ShieLD that it does not incorporate code analysis capabilities. As a result, the system remains prone to vulnerabilities like input validation and system bugs leading to buffer overflows. Such flaws in the secure code can be exploited to inject maliciously crafted parameters to install rootkits which are extremely hard to detect. Since the secure code placed in the TEE has access to the entire secure software stack, having third-party applications in the TEE could lead to eavesdropping by curious applications, this presents itself as an interesting research problem and is part of our future work. The scalability of ShieLD with respect to the number of applications ShieLD can protect on a system depends on the resources available. As mentioned in Section 10, the availability of a limited number of MPU regions (8-16 depending on the platform), impacts the number of applications it can protect. Since ShieLD protections depend on the availability of MPU regions, it presents itself as a limitation.

13 Conclusion

We have presented ShieLD, a framework that enables secure communication, in the presence of a vulnerable software stack (including the normal-world OS), between the two worlds of TrustZone-M within an IoT device. ShieLD exploits the novel use of MPU and enables a secure vault that is exclusively accessible to the legitimate application in the Normal world that wants to access and execute security-critical operations in the Secure world. ShieLD provides similar security services (authentication, confidentiality, and integrity) as provided by the conventional crypto-based secure communication. We have implemented ShieLD in a TrustZone-M enabled IoT device and evaluated its memory and execution time (that translates to power/energy) overhead. Our empirical evaluation shows that ShieLD is extremely efficient when compared with the crypto-based

communication protection. Though ShieLD targets IoT devices featuring TrustZone-M, the techniques proposed in this paper could be extended to other TEEs. We plan to extend this work to IoT devices that use the RISC-V architecture, exploiting the Physical Memory Protection (PMP) [47] unit of RISC-V.

ACKNOWLEDGMENTS

This work was partly supported by the Swedish Foundation for Strategic Research (SSF) aSSIsT and RISE KP, and partly by the H2020 VEDLIoT (GA No. 957197) and H2020 CONCORDIA (GA No. 830927) projects.

REFERENCES

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*.

[2] ARM Ltd., "TrustZone technology for the ARMv8-M architecture," https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf, 2019.

[3] Prove & Run, "ProvenCore-M," https://www.provenrun.com/products/provencore-m/, 2017.

[4] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment." in *NDSS*, 2015.

[5] Discretix, "Hardware-Assisted DRM with ARM TrustZone on the Samsung GALAXY S III Smartphone Secured by Discretix," https://www.discretix.com/press-release/hardware_assisted_drm_with_arm_trustzone_on_the_samsung_galaxy_s_iii_smartphone_secured_by_discretix/, 2012.

[6] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments." in *NDSS*, 2017.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5.

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

[9] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a {TPM} chip," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*.

[10] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013.

[11] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." *HASP@ ISCA*, vol. 11, 2013.

[12] ARM, "ARM security technology, building a secure system using TrustZone technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html, 2019.

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

[14] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

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

[16] T. Kobayashi, T. Sasaki, A. Jada, D. E. Asoni, and A. Perrig, "Safes: Sand-boxed architecture for frequent environment self-measurement," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*.

[17] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Assured: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11.

[18] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *Proceedings of the 17th ACM conference on Computer and communications security*.

[19] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voila!"

[20] Sequitur Labs, "CoreLockr-TZ," https://www.sequiturlabs.com/corelockrtz/, 2017.

[21] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "Ltzvisor: Trustzone is the key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[22] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "Tz-virtio: enabling standardized inter-partition communication in a trustzone-assisted hypervisor," in *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*.

[23] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5.

[24] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGPLAN Notices*, vol. 48, no. 4.

[25] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *ACM SIGPLAN Notices*, vol. 49, no. 4.

[26] ARM Ltd., "ARMv8-M Memory Protection Unit," https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf, 2017.

[27] ARM Mbed, "Mbed uVisor," https://www.mbed.com/en/technologies/security/uvisor/, 2015.

[28] TOCK, "TOCK," https://www.tockos.org/, 2015.

[29] F. Paci, D. Brunelli, and L. Benini, "Lightweight io virtualization on mpu enabled microcontrollers," *ACM SIGBED Review*, vol. 15, no. 1.

[30] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2mon: Building an mmio-based security reference monitor for unmanned vehicles," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[31] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "{ACES}: Automatic compartments for embedded systems," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*.

