Baremetal TEE with MPU (MCU-Fortifier): Design and implementation description

Emanuele Beozzo
emanuele.beozzo@unitn.it

Daniele Giuliani
daniele.giuliani@unitn.it

February 2025

Revision History

Revision	Date	Author(s)	Description
1.0	24/01/2022	Daniele Giuliani	First version
2.0	03/02/2025	Emanuele Beozzo	New documentation after TEE re-design

Contents

1 Scope of document

This document presents the design principles of the Baremetal TEE with MPU (also called MCU-Fortifier or Paladino), the resulting implementation and the reasoning behind the different choices made during development. This can be used to understand, modify, maintain, and update the system.

To understand how to design, develop and deploy both Client Applications (CAs) and the Trusted Applications (TAs) for the solution, have a look at the README file in the Baremetal TEE repository of the CROSSCON project (link: `https://github.com/crosscon/baremetal-tee/tree/main/MPU-version`)

2 Introduction

The BareMetal-TEE with MPU is a solution for memory isolation and supervised execution aiming at increasing the security of embedded devices in a way that is mostly transparent to application developers. The architecture is fully compliant with the Global Platform TEE Client API specification and with a subset of the TEE Core API. The BareMetal TEE with MPU supports the deployment of up to two custom Trusted Applications (TAs) compliant with the Global Platform TEE Core API and up to one untrusted Client Application (CA).

MCU-Fortifier is designed to work on Cortex M4 devices (ARMv7-M architecture) equipped with a Memory Protection Unit (MPU). In particular, the Baremetal TEE was developed using a B-L475E-IOT01A1 board (STM32L475VG MCU) and tested in that specific environment. Compatibility with other M4 boards is theoretically possible, but hardware-dependent configurations (e.g., memory map and MPU configurations) should be changed.

The software package can be decomposed in the following macro components which will be explained throughout this document:

- **Fortified Application**: an embedded application specifically compiled in order to be compatible with the Baremetal TEE. The fortified application can also be referred as *user application*, *client application* or *CA*.
- **BareMetal TEE with MPU (MCU-Fortifier Microvisor):** the first piece of software executed on boot of the embedded device, acts as an OS, bootstraps the execution of the Fortified Application and constantly monitors it. The TEE also offers core security functionalities like secure storage, key generation and cryptographic primites compliant with the Global Platform Core API. It is sometimes referred to simply as *microvisor* or *TEE* and it is the main component of the software solution.
- *Trusted Applications*: embedded security services that run in an isolated environment (separate memory and storage). It is an interface between unstrused applications and the TEE. The trusted applications are usually referred as *TA*.
- *Instrumenter*: tool that during compilation instruments source files at assembly level allowing the creation of fortified applications.

MCU-Fortifier operates at a very low-level, for this reason a good knowledge of the ARMv7-M Architecture Manual[1] is required as a pre-requisite to this document. Nevertheless, some of the more obscure aspects and implications of the architecture will be clarified to the reader when considered appropriate.

This document will begin by detailing the design principles of MCU-Fortifier in Section 3. In Section 4, we will examine the resulting implementation, decomposing it into separate modules and analyzing the different problems that arose during development. Finally, in Section **??**, we provide a clear structure of the Baremetal-TEE with MPU repository, highlighting the different source files and the components implemented by each one.

3 Design

In this section the design of Baremetal TEE with MPU will be analyzed while presenting a high-level overview of its main components. Some features of the ARMv7-M architecture will also be introduced in order to provide the reader with the minimal necessary context.

3.1 Execution Modes and Privileges

MCU-Fortifier is designed to protect the device by isolating the untrusted application from the trusted application and the Microvisor, and the trusted application from each other. This can be achieved by combining different features of the ARMv7-M architecture.

In particular, we employ the two execution modes and two levels of privileges provided by the architecture. These execution modes are:

- *Thread Mode*: this is the standard execution mode selected after a reset of the device. This mode can run with and without privileges.
- *Handler Mode*: this execution mode is enabled when executing exception handlers (both for system exceptions and interrupts). It always runs with privileges.

Sometimes we refer to code executing with privileges as running in *privileged mode* (or *unprivileged mode* when executing without privileges) although this nomenclature does not provide full information on which actual execution mode is selected.

The level of privileges influences what operations can be performed by the code being executed. Generally, privileges are needed to execute assembly system instructions (e.g. `MSR`, `MRS`, `CPS`) but are also required to access the Private Peripheral Bus (PPB) containing, among other stuff, the system configuration.

With the use of a Memory Protection Unit (MPU), privileges become much more powerful, allowing us to divide the addressable space into different regions and specify which region of memory can be read, written and executed by privileged and unprivileged code separately.

3.2 Memory Map and ACP

Using privileges and the MPU, MCU-Fortifier provides memory isolation by dividing the addressable memory into different sections and subsections as shown in Figure 1. This simplified memory map can be used to understand where the different components (Microvisor, Trusted Apps and Client App) are deployed and can be useful to understand how the TEE operates.

The TEE is stored in two flash memory segments: one accessible (R/W) and executable only by privileged code which contains the core of the Microvisor, while the other accessible and executable by every component containing unprotected code and shared libraries (e.g. C standard library). The Client

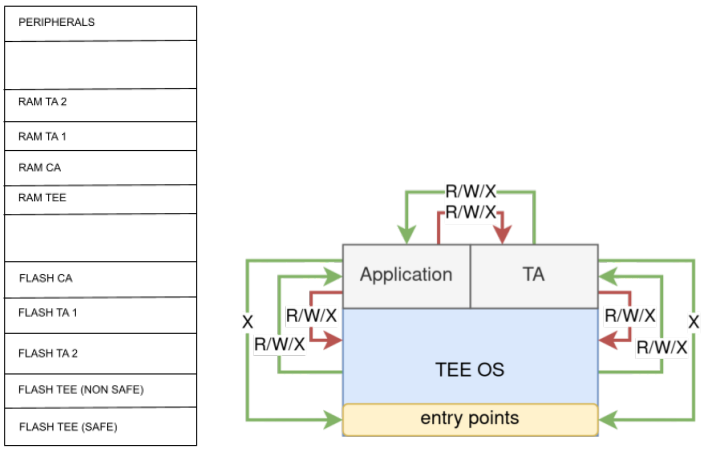

Figure 1: Simplified memory organization of device running MCU-Fortifier and the access control policy for each component

Application is stored in another section and can be read and executed by unprivileged code. Additionally, privileged code has full access (R/W/X) on it. Regarding the Trusted Application, each of them has a defined section that can be accessed only by itself and the privileged code. Other trusted applications and the untrusted application can not access it.

The CA RAM is accessible by both privileged and unprivileged code, while similar rules as for the flash memory apply to the RAM sections dedicated to the microvisor and the trusted applications.

3.3 Enforcing ACP

After illustrating how memory is organized, in order to enforce the Access Control Policy (ACP) we need to ensure that the CA executes always without privileges and has no way of obtaining them. Should this constraint not be respected the memory isolation would fail since any code running with privileges can access all of the memory and can also modify the configuration of the MPU itself.

