LIPS: Lightweight Intra-Mode Privilege Separation against New Control Hijacking Attacks on RTOS Task Sandboxing

ABSTRACT

Task sandboxing, as a fundamental security mechanism in x86 computer systems, is achieved by address space isolation and virtualization, among others. However, the tight resource constraints and software design patterns disable IoT devices to use the above mentioned sandboxing mechanism.

This paper takes ARM Mbed as an example to research the task sandboxing techniques in Exokernel-based RTOS, and presents the following three new findings: 1) A new form of control hijacking attack against Mbed task sandboxing. 2) Although the existing Mbed memory isolation mechanisms are incompetent to handle this attack, a lightweight intra-mode privilege separation mechanism can be invented to defeat it. Based on this, we present LIPS, which further uses MPU to divide "the memory (i.e., Flash and RAM) accessible in unprivilege mode" into Inside and Outside domains. The domain switching is ensured by Domain Notification Calls. 3) Thorough evaluation and experimental tests are carried out and LIPS proves to be able to effectively defend the widely existing control hijacking attack against Mbed task sandboxing and with small runtime overheads and good portability.

1 INTRODUCTION

TASK sandboxing, as a fundamental security mechanism in a x86 computer system, is achieved by address space isolation [15] and virtualization [3], among others. This paper, however, will focus on the task sandboxing problem towards ubiquitous IoT devices [5, 6, 52].

Nowadays, most of the commercial IoT devices use low-end and low-power ARM Cortex-M processors. However, the tight resource constraints and software design patterns make OSes that run on Cortex-M processors unable to employ virtual memory. Besides, these OSes' codes only run in Flash and are set to be unwritable when OS is running, and the data are stored only in RAM and are modifiable [29]. What's more, x86 processors contain complicate and mature security components (e.g. MMU, ring0-ring3 [50]), while in Cortex-M processors corresponding security components are less powerful. For example, MPU (Memory Protection Unit), which is proposed by ARM in its lightweight OS' security design, can only enforce read, write, and execute permissions on a fixed number of regions of the physical memory[12].

These limitations cause task (process) sandbox isolation mechanisms between x86 and Cortex-M processors different in the following three aspects. [25, 30]. First, in x86 processors, a sandboxed process cannot directly call other sandboxed processes' codes, while in Cortex-M processors the sandboxed task can call codes of other sandboxed tasks running in the same privileged mode. Second, in x86 processors a sandboxed process cannot read/write other processes' data. However, whether one task can read/write the data of a task in another sandbox depends on whether MPU is dynamically reconfigured in Cortex-M processors. Third, x86 processors isolate every sandboxed process from each other through 4 privilege levels (ring0-ring3) and MMU, and finally realize the two-way sandbox isolation. However, as Cortex-M processors lack MMU and only support 2 privilege levels (privileged and unprivileged mode), only one-way sandbox isolation can be achieved. For example, by OS configuring MPU, the whole RAM spaces are roughly set into two areas, called the "Accessible in Unprivileged Mode" RAM (which we denote as AUM RAM) and the "Only Accessible in Privileged Mode" RAM (which we denote as OAPM RAM). And the tasks data sandboxed in AUM RAM is exposed to the task whose data are sandboxed in OAPM RAM, while the exposure is shielded the other way around. Based on these differences, the vunlerbility of task sandbox isolation mechanism in Cortex-M processors is unique [13, 14], and cannot be fixed by the similar security solutions available in x86 processors.

In fact, the implementation and operation of IoT task sandboxing greatly rely on the OS design. One of the most popular type of task sandboxing mechanisms is that implemented by FreeRTOS (Free Real-Time Operating System/RTOS) [7] and the variants of it. According to tasks' pre-set memory accessibility permissions, FreeRTOS uses MPU to divide all tasks into unprivileged tasks and privileged tasks. Both privileged tasks and lib codes of FreeRTOS (RTOS_Lib) run in privleged mode and their data are stored in OAPM RAM, while unprivileged tasks run in unprivileged mode with their data stored in AUM RAM. Depending on these permissions, unprivileged tasks can not directly modify data stored in OAPM RAM. However, an unprivileged task can exploit the buffer overflow vulnerablity to compromise the privileged codes (i.e., RTOS_Lib) and disable the MPU [8, 12, 56].