[32] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching." in *NDSS*, 2018.

[33] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "urai: Securing embedded systems with return address integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[34] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uxom: Efficient execute-only memory on {ARM} cortex-m," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 231–247.

[35] ARM Ltd., "ARMv8-M Architecture Reference Manual," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0553a.b/index.html, 2019.

[36] ARM Keil, "Using trustzone on arm®v8-m," http://www.keil.com/appnotes/files/apnt_291.pdf, 2019.

[37] ARM Ltd., "ARM® Cortex-M33 Processor Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.100230_0002_00_en/cortex_m33_trm_100230_0002_00_en.pdf, 2017.

[38] ——, "ARM®v8-M Fault Handling and Detection ," https://static.docs.arm.com/100691/0200/armv8m_fault_handling_and_detection_100691_0200_en.pdf, 2017.

[39] "The State of Enterprise IT Infrastructure & Security," Ping Identity Corporation, Survey, 2018. [Online]. Available: https://www.pingidentity.com/en/company/press-releases-folder/2019/security-concerns-preventing-cloud-saas-adoption.html

[40] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, vol. 7.

[41] ARM Ltd., "Platform security," https://developer.arm.com/architectures/architecture-security-features/platform-security, 2019.

[42] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*.

[43] ARM Ltd., "Trusted firmware-m documentation," https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/tfm/index.html, 2021.
[44] ——, "ARM® platform security architecture overview," https://www.arm.com/why-arm/architecture/platform-security-architecture, 2018.
[45] ——, "Arm® Musca-A Test Chip and Board Technical Reference Manual," https://static.docs.arm.com/101107/0000/arm_musca_a_test_chip_and_board_technical_reference_manual_101107_0000_00_en.pdf, 2018.
[46] ——, "RTOS Design Considerations for ARM®v8-M based Platforms," https://static.docs.arm.com/100689/0101/rtos_design_considerations_for_armv8_m_based_platforms_100689_0101_00_en.pdf, 2016.
[47] K. Cheang, C. Rasmussen, D. Lee, D. W. Kohlbrenner, K. Asanovic, and S. A. Seshia, "Verifying risc-v physical memory protection," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) Workshop on Secure RISC-V Architecture Design*, 2020.

Anum Khurshid is an Industrial PhD Student at RISE Research Institutes of Sweden. She is doing her PhD in Hardware-assisted Trusted Execution Environments focusing on IoT security. Her research interests broadly encompass software security, embedded systems security, IoT Certification, TrustZone-M. Anum did her Master of Science in Computer Science specializing in Cybersecurity and BS in Computer Science from COMSATS University, Islamabad in 2017 and 2014 respectively.

Sileshi Demesie Yalew is a PostDoc Researcher at RISE Research Institutes of Sweden. He did his Joint Doctorate in Distributed Computing at Instituto Superior Técnico, Universidade de Lisboa, Portugal and KTH, Sweden in 2018. His research focuses on mobile device (Android) security using the ARM TrustZone. He started his research career in 2009 when he obtained the MSc degree from the Universiti Putra Malaysia in Malaysia, addressing garbage collection issues in the LINDACAP system.

Mudassar Aslam is a Senior Researcher in the Cybersecurity Unit at RISE Research Institutes of Sweden. His research interests include IoT, Edge and Cloud security with expertise in platform security mechanisms provided by TEEs. Previously, he remained Assistant Professor in COMSATS University, Islamabad for over 9 years. He completed his PhD in 2014 from Mälardalen University Sweden co-hosted by RISE (formerly SICS). His research, during PhD, was recognized with the Best Paper Award in SIN'13 conference. Earlier, he did his Masters in Security from KTH, Sweden in 2009, and BS in Computer Science in 2002.

Shahid Raza is the Director of Cybersecurity Unit at RISE Research Institutes of Sweden, and an Associate Professor in Uppsala University. Shahid's research interests are focused on security and privacy in IoT. His work on IoT security is published in prestigious journals and conferences and, only in last two years, he has received over 1700 citations on his IoT security papers. Shahid received Industrial PhD degree in 2013 from Mälardalen University and a Master of Science degree in cybersecurity from KTH Sweden in 2009. Shahid is currently supervising 4 PhD students (including Anum Khurshid) as the main supervisor. [http://shahidraza.net]