As introduced in Section 3.1 any code running in handler mode always executes with privileges, thus we must ensure that the fortified applications always stays in thread mode. In order to execute exception/interrupt handlers without privileges we created a special “interrupt deprioritization routine” which activates immediately upon entering handler mode, performing a context switch and returning to thread mode while setting up the execution of the correct exception handler.

Unfortunately executing an application without privileges introduces a series of problems. As previously mentioned system configuration aspects such as

interrupts enabling and priorities are stored in the PPB which requires privileged access. Since the fortified application must be able to perform these essential configuration steps but cannot access the required memory, the microvisor needs to catch each attempted access and fulfill them in a controlled manner.

Another problem are system instruction. Contrary to PPB accesses, system instructions do not trigger any exception upon being executed without privileges. Instead, their behavior is different sometimes being completely ignored and acting as `NOP` instruction. To handle these, we created a custom module which is called during the compilation process of the fortified application and performs assembly-level instrumentation in order to highlight for the microvisor the presence of these instruction. This module is what we introduced in Section 2 as *Instrumenter* and together with the GNU ARM Embedded Toolchain forms what we refer to as *custom toolchain* or *modified toolchain*.

Regarding the enforcement of the strict access rules defined for the memory, the MPU (Memory Protection Unit) was extensively used. In particular, as the number of regions for the MPU of the target device was limited to 8, extensive use of the background region and subregions was made. In particular, when the CA is launched, it does not have access to areas other than its own. As soon as a call to a TA API is made, the MPU regions are reconfigured based on the Trusted Application called to allow access to the specific areas needed. When the API call terminates, the MPU is configured back to its original setup.

3.4 Previous Design

Before discussing the implementation of MCU-Fortifier we want to analyze the design of a previous version of MCU-Fortifier and the reasons why this was quickly abandoned.

Previous design iterations of MCU-Fortifier used a completely software-based approach. The goal was to provide memory isolation to a wider range of devices even those not equipped with an MPU. This design follows an approach similar to SμV[2] and μMSP[3] dividing the assembly instruction in two groups: *safe* and *unsafe* instructions. Additionally, unsafe instructions can be further classified as *implicit* or *explicit*.

An instruction is classified as *unsafe implicit* if its execution might result in a violation of the ACP. Unfortunately, this cannot be determined statically at compile time but only at runtime. The solution is to virtualize all unsafe implicit instructions, which can be done using assembly-level instrumentation in order to replace each instruction with a call to the correct virtual function.

However, this approach is not practical for the ARMv7-M architecture. In fact, ARMv7-M mainly uses only two instructions to access memory: load and store. These instructions are used very frequently throughout the program and almost all occurrences are classified as implicitly unsafe.

This result in the need of virtualizing an extremely large number of instructions resulting in heavy overhead, both in terms of code footprint and CPU time. In our initial implementation following this design, instrumenting applications resulted in a 3-fold increase in binary size.

Considering the limited resources of embedded devices, both in terms of memory and CPU, we decided to abandon this approach in favor of the current MPU-based design.

4 Implementation

In this section we will discuss the implementation, analyzing the different challenges that arose during development and how they were handled.

MCU-Fortifier was initially developed on an STM32WB55 board equipped with a Cortex M4 processor. Although MCU-Fortifier is conceived to work on Cortex M3 devices using a Cortex M4 board for development does not represent a problem. In fact, Cortex M4 devices constitute an evolution of Cortex M3 and as such are capable of running all of their instructions and also additional ones. For this reason, although the development was carried out on Cortex M4 hardware, only features available also on Cortex M3 devices were actually used.

Later in the development a decision was made to switch to a B-L475EIOT01A1 board, equipped with an STM32L475 MCU. This was done in order to have access to a wider range of communication peripherals (mainly 2.4GHz Wi-Fi) which allows to better demonstrate the OTA secure update capabilities that MCU-Fortifier provides. For the rest of the manual every mechanics or features specific of the target board used during development will be explicitly pointed out if necessary.

4.1 Removing Privileges

As previously mentioned, in order to enforce the ACP, the fortified application must always be executing without privileges. Code executing on an ARMv7-M architecture can "obtain" privileges in two different ways:

1. During *Thread Mode* by clearing[1] the *nPRIV* bit of the *CONTROL* register.
2. By entering *Handler Mode*, which as already explained in Section 3.1 always executes with privileges.

Preventing the application from clearing the nPRIV bit is quite easy. In fact, before the fortified application is executed, the microvisor code sets this bit to one. Once nPRIV has been set, it can only be cleared by code executing with privileges. Therefore, if an application always executes in thread mode there would be no way for it to obtain privileges and change this value once again. In practice this assumption is false because entering handler mode allows the fortified application to obtain privileges. For this reason we created a special *interrupt deprioritization* process which prevents the fortified application from ever handler mode.

4.2 Interrupt Deprioritization

Assuming that an application never enter handler mode is unrealistic. In fact, the architecture automatically enters handler mode when servicing interrupts/exception which are a fundamental feature used by almost all applications.

[1] Setting to zero

In the ARMv7-M architecture, interrupts are considered as a special type of exceptions. Furthermore each interrupt/exception is paired with its own *handler*, sometimes called "exception handler" or "interrupt service routine".

The association between each exception and the respective handler are stored in the *vector table* in the form of pointers to functions. During the normal execution of the application when an exception is triggered, the processor looks through the vector table, finds the handler associated with the exception, switches to handler mode and starts executing it. This is done automatically by the architecture without any need for the developer to write custom code. Furthermore the architecture follows the AAPCS (ARM Architecture Procedure Call Standard), therefore the exception handler can be implemented exactly as any "regular" function.

For this mechanism to work the processor must know where the vector table is stored, which can be specified through the use of the *Vector Table Offset Register* or *VTOR* in short. Normally, the table is stored at the lowest available address of flash memory, but it can be useful to change its location which, as will be explained later, is exactly what MCU-Fortifier does.

MCU-Fortifier Interrupt Deprioritization is composed of the following 3 routines:

1. *Exception Catcher*: catches an exception when it occurs. This function replaces the original exception handler normally called by the exception.

2. *Exception Simulator*: sets up the context (priority and privileges) in order to execute the original exception handler while being in thread mode.

3. *Exception Return*: performs a correct return form exception once the original exception handler has been executed.

4.2.1 Exception Catcher

Our end-goal is to prevent fortified application code from being executed in handler mode. Additionally, such mode is automatically entered upon triggering an exception. For this reason, MCU-Fortifier replaces all exception handlers in the vector table in order to execute a routine under its own control, this being the *Exception Catcher*.

In the actual implementation, the original vector table of the fortified application is not modified. Instead, the flash memory stores 3 different vector tables at all times:

- Vector table 1: stored at the beginning of the microvisor flash memory, contains "conventional" exception handlers used by the microvisor during the boot phase (e.g. communication with GPIO, I^2C busses, SysTick etc.).