Another type of operating system is called Exokernel-based RTOS by some researchers [4] and has been widely adopted in more and more industry for its unique merits [54]. These merits include: 1) the security enhancement by separating memory management from RTOS_Lib and make RTOS_Lib run in unprivileged mode, 2) the performance improvement by tasks directly calling RTOS_Lib in unprivileged mode, and 3) the flexibility increase by tasks manipulating peripherals efficiently [18]. Accordingly, Exokernel based RTOS, such as Mbed [33], adopts the second type of rising-Exokernel-design [17] based solution. Mbed creates uVisor [34] for memory management. uVisor always runs in privileged mode, whose data are unmodifiable in any time, so attacks by modifying its data are invalid. The specialness of uVisor design is that it make all tasks, both Secure Tasks (STs) and Regular Tasks (RTs), which are classified by the closeness of Mbed security, run in unprivileged mode. But, STs' data are classified as sandboxed and stored in OAPM RAM, while RTs' data are stored in AUM RAM. To solve the contradiction that a ST running in unprivilged mode has to operate with its data stored in OAPM RAM, uVisor make sure only when a ST runs, ST's data will be brought back to AUM RAM. Due to this strong isolation, exokernel-based RTOSes prevent any vulnerable codes in unprivileged mode to directly affect privileged modules.

Besides, some researchers introduce a brand new task sandboxing mechanism, employing compiler-based instrumentation scheme to create the sandbox for particular compartment in a program [11, 12, 22]. When a compartment is called by others, its instrumentation code will execute particular Supervisor Call (SVC) instructions to dynamically change the memory accessibility by reconfiguration of MPU. While such designs are not much seen in industry yet.

In this paper, we research the task sandboxing techniques on Mbed as an example of Exokernel-based RTOS. During the research, we craft two attacks to assess the vulnerability of Mbed. To be specific, the first attack is named Context Table Remapping (CTR) which dexterously utilizes address pointers in the context table to perform a novel form of control hijacking attack against Mbed task sandboxing. The vulnerability of the context talbe is known to some reseachers, but CRT exploiting this vulnerability is not reported in the literiture. The second attack is named Function Code Reuse (FCR). FCR is proposed to counter a theoretical defense scheme for CTR, which stores the context table into the OAPM RAM. However, the context table initialization process depends on RTOS_Lib codes running in unprivileged mode. FCR can maliciously recall the newly created SVC functions, which configure the context table to AUM RAM for initializing the context table, and launch CTR again.

The further research discovers that effectiveness of two attacks is the result of their exploiting the Mbed task management process. Mbed task management contains two phases: the first phase is locating the target task, and the second is uVisor's memory operation on the task data. In the second phase Mbed must seek the context table for address pointers of memory functions in uVisor. However, the context table will be tampered by CTR anyway no matter it's located in OAPM RAM or not, and this is how attackers break Mbeds sandbox isolation mechanism and attack IoT devices. Based on this observation we introduce a new isolation mechanism: build an extra protection domain to separate RTs from the context table which can be exploited to invalidate the Mbed sandboxing, making up for the insufficiency of privilege isolation in Exokernel-based RTOS.

We present LIPS (Lightweight Intra-Mode Privilege Separation), which uses MPU to divide "the memory (i.e., Flash and RAM) accessible in unprivilege mode" into Inside and Outside domains. Context table in Inside domain is well protected from CTR attack by forbidding its accessibility from RTs in Outside domain. Necessary switching between the two domains is ensured by Domain Notification Calls (DNCs) that can be called by STs only. LIPS uses MPU to set the "Nonexecutable ("execute never" in ARM's terminology) in Unprivileged Mode" Flash region (which is denoted as NUM Flash), called Notification Guard area. This area stores the only entries of each DNC. So that any tasks' calling these entries will cause a memory exception, and its runtime information is frozen and role-identified by Reference Monitor which only allows the ST's calling request and blocks that of RT's. What's more, the codes and data of LIPS are embedded in the uVisor, and can't be modified by either STs or RTs. Through these methods the security of Exokernel-based RTOS task sandboxing is improved. Because our technology doesn't need instrumentation, this paper has nothing to do with compiler-based sandbox creation scheme.

In summary, our contributions are as follows:

- By crafting two attacks (Context Table Remapping (CTR) attack and Function Code Reuse (FCR) attack) via uVisor on Mbed as an example, we uncover the vulnerabilities of task sandboxing mechanism on Exokernel-based RTOS. Exploiting the vulnerabilities attackers can hijack the privileged control flow and affect secure number protected in the sandbox.
- The task sandboxing design under two-privilege-mode-based memory isolation in Cortex-M processors, even the advanced Exokernel-based RTOS task sandboxing design, has the risk of being broken by CTR or potential FCR attacks.
- We propose an original intra-mode privilege separation mechanism named LIPS that uses MPU to create protection domains, and design Reference Monitor which only permits STs' domain switching.
- We perform a thorough evaluation and experimental tests of LIPS. The experimental results prove that LIPS not only can effectively defend the widely existing CTR and FCR attacks, but also has small runtime overheads and good portability.