- Vector table 2: stored after the previous vector table. This table is mostly filled with *Exception Catcher* entries replacing the conventional handlers. The microvisor sets the VTOR in order to point to this table immediately before beginning the execution of the fortified application.

- Vector table 3: stored at the beginning of the fortified user application memory. This is the original vector table of the fortified application. The VTOR never points to this table, instead the *Exception Simulator* access it during its execution in order to recover the original handlers that must be executed without privileges.

During the execution of the fortified application each time an exception is triggered, the architecture automatically looks up in vector table 2 (because pointed by VTOR) the handler to execute. This handler will be the *Exception Catcher* routine.

The Exception Catcher then performs the following actions:

- Disables interrupts, allowing to execute the rest of the actions in an atomic fashion.
- Saves key contextual information of the pre-empted code on the main stack pointer. The ARMv7-M architecture provides two different stack pointers, PSP (Process Stack Pointer) and MSP (Main Stack Pointer). During handler mode MSP is always used. In Thread mode instead, either of the two can be used depending on the value of the *SPSEL* bit the *CONTROL* register.

  In particular are saved: all the registers not automatically saved by the architecture, the *EXC_RETURN* value (indicating which mode and stack the processor should use after exception return) and the priority of the pre-empted code (in the form of the *BASEPRI* value of the special purpose *mask registers*).

  This information is necessary in order to correctly resume the execution of the pre-empted code after the original handler has been executed without privileges.
- Enables Thread Mode privileges, required for the correct execution of *Exception Simulator*.
- Creates an additional "fake return frame" on the MSP and performs the exception return. During the exception return, the architecture automatically unpacks this frame which is set-up in order to return the processor to Thread Mode (with MSP in use) and pass the control to *Exception Simulator*.

4.2.2 Exception Simulator

After the exception return, *Exception Simulator* beings executing (using Thread Mode, with privileges enabled and MPS in use) and performs the following:

- Sets the correct execution priority for the handler to be executed. In ARMv7-M, the execution priority governs whether another exception can pre-empt the current instruction stream or not.

- Re-enables interrupts.
- Removes Thread Mode privileges.
- Partially restores the original context (r4-r11) that was manually saved during Exception Catcher.
- Performs a branch and link (`BL` instruction) to the original exception handler that must be executed without privileges.

At this point, the original exception handler starts begin executed in Thread Mode without privileges while retaining the original execution priority. This way, if an interrupts with higher priority occurs, it will still pre-empt the instruction stream therefore maintaining the original behavior of the ARMv7-M architecture.

As any other function, the last instruction of an exception handler is a return, which normally begins the exception return process. However, since we removed the *EXC_RETURN* from the link register, the exception return process is not performed and the control is returned to Exception Simulator as wanted.

At this point ideally we would like to re-obtain the privileges and resume the instruction stream originally pre-empted. Unfortunately this is not straightforward due to two reasons:

- As previously mentioned, code running in Thread Mode without privileges cannot change the value of *nPRIV*. The only way to re-gain privileges is to enter Handler Mode and modify *nPRIV* form there.
- Performing the stack unwinding process, necessary to resume the instruction stream pre-empted, consists in writing content to 16 registers: r0-r12, LR, PC, xPSR. Unfortunately, the xPSR register (combination of IPSR, APSR and EPSR) cannot be explicitelly written to, in fact the ARMv7-M architecture implements it as WI (Write Ignored).

To solve both of these problem the Exception Simulator triggers a HardFault on purpuse in order to enter Handler Mode. A HardFault is a type of exception always enabled which can be easily triggered by performing an unprivileged access to the PPB. From the HardFault hanlder, the control is passed to the *Exception Return* routine.

4.2.3 Exception Return

The Exception Return routine is tasked with restoring the original context and resuming the pre-empted instruction stream.

It begins by performing a check on the PC to evaluate at which point during the execution the HardFault occurred. If the HardFault occurred at the end of the Exception Simulator routine, as expected, than it's treated as an "exception return request". Otherwise the HardFault must have been triggered by a different error so the control is returned to the HardFault handler.

If the HardFault was generated as a way to request an exception return, the routine does the following:

- Removes the stack frame created on exception entry. This was created due to Exception Simulator triggering the HardFault, of course we have no interested in resuming that execution since we triggered the exception on purpose as a way to properly return to the original instruction stream.
- Clear the *CFSR* (Configurable Fault Status Register) which contains information about the fault that was triggered. In our case its content was altered during after Exception Simulator triggered the HardFault.
- Unwind the remaining content of the stack stored manually during the Exception Catcher, while also restoring the original execution priority of the pre-empted code.
- Perform the exception return using the original frame, created automatically by the architecture during the transition form the originally pre-empted code to the Exception Catcher.

As a side note, during the execution of this routine there is no need to manually disable and re-enable interrupts in order to execute it atomically. In fact, when a HardFault is triggered, the corresponding handler executes automatically at the highest possible priority which ensures that our code will not be pre-empted.

4.2.4 Mask Register and Priority

System instructions such as `CPS` (Change Processor State) can modify the value of *PRIMASK* and *FAULTMASK*. These are normally used to disable interrupts because they raise the execution priority to a high value (0 or -1 respectively). In the Exception Catcher we set both *PRIMASK* and *FAULTMASK* to 1, but in the Exception Simulator we only consider *PRIMASK*. This is because *FAULTMASK* is automatically cleared by the architecture during a return from exception. Furthermore, Exception Simulator can keep *PRIMASK* set as a way to raise the execution priority to 0, which is required if the original exception handler was configured with that priority value.

4.2.5 Limitations

Due to the stack manipulation that occurs both in Exception Catcher and Exception Simulator, the stack differs from what would normally happen in a regular execution without interrupt deprioritization. For this reason, the fortified application handlers cannot rely on contextual information stored on the stack (e.g. by pushing data onto the stack before triggering the exception on purpose). This assumption is reasonable. In fact, interrupt handlers generally behave independently of the address on which the interrupt was triggered and tend to access global data structures using static references, which are stored intermixed with the assembly code. *SVCall* exceptions, however, violate this assumption and require the additional measures described in Section 4.3.3.

4.2.6 Virtual IPSR

Some OSes commonly used on embedded devices (e.g. Mbed OS), explicitly read the IPSR (Interrupt Program Status Register) to check whether an exception is being processed and branch to different code. Of course MCU-Fortifier must be transperent in this regard, ensuring that the behavior of the fortified application is the same as the non-fortified one.

Unfortunately, as previously described, the interrupt deprioritization routine executes handlers in thread mode (meaning that during their execution IPSR is set to 0). For this reason, if the check is performed during an interrupt handler, the result would differ when using the fortified application.

To solve this problem we implemented a *Virtual IPSR*, which is a global variable stored in the microvisor reserved RAM. The virtual IPSR reflects what the real value of the IPSR should be, meaning that during the execution of the deprioritized handlers it will still contain the correct number of the exception being handled even though the code is running in thread mode.