2 BACKGROUND

2.1 Cortex-M Architecture

2.1.1 Memory Protection Unit. LIPS's isolation employs basic Memory Protection Unit (MPU) operations in Cortex-M processors. MPU can add an access control layer over the physical memory, and defines different memory regions that own diverse access permissions under both privileged and unprivileged modes, but memory is still addressed by its physical addresses. Its actual design varies in different platforms. For example, NXP FRDM-K64F[1], which is our prototyping platform, observes the following MPU design rules: (1) it supports 12 MPU regions, numbered from 0 to 11; (2) the MPU must define the access permissions of every region under both modes; (3) memory regions are allowed to overlap each other and the overlapped area will be endowed with all the assigned permissions [51].

For the remainder of this paper we will use the following notations to describe permissions for a memory region. (P-R/W, U-NX) means readable and writable permissions

[1]https://os.mbed.com/platforms/FRDM-K64F/

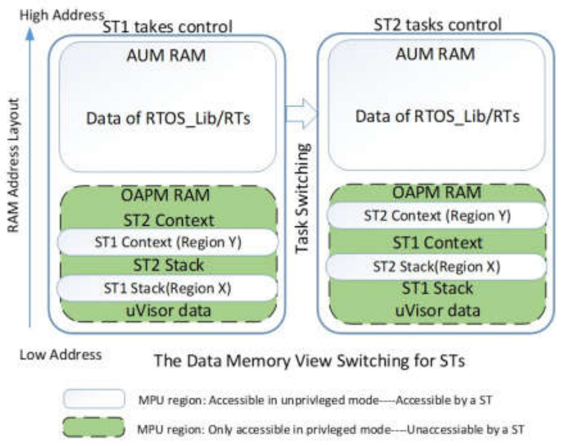


Figure 1: When a ST runs, its data must be set to be readable and wirtable in unprivileged mode

for privileged mode (P) and nonexecutable permission for unprivileged mode (U).

2.1.2 Exception Handling Process. LIPS takes advantage of the exception handling mechanism to store the register information and triger the role-identification process in privileged mode, so that it's necessary to understand the mechanism in Cortex-M processors. If a task forces the access to the unauthorized memory, Cortex-M processors will raise a memory access fault. And all registers will be forzen at once and automatically pushed to the stack (PSP) by the processor, the frozen registers include: (1) r0-r3, storing function parameters. (2) r12 (sp), storing the stack address of the task. (3) r13 (lr), storing the return address. (4) r14 (pc), storing the address of next instruction [30]. After stack pointer changes from Process Stack Pointer (PSP) to Main Stack Pointer (MSP), the system will go into privileged mode and cause the fault handler to execute in privilged mode [39].

2.2 Exokernel based Mbed Design

To illustrate why a malicious RT has to compromise the privileged control flow to get the sandboxed data of a ST, it's necessary to give the background on how Mbed achieves task switching under its sandboxing design.

2.2.1 Mbed Task Sandboxing. ARM Mbed adopts an Exokernel [21] based RTOS design, which makes tasks directely call RTOS_Lib for flexible system behavior in unprivileged mode, leaving uVisor to build task sandboxes in privileged mode. In Mbed, all of the codes are stored in Flash and are unwritable. The data are stored in RAM and their accessibility depends on permissions. With higher privilege level than tasks, uVisor can read and write unprivileged data and privileged data with special access. And all tasks cannot directly access the data of uVisor under any circumstance. uVisor rearranges the RAM layout for the data of STs and RTs, and defines separate MPU regions in RAM to store the data. The one for RTs and RTOS_Lib is in "AUM RAM", it enables access to these data in unprivileged mode. The other region for STs is in "OAPM RAM", which stores the data of STs and is only accessible in privileged mode. The number of STs is determined before Mbed starting up, so OAPM RAM is fixedly located in low RAM address.

2.2.2 Mbed Task Switching. As Fig. 1 (left) shows, the data of STs are originally stored in OAPM RAM. When ST1 is running, uVisor assigns another two MPU regions (Regions X and Y) to point to its stack and context (e.g. heap) section separately, and make these two regions readable and wirtable in unprivileged mode. So that ST1 can run in unprivileged mode. Before ST2 (right) starts to run, uVsior reconfigures MPU to make Regions X and Y point to the stack and context section of ST2. So that the unpointed stack and context section of ST1 will be located in OAPM RAM again. After that process ST2 will take control, but it cannot access the data of ST1. To sum up, the data of not running STs are unwritable and stored in OAPM RAM, while the data of running ST and RT are writable and in AUM RAM.