The virtual IPSR is manipulated in 3 different locations:

1. During the Exception Catcher, the current value of the virtual IPSR is saved on the stack, in order to be restored when resuming the execution of the pre-empted code.

2. During Exception Simulator, the value of the virtual IPSR is set to the correct exception that will be handled in thread mode.

3. During Exception Return, the value of the virtual IPSR saved on the stack is restored.

The implementation of the virtual IPSR is particularly straight forward. Every access to the "real" IPSR must be done using the system instruction `MRS` and `MSR` (respectively for read and write access). As we will explain in Section 4.3.2 these instruction must be virtualized because their behavior changes based on whether they are executed with privileges or not. During this virtualization, we can simply redirect every read/write access of the real IPSR to our virtual IPSR.

4.2.7 Integrity checks on exception return

The ARMv7-M architecture performs automatically different integrity checks during exception return. In particular, when returning from an exception to thread mode, the architecture checks that no other exception (beyond the returning one) are active. This would indicate a mismatch in the number of exception return, since when returning form an active exception to another one handler mode should be maintained.

Unfortunately, this checks directly affects our interrupt deprioritization routine. We will use an example to illustrate why this integrity check is problematic:

- Suppose that there are two interrupt A and B, with B having higher priority than A (therefore able to pre-empt its execution).

- Interrupt A is triggered and the Exception Catcher routine for interrupt A starts being executed (interrupt A becomes active).
- Before Exception Catcher disables interrupts (with the `CPS` instruction), interrupt B is triggered and pre-empts the execution of interrupt A (both B and A are now active). The Exception Catcher routine for interrupt B starts being executed).
- The Exception Catcher routine for interrupt B completes and an exception return is performed in order to pass control to Exception Simulator.
- This will result in the integrity check failing because B is trying to return in thread mode (in order to execute Exception Simulator) but A is still active.

Fortunately for us, this is the only integrity check relevant for our implementation. Additionally, this check can be disabled by setting the *NONBASETHRDENA* bit of the *CCR* (Control and Configuration Register). This operation is done at boot by the microvisor, before switching vector table and passing control to the fortified application.

4.3 Issues of Unprivileged Execution

Now that we have shown how to obtain the fortified application running always without privileges, we need to tackle the different problems that this practice creates. Mainly the problems are:

1. PPB accesses: unprivileged access to the Private Peripheral Bus trigger an HardFault.
2. System instructions: ARMv7-M system instructions behave differently depending whether executed with privileges or not.
3. SVCall exception: SVCall exception needs to recover SVC number by accessing stack and recovering PC value during exception entry. This cannot be done due to the limitations highlighted in Section 4.2.5.

4.3.1 PPB Access

Accessing the PPB is fundamental for every application. In fact, this region of memory stores global system configuration, such as the list of enabled interrupt and their priority, therefore we need a way to provide access to unprivileged applications. Simultaneously, we cannot allow access to every register in the PPB. For example, the fortified application should not be able to modify the configuration of the MPU, neither it should be able to access the VTOR.

Granting access to these registers would allow the fortified application to circumvent the ACP, either by removing MPU restrictions and allowing access to the whole system memory, or by removing the interrupt catching sequence

and allowing it to enter handler mode uncontrolled. For these reasons we need to control PPB accesses on a one-by-one basis.

To do this, a special "PPB Access Recovery" routine was created which exploits the fact that an unprivileged access to the PPB triggers an HardFault[2]. The PPB recovery routine is called inside the HardFault handler, after having checked that the HardFault was not triggered by an exception return request (Section 4.2.3). It performs the following:

1. Checks the PC address at the time of the HardFault and reads the instruction which caused it.
2. Checks whether the instruction is a variation of load or store (considering single byte, half word and word accesses).
3. Using a complex pattern matching sequence, decomposes the operands of the instructions (accessed address, registers that would be stored/loaded).
4. Checks the permission that the fortified application has on the accessed address.
5. Depending on the permissions, carries out the operation for the application by "simulating" it and advances the program counter.
6. Clears the CFSR and resumes the execution of the pre-empted code.

Permissions In the context of PPB access recovery, permissions refer to the following:

- Not a PPB access: the address the fortified application is trying to access is not part of the PPB. In this case the functions returns. The HardFault was not caused by an unprivileged PPB access, the HardFault handler will regain control and process it.
- Read, Write: the address accessed is indeed part of the PPB. The fortified application has both read and write access on it, the instruction should be executed. For example, this is the case for all NVIC configuration registers used for enabling, disabling and setting interrupt priorities.
- Read Only, Write Ignored: the address accessed is part of the PPB. The fortified application can read the value of such registers but should not be able to write to them. More specifically, during a write attempt, the program counter is advanced but the write operation has no effect, therefore continuing the execution of the fortified application while effectively skipping the store instruction. For example, this is the case for registers concerning MPU configuration and VTOR.

[2]To be precise a BusFault is triggered, if such exception is disabled the ARMv7-M architecture escalates the fault to a HardFault

Instruction Simulation By "simulation" we refer to the way the problematic instruction is executed by the microvisor. In particular, the execution is performed by creating a *virtual function*[3] in a specific region of RAM. In this virtual function there are only two instructions: the problematic load/store instruction, followed by a return. At most this only requires 3 half word of memory (48 bits): one word for the problematic instruction (which could be either 16 or 32 bits depending on the encoding), one half word for the return instruction (`BX LR`, always encoded using 16 bits).

This approach was chosen because it massively simplifies the implementation. In fact, the alternative would be to dissect every instruction down to every single bit of their encoding (e.g. evaluate whether offset, pre-indexed or post-indexed addressing mode is being used, checking whether the instruction updates the condition flags in the *APSR*, and many other aspects) and the effect manually. Naturally, such an approach would massively increase the complexity and footprint of the code. Instead our "simulation" approach is much simpler as it does not need to consider all of these aspects.

Unfortunately, our approach has a limitation: not every single instruction can be executed in this way. In order for the simulation to be accurate, we must restore the execution context present during the original instruction. Additionally, this context will be updated with new values generated during the simulation and will be used to resume the original instruction stream.

In practice we can do this only for the general purpose register r0-r12 and the *APSR*. Which leaves us with altered values (compared to the original instruction execution) for registers SP, LR and PC. SP register is altered by the stacking operations performed on exception entry and our additional function calls. Altering LR is necessary in order to perform the return from the virtual function to its caller. Finally, PC will differ because we are executing the virtual instruction (stored in RAM) instead of the original one (stored in its original location in flash memory).

Fortunately, none of these registers are normally used as operands in the instructions concerning us. Intuitively, we can understand how none of these registers (SP, LR and PC) would sensibly be used as source or destination. No real-world code would find use into reading content from the PPB and storing it in one of these register, nor it would be useful to store the content of these registers into any part of the PPB. Additionally these registers are also never used to address the PPB. PC and SP-relative addressing are indeed used in the ARMv7-M architecture, but these operation are useful only to access memory areas near the SP (to retrieve data on the stack) or near the PC (to retrieve data intermixed within the code). Obviously in no real-world application SP will point anywhere near PPB areas, instead it will point to volatile memory. Neither will PC, since the PPB and regions of memory in its proximity are not executable

Nevertheless, even if such corner cases were to happen our implementation is capable of detecting them and prevent violation of the ACPs.

[3] A temporary function stored in volatile memory

4.3.2 System Instructions

Although barely mentioned in the manual, the ARMv7-M system instructions behave differently depending on the level of privilege during their execution. In particular the architecture provides 3 system instructions:

- `CPS` (Change Processor State), is used to change the value of *PRIMASK* and *FAULTMASK* in the mask register. Any unprivileged execution of this instruction has no effect on the masks.
- `MRS` (Move to Register from Special register), allows to read the value of different registers (xPSR, MSP, PSP and mask register). When executed without privileges any attempt to read IPSR, MSP, PSP or mask register will return zero.
- `MSR` (Move to Special register from Register), allows to write values to special registers (APSR, MSP, PSP, mask register and CONTROL register). When executed without privileges, any attempted write to MSP, PSP, mask register and CONTROL register will be ignored.

Considering that our goal is to make the fortified application run without privileges while keeping its original behavior, we need a way of executing these instructions as if they were executed with privileges, while also being transparent to the fortified application. Unfortunately, unlike PPB accesses, the unprivileged execution of these instruction does not cause any "observable" exception.

For these reason it was necessary to implement custom assembly-level instrumentation, which allows the microvisor to detect any attempt of the fortified application to execute system instructions. After these have been detected, a mechanisms analogue to the PPB access recovery is triggered in order to carefully control and carry out the accesses on a one-by-one basis.

Instrumentation The assembly-level instrumentation is implemented through the use of a Python 3 script.

The purpose of this script is to insert an `SVC 0` instruction immediately before every system instruction. In this way, an *SVCall* exception is triggered before the execution of every system instruction, rendering them observable.

Additionally, the instrumentation decomposes every `IT` conditional instruction whose block covers system instructions. This is performed due to complexities that arises when dealing with conditional execution through `IT`[4] and the correct restoration of context during a return from exception.

The decomposition consists in removing the whole `IT` instruction and inserting, before each of the instructions covered by `IT`, a conditional branch. The purpose of the branch is to skip the instruction when the necessary conditions for execution are not met. This provides the same behavior of the originally replaced instruction while slightly increasing the size of the code.

[4] See *ITSTATE* in ARMv7-M Manual[1]

SVCall Exception At runtime, the `SVC` instruction inserted triggers an exception: *SVCall*. This exception is then handled by the microvisor *SVCall* exception handler (sometimes called *SVC Handler*), which recovers the execution of system instruction similarly to what explained in Section 4.3.1 for PPB accesses.

In particular the following operations are performed:

1. The value of the program counter prior to exception entry is recovered. And the instruction following the one that triggered the exception (`SVC 0`, inserted by our instrumentation) is recovered.

2. If such instruction is one of `CPS`, `MRS` or `MSR`, a "simulation" process analogue to the one described in Section 4.3.1 is performed, using a virtual function containing a copy of the problematic instruction (in this case a system instruction instead of a load/store).

3. Otherwise, an "original" (not inserted by our instrumentation) `SVC` instruction was encountered. The microvisor manually launches the interrupt deprioritization process by calling the Exception Catcher routine.

Additionally, during the simulation of system instruction the behavior of `MRS` and `MSR` is altered on purpose in the following way:

- In `MRS`, any read targeting the *IPSR* is redirected to our virtual IPSR. This renders the interrupt deprioritization process transparent to the fortified application as discussed in Section 4.2.6.

- In `MSR`, any attempt to write the *nPRIV* bit of the *CONTROL* register is ignored[5]. This prevents the fortified application from re-gaining privileges in thread mode execution.

4.3.3 SVC Instruction and SVCall Exception

The `SVC` (SuperVisor Call) instruction, is provided by the ARMv7-M architecture in order to facilitate the separation between an application and a possible operative system running on the device. The `SVC` encodings allows to specify an 8-bit integer (sometimes called *SVC number*) which can be used as a way to pass an argument to the *SVCall* handler.

TEE SVC Handler The Trusted Execution Environment (TEE) heavily relies on the Supervisor Call (SVC) mechanism to simulate privileged instructions, change the context, privileges level, and call the appropriate GP API. In particular, the SVC Handler of the Microvisor can be called and executed for the following reasons.

[5]Done automatically by the architecture when MSR is executed in thread mode, however simulation occurs in handler mode so we need to manually alter its behavior

- Simulation of a system instruction (the SVC was called due to the CA code instrumentation): in that case, the instruction is simulated without priviledges and the exception return procedure is performed;
- Call of a TEE Client API: the SVC was used by the client application to perform a Global Platform Client API call (SVC number is in the range 0-4). In that case, the API call is used to invoke the requested TA and the exception return is performed;
- Call of a TEE Core API: the SVC was called by a trusted application to perform a TEE Core API call (SVC number is in the range 5-50). In that case, the API call is performed and the execution is returned to the trusted application;
- CA SVC call: the original SVC_Handler is needed and the control should be passed to the appropriate deprioritized handler.

CA SVC Handler The way *SVCall* handler of the CA is supposed to recover this argument is the following:

- Read the frame created during exception entry and retrieve the pre-entry PC value.
- Read the instruction stored at the address specified by PC (the encoded `SVC` instruction).
- Extract the 8-bin integer by reading the least significant byte of the instruction.

Unfortunately, this goes directly against the limitation (discussed in Section 4.2.5) which prevents deprioritiezed handlers from using the stack values stored by the architecture during exception entry. Due to the stack being manipulated by our interrupt deprioritization routines, *SVCall* handler does not find the value of the PC in the expected location.

This is an instance where source code modification to the fortified applications is required, albeit being minimal. In fact, the microvisor *SVCall* is designed to recover the SVC number (during the microvisor own *SVCall* handler) and pass it to the deprioritized handler (through *Exception Catcher* and *Exception Simulation* routines) as the first argument using register `r0`. For this reason, the source code modifications are relatively simple, and in particular, if the *SVCall* is written in C, it suffices to:

1. Change the signature of *SVCall* in order to expect an 8-bit unsinged integer as argument (e.g. `void SVCall(uint8_t SVCnum)`).
2. Remove from *SVCall* the source code originally used to retrieve the SVC number.

4.4 Memory Protection Unit

As previosly discussed, the Memory Protection Unit (MPU) plays a fundamental role and is used, in combination with the interrupt deprioritrization routines, to enforce the ACP.

4.4.1 Using the MPU

Before we discuss the default configuration of the MPU provided by MCU-Fortifier, we need to briefly introduce how the MPU works. The MPU allows us to divide the addressable memory space in different sections called *MPU regions*. Each of these region is associated with a set of attributes that specify:

- *Region Enable*: whether the region is currently enabled or not.
- *Region Number*: numeric index of a region. Higher numbered regions have priority over lower numbered ones.
- *Size*: size of the region. Must be defined as a power of two (e.g. 512, 1024, 2048 bytes).
- *Base Address*: starting address of the region. Must be naturally aligned based on the region size (e.g. a 2048 bytes sized region must also be 2048 bytes aligned).
- *Access Permissions*: read and write permissions. Can be specified separately for privileged and unprivileged code, with the constraint that privileged code permissions are a super set of unprivileged code ones.
- *Execution*: whether the region can be executed as code. A pre-requisite for execution is the read access, needed in order to fetch the instructions before execution.
- *Sub Region Enable*: allows further sub-divide the region into smaller chunks which can be enabled or disabled individually. Each chunk has inherits the permission from the base region.
- *Type Ext (Tex)*, *Sharable*, *Cachable*, *Bufferable*: different combinations of these parameters are set depending on the memory type and properties. For example, "strongly ordered shared memory" can be obtained by setting the parameters to respectively: `0 1 0 0`. For more information, consult the ARMv7-M Architecture Manual[1] and the STMicroelectronics AN4838-Application Note[4].

When a memory access is performed, the MPU matches the requested address against the different enabled regions. If there is a single matching, the configuration of the region is checked (including if the address match the subregion of that region allocated for that application) in order to evaluate whether the access should be allowed or not. Additionally the regions can be overlapping, meaning that multiple regions might match with a single access. In this

case, the rules of the region with highest *region number* are applied. Accesses that do not match any memory regions trigger a *MemManage* exception.

Setting the *PRIVDEFENA* bit of the *MPU_CTRL* register allows non-matching accesses and adresses belonging to disabled subregions to use the default memory map specified in the ARMv7-M Architecture Manual[1]. However, this only applies to accesses performed by privileged code. Unprivileged code is not allowed to use the default memory map when the access does not match any region.

Finally, any access to the Private Peripheral Bus (PPB) always uses the default memory map, regardless of how the MPU is configured. Meaning that the MPU is restricted on how it can change memory attributes of the system address space.

4.4.2 MCU-Fortifier Memory Map

Before we outline the configuration of the different MPU regions, let's begin by examining MCU-Fortifier memory map for our STM32L475 MCU, detailed in Listing 1.

```
MEMORY
{
FLASH_BOOT (rx)          : ORIGIN = 0x08000000, LENGTH = 0x20000
FLASH_TA1 (rx)           : ORIGIN = 0x08020000, LENGTH = 0x3C000
FLASH_SECURE_TA1(rw)     : ORIGIN = 0x0805C000, LENGTH = 0x4000
FLASH_TA2 (rx)           : ORIGIN = 0x08060000, LENGTH = 0x3C000
FLASH_SECURE_TA2(rw)     : ORIGIN = 0x0809C000, LENGTH = 0x4000
FLASH (rx)               : ORIGIN = 0x080A0000, LENGTH = 0x58000
FLASH_BOOT_NOPRI (rx)    : ORIGIN = 0x080F8000, LENGTH = 0x8000
RAM_BOOT (xrw)           : ORIGIN = 0x10000000, LENGTH = 0x20
RAM (rw)                 : ORIGIN = 0x10000020, LENGTH = 0x7FE0
RAM_TA1 (xrw)         : ORIGIN = 0x20000000, LENGTH = 0xC000
RAM_TA2 (xrw)            : ORIGIN = 0x2000C000, LENGTH = 0xC000
}
```

Listing 1: MCU-Fortifier memory map linker script (memory_map.ld)

As we can see there are 11 different memory regions:

- `FLASH` and `RAM` are the regions which should be accessible to the fortified application.

  In particular `RAM` is one of RAM banks available on the board and the fortified application must have read and write access to it, but no execution permission in order to prevent possible self-writing code or remote code injection attacks.

  On the other hand, `FLASH` is the region of flash memory where the fortified application is stored. As such, the fortified application should be able to read and execute instructions from this region. Additionally, it should not

be able to write to this region, in order to prevent remote code injection and also to protect the integrity of the code.

The microvisor and the TAs, on the other hand, has full access over these regions. In particular, the TAs have access due to the current implementation of shared memory.

- `RAM_BOOT` is the region of RAM reserved purely for microvisor usage. It contains the value of the Virtual IPSR (Section 4.2.6) and the memory used for instruction simulation (Section 4.3.1 and 4.3.2). The fortified application and the trusted applications should not be able to access this region.

- `FLASH_BOOT` is the region of flash memory where the microvisor is stored. This region should be accessible in read,write and execute modes only by the microvisor. The fortified application and the trusted applications should not be able to access it.

- `FLASH_BOOT_NOPRI` this region contains microvisor code that must be executed without priority. An example is the code used to bootstrap the execution of the fortified application after the thread mode privileges have been disabled by the microvisor, the shared libraries and the entry points for the Global Platform Core API. This region should be readable and executable by unprivileged software. As a consequence, fortified applications and trusted applications are also able to access this region meaning that sensitive data should never be stored here.

- `FLASH_TAx`, `FLASH_SECURE_TAx` and `RAM_TAx` are the regions which should be accessible to the trusted application number .

  In particular `RAM_TAx` is (part of) the other RAM bank available on the board and the trusted application "X" must have read and write access to it, but no execution permission in order to prevent possible self-writing code or remote code injection attacks. The physical bank is shared with the `RAM_TAy` using subregions and activating only the part of the RAM used by the current TA. To recap, the TA "Y" does not have access neither for read, write or execute on this section, as well as the fortified application. The microvisor can perform every operation on this memory area.

  On the other hand, `FLASH_TAx` and `FLASH_SECURE_TAx`are the region of flash memory where the trusted application "X" is stored and store its secure data. As such, only the trusted application x should be able to read and execute instructions from the flash region and read and store data in the flash secure one. As for the RAM, the TAy and the fortified application have no permission on that area, while the microvisor has them all.

The ARMv7-M architecture does not provide a standard number of MPU regions supported by the processor. Instead, each manufacturer implementation

might provide a different number. In our case, the STM32L475 processor is capable of providing support for up to 8 MPU regions (numbered from 0 to 7). Each region can be further divided into 8 sub-regions of the same size (numbered from 1 to 8), that can be activated or deactivated based on the needs.

Although we have provided a list of different memory ares, these are not the only areas that we need to consider using the MPU. In fact, every memory address performed by the fortified application must be covered by an MPU region (otherwise a MemManage exception would be generated), this includes also memory areas such as: memory mapped peripherals, system memory, option bytes and OTP (One Time Programmable) memory normally available on STM32 boards.

In general, the MPU configuration we provide is specific to the STM32L475 MCU and based on the memory map described in its reference manual[5]. Additionally, we have tested this configuration with multiple fortified applications and we have observed that all accesses are covered by the different regions. Nevertheless, our testing is not exhaustive by any mean and special purpose applications might access memory regions not covered by any MPU region.

4.4.3 MPU Configuration

Since some regions of memory previously outlined are not sized as powers of two and are neither naturally aligned, we cannot simply map them one-by-one to MPU regions. Instead, we use a set of MPU regions to grant permissions to the unprivileged code in a coarse-grained fashion. Then, we use a second set of MPU regions, overlapping with the first set, in order to restrict access to the memory areas that should be accessible only by privileged code. To ensure that the fine-grained restrictions applied by the second set have priority over the first set, we use regions with a higher MPU number.

In our default configuration, used during the startup of the system and when the system normally executes the fortified application (not the trusted ones) we identified the following 8 MPU regions, with MPU numbers ranging from 0 to 7:

0. Flash memory: region covering the entirety of the flash memory, with a base address `0x08000000` and size of `0x100000` (1 MB). This area contains, respectively, the code and data for the CA, both TAs, the TEE and TEE area accessible without priority and it is readable and writable by privileged code, but only readable by unprivileged. Execution allowed. To restrict the possibility of unstrusted application on this huge are, the Region 1 is used to make the Microvisor reserved flash unaccessibile and subregions to make the flash for TAs inaccessible. This requires to have dedicated sections for the TAs that are aligned with the size of one subregions, each of size `0x20000` (128KB), or a multiple of that. During this phase, the subregions assigned to the TA1 and TA2, namely the number 2,3,4 and 5 should be disable in order to prevent access, even in read only mode, to untrusted applications.

1. Microvisor reserved flash: region covering the flash memory reserved for microvisor usage (excluding `FLASH_BOOT_NOPRI`). With base address `0x08000000` and size `0x20000` (128 KB). Readable and writable by privileged code, not accessible by unprivileged. Execution allowed. This region overlap the previous one and restricts CA and TAs form accessing microvisor reserved flash.

2. RAM for TAs: region covering the entirety of bank 1 RAM. With base address `0x20000000` and a nominal size of `0x20000` (128 KB). The real allocatable size of `0x18000` (96 KB) would suffice, but is not a power of two so we select the smallest immediately greater power of two. In this configuration the MPU region "overflows" the RAM but it does not represent a problem since the following memory is reserved[6] and should never be accessed anyway. Readable and writable by privileged code only during the execution of CA. Execution not allowed.

3. RAM for CA: region covering the entirety of bank 2 RAM, with a base address `0x10000000` and a size of `0x8000` (32 KB). Readable and writable by both privileged and unprivileged code. Execution not allowed.

4. Microvisor RAM: region covering the RAM reserved for microvisor usage. With base address `0x10000000` and size `0x20` (32 B). It partially overlaps the RAM region for the CA and imposes stricter privileges, thus the section is readable and writable by privileged code, not accessible by unprivileged. Execution allowed, needed for the execution of the virtual function explained in Sections 4.3.1 and 4.3.2.

5. Peripheral address space: memory space used to access memory mapped peripherals. With base address `0x40000000` and size `0x40000000` (1 GB). Readable and writable by both privileged and unprivileged code. Execution not allowed. Configured as "device" memory.

6. Peripheral address space (continuation): configured exactly as the previous entry but with base address `0x80000000`. Unfortunately, using a single MPU region with size of 2 GB caused problems, therefore both region 4 and 5 are necessary.

7. System memory, option bytes, OTP area: region covering multiple features of STM boards normally used by applications. With base address `0x1fff0000` and size `0x10000` (64 KB). Readable and writable by privileged code, ready-only by unprivileged. Execution not allowed.

With these regions, we can obtain the properties highlighted in Section 4.4.2.

We can notice how the `FLASH_BOOT_NOPRI` does not require a region on its own. In fact, MPU region 0 already covers this memory section and allows the execution of unprivileged code.

[6] As indicated by the MCU reference manual[5]

The attributes TEX, Sharable, Cachable and Bufferable for all the regions discussed are set accordingly to the specification of the ARMv7-M manual for the system address map[1]. A comprehensive explanation for these attributes can be found in the STMicroelectronics AN4838-Application Note[4].

When a TEE Client API is called, triggering the execution of a Trusted Application (TA), the MPU must be reconfigured to ensure the correct access control policy is enforced for the target TA (e.g., TA1 or TA2). The key changes to the MPU regions include:

0. Flash memory: the subregions associated with the target TA should be enabled (namely 2 and 3 for TA1 and 4 and 5 for TA2), while those corresponding to other TAs should remain disabled. The other subregions (associated with the TEE or with CA) are left unchanged.

1. Microvisor reserved flash: no changes.

2. RAM for TAs: the region is divided into subregions of 16KB (considering a nominal size of 128KB for the whole region), thus only the first 6 subregions can be used (given the physical size of 96KB). For this reason the the first three subregions are dedicated to TA1 (namely 1,2 and 3) while the following three (4, 5 and 6) are dedicated to TA2. The subregions associated with the target TA should be enabled, while those corresponding to other TAs should remain disabled. The last two subregions can not be used.

3. RAM for CA: no changes, as this regions contains also the shared memory data between CA and TA.

4. Microvisor RAM: no changes.

5. Peripheral address space: no changes.

6. Peripheral address space (continuation): no changes.

7. System memory, option bytes, OTP area: no changes.

Once the API call is completed, the MPU is restored to its original configuration, allowing the CA to resume normal execution.

For a TEE Core API call originating from a Trusted Application, no MPU reconfiguration is required, as Core APIs are services provided by the TEE and execute directly with the necessary privileges.

4.5 Permanent Storage

When using MCU-Fortifier, the microvisor has a need for an area of memory that maintains its content through a board reset. This area is called “permanent storage” and is used for storing error logs, among other information, before the device is restarted. Additionally, after these error logs have been reported to an update server, they are wiped from memory.

Given such requirement, the best candidate for this area is flash memory. Unfortunately, using flash memory to store information in such way (logging data and then deleting it successively) is heavily dependent on the flash controller, therefore the device manufacturer.

In our specific case, with the B-L475EIOT01A1 board, the STM flash controller allows in-application programming of the flash albeit subjected to some restrictions. In particular:

1. Programming and erasing flash memory is done by accessing the flash controller peripheral.
2. Write access to flash memory are carried out using double words (writing 64 bits at the time).
3. Only "empty" flash memory can be programmed, attempting to program any non-empty flash memory results in an error. A word of flash memory is considered empty when it stores the value `0xffffffff`.
4. Flash memory can be erased in mass (completely erasing both banks of memory available on the device), or on a page-by-page basis.

Due to restriction 3 and 4, although we might only need a few bytes to store the information needed, we must reserve an entire page of flash memory. The B-L475EIOT01A1 board is equipped with 1 MB of flash memory split in 2 banks. Each bank contains 256 pages of 2 KB. This means that our permanent storage will be at least 2 KB (2048 bytes) in size.

In the current implementation a single page of 2 KB is more than sufficient to store:

- MCU-Fortifier Activation Code (32 bytes).
- Fortified application size (4 bytes).
- Error ID (4 bytes).
- Error Data (variable).

Each of these fields and their purpose will be described in future sections of this document.

4.6 TEE Client API

A TEE Client API is invoked by the Client Application when it needs to use a service provided by either a TA or the TEE. To comply with the GlobalPlatform standard, each TA must implement the following required TEE Client APIs to allow the communication with the Client Application.

- Create an entry point for the CA: *TEE_Result TA_CreateEntryPoint(void);*

- Open a session to invoke the functionalities of the TA:
  *TEE_Result TA_OpenSessionEntryPoint(uint32_t param_types, TEE_Param params[4], void **sessionContext);*
- Execute a command inside the TA:
  TEE_Result TA_InvokeCommandEntryPoint(void sessionContext, uint32_t commandID, uint32_t paramTypes, TEE_Param params[4]);*
- Close the current session:
  void TA_CloseSessionEntryPoint(void sessionContext);*
- Destroy the entry point for the CA: *void TA_DestroyEntryPoint(void);*

When one of these Client APIs is called using the corresponding function on the CA side (namely *TEEC_InitializeContext*, *TEEC_OpenSession*, *TEEC_InvokeCommand*, *TEEC_CloseSession* and *TEEC_FinalizeContext*, several steps must be performed to ensure the successful execution of the TA.

The CA functions use an *SVC Call* with a number in the range of 0 to 4 to transfer control to the TEE. Additionally, a shared memory data structure, stored in the CA memory area, is used to pass parameters. Specifically, the address of this data structure is stored in register R4 (after saving its original context) and is later retrieved by the appropriate function once the SVC handler is invoked. In the case of a command execution request, the command ID is stored in another register, R5, and recovered in the same manner. All the logic for this process is encapsulated in the library files (*tee_client_api.c*, *tee_client_api.h* and *tee_common.h*) which should be included in each CA. After the *SVC Call* invocation, various steps take place inside the Microvisor.

1. The SVC handler intercepts the request, recognizes that a Client API call was made, saves the context, and creates a fake return frame.
2. The Microvisor exits the SVC handler and the corresponding handler mode (still maintaining the privileges), passing control to the *call_TA* function. This function is responsible for:
   (a) Transforming the parameters from the CA format to the TA format.
   (b) Identifying which TA is being requested by the CA.
   (c) Reconfiguring the MPU to run the targeted TA, as described in subsubsection 4.4.3.
   (d) Calling the correct entry point based on the TA number and the SVC number.
   (e) Adjusting the stack pointer, as TAs use the Process Stack Pointer (PSP) during execution, while the CA always uses the Main Stack Pointer (MSP).
3. The API call is executed, and the TA code runs without privileges.
4. The output parameters of the TA are copied back and saved.

5. The Microvisor regains privileges by simulating a HardFault exception and verifying the source of the error (if it is known, the exception was intentionally triggered).

6. The exception return is executed, and the MPU is reconfigured to its original state, allowing the CA to resume execution.

7. The control returns to the original point where the API was called and the context before the SVC is restored.

Note that TAs have standard entry point names, as each TA should offer the same API to communicate with the CA. Since both TAs are compiled alongside the TEE, this poses a problem: multiple instances of a function with the same name would cause a linking error. To address this, the functions in *call_TA* are defined as *entryPointName_TaNum*, where TaNum is either 1 or 2, referring to the respective TA. However, within each TA's source code, functions can still be referenced without the suffix. This is achieved by employing the Makefile, which is configured to replace function names in each TA's source code. Specifically, for TA1, the function TA_CreateEntryPoint is replaced with TA_CreateEntryPoint1, while for TA2, it is replaced with TA_CreateEntryPoint2. This renaming applies to all five TEE Client API functions inside a TA.

4.7 TEE Core API

The TEE core API are entry points for basic security services offered by the TEE that can be used only by Trusted Applications (TAs). CA can not use directly TEE Core API, and controls are enforced to prevent this.

The core APIs exposed by the TEE are divided into different categories, based on their scope. The main ones are the following.

- Operation Management functions (e.g., AllocateOperation, FreeOperation, SetOperationKey);
- Object Management and Secure Storage functions: operations on transient objects (AllocateTransientObject, FreeTransientObject, PopulateTransientObject, InitRefAttribute, InitValueAttribute, etc.) and on persistent objects (CreatePersistentObject, OpenPersistentObject, CloseAndDeletePersistentObject1, ReadObjectData and WriteObjectData);
- Memory Management functions (e.g., Malloc, Free, MemMove);
- Random Value generation and Key Generation functions (e.g., GenerateKey and GenerateRandom);
- BigInt Functions (e.g., BigIntConvertToS32, BigIntMod, BigIntConvertFromOctetString);

- Symmetric Cipher Functions (e.g., CipherInit, CipherUpdate, CipherDoFinal);
- MAC functions (e.g., MACInit, MACUpdate, MACComputeFinal);
- Message Digest Functions (e.g., DigestUpdate, DigestDoFinal, DigestExtract)
- Asymmetric functions for Sign and Verify operations (e.g., AsymmetricSignDigest and AsymmetricVerifyDigest).

More details on the implemented functions and supported ciphers are listed in the README file of the code repository [7].

Since the TEE Core API code is stored inside the TEE and is therefore inaccessible to unprivileged code and since TAs run without privileges and the Core API must execute in the secure world with privileged access, there needs to be a mechanism for exposing the Core API functions that TAs can call and for running the API functions with privileges. A list of entry points that act as wrappers placed in an unprivileged memory region that allow switching context to gain privileges before invoking the actual Core API implementations and managing input and output parameters is the solution adopted to solve this problem.

This list, which can be seen as a library, is implemented in the *tee_core_api.c* file and stored in an unprivileged memory area, deeply relying on the SVC Handler. When a TA calls a TEE Core API function, the entry points for Core API are called and, transparently, the context is switched to the secure world by using the SVC instruction. Before performing the SVC, the function parameters are copied into CPU registers (R4–R11), with the original values temporarily saved before being overwritten. The SVC instruction is then called with a number ranging from 4 to 50. Based on the SVC number, the secure world maps the request to the corresponding internal function that implements the TEE Core API. The real implementations of the TEE Core APIs are on the *internal_core_api.c* file and each function uses the same name as the Core API it implements with an additional "internal" prefix (e.g., *internal_coreAPI*). Once the API execution is complete, the return value is placed in R4, the privileges are dropped and the control is returned to the original caller.

[7] `https://github.com/crosscon/baremetal-tee/tree/main/MPU-version`

References

[1] ARM. *ARM v7-M Architecture Reference Manual*.

[2] A. Mahmoud, C. Bruno, J. Bart, H. Danny, and D. Wilfried. Sμv—the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019. doi: 10.1109/TDSC.2019.2928541.

[3] G. Michele. μmsp: Design and implementation of a software-based memory isolation technique for msp430-based embedded systems. 16(5):885–901, 2020. doi: 10.1109/TDSC.2019.2928541.

[4] STMicroelectronics. *AN4838 Application Note - Managing memory protection unit in STM32 MCUs*, September 2021.

[5] STMicroelectronics. *RM0434 Reference manual - Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® Low-Energy and 802.15.4 radio solution*, June 2021.