3 ATTACKS AGAINST MBED TASK SANDBOXING

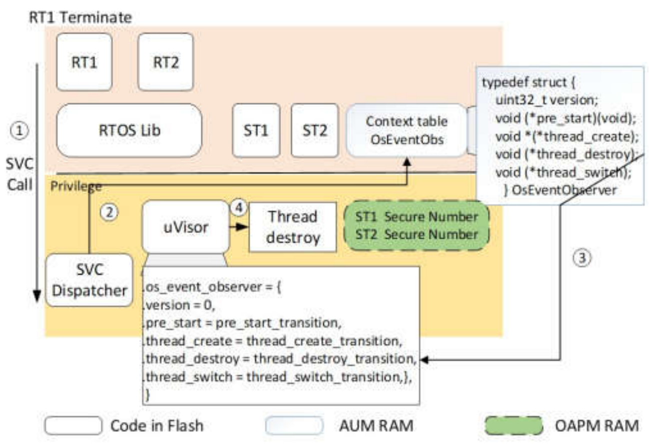
3.1 Context Table Remapping Attack

3.1.1 **Role of Context Table .** To locate particular memory operation in uVisor, Mbed needs to search *context table*. Context table, which contains a set of address pointers pointing to particular memory function codes in uVisor, is instantiated as a structure (named OsEventObserver) and pointed by a structure pointer (named osEventObs) in R-TOS_Lib [36]. For example, one of the address pointers is called void(*thread_destroy)(void), which links to a static fuction called thread_destroy_transition in uVisor [35]. The vulnerability of context table is known to some reseachers [36], but specific attacks exploiting this vulnerability is not reported in the literiture.

3.1.2 **RT Terminating in Mbed .** The task memory management is one of the most important mechanisms in RTOS design. uVisor is responsible for creating (create_transition), deleting (destroy_transition) and switching (switch_transition) the task stacks. In this section we only talk about the thread_destroy. To better understand CTR attack it is essential to understand the RT terminating first. Fig. 2(a) shows steps of the system flow when Mbed deletes a RT. *(1)* RT2 makes a task terminating SVC request to delete RT1, which will cause a software interrupt. *(2)* SVC dispatcher identifies the type of SVC instructions and *(3)* looks up the osEventObs to call thread_destroy. *(4)* Thread_destroy_transition will delete the data of RT1 in uVisor.

3.1.3 **Attack Details.** uVisor protects the data (including stack and context) of STs in OAPM RAM, and forbids other tasks to access them. Therefore a ST can store the secure number like encryption key in its sandbox. But a malicious RT can get and modify the secure number in following steps.

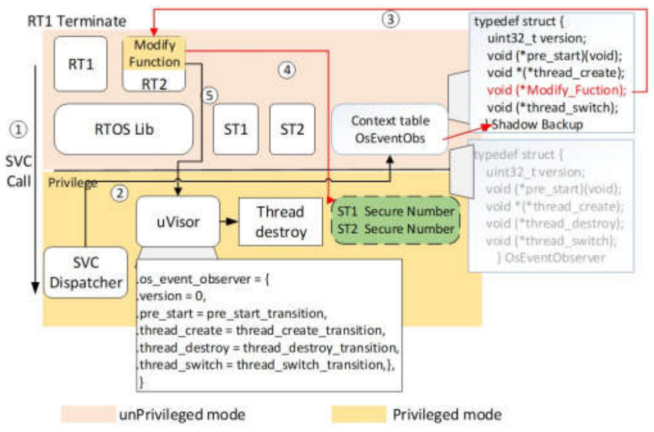
Step 1: Create a Malicious Task. We assume an attacker can perform the reverse analysis of the program binary by



```
typedef struct {
  uint32_t version;
  void (*pre_start)(void);
  void *(*thread_create);
  void (*thread_destroy);
  void (*thread_switch);
  } OsEventObserver
```

```
os_event_observer = {
.version = 0,
.pre_start = pre_start_transition,
.thread_create = thread_create_transition,
.thread_destroy = thread_destroy_transition,
.thread_switch = thread_switch_transition,},
}
```

(a) RT Terminating Process in uVisor



```
typedef struct {
  uint32_t version;
  void (*pre_start)(void);
  void *(*thread_create);
  void (*Modify_Fuction);
  void (*thread_switch);
  Shadow Backup
```

```
typedef struct {
  uint32_t version;
  void (*pre_start)(void);
  void *(*thread_create);
  void (*thread_destroy);
  void (*thread_switch);
  } OsEventObserver
```

```
os_event_observer = {
.version = 0,
.pre_start = pre_start_transition,
.thread_create = thread_create_transition,
.thread_destroy = thread_destroy_transition,
.thread_switch = thread_switch_transition,},
}
```

(b) Steps to Get the Secure Number in a ST through CTR

Figure 2: RT Terminating Behavior and Steps of Context Table Remapping Attack

IDA[2], thus he knows the exact address of the secure number. Then he creates a "Modify Function(MF)" in the RT2 to modify the secure number, which resides in OAPM RAM and beyond the access of MF at present.

Step 2: Copy the Context Table. Mbed uses the keyword "const" to define the function pointers stored in OsEventObserver, making which unrewritable, so RT2's direct modification to the pointers will cause a bus fault exception. However, RT2 has the permission to copy the contents (i.e., pointers) of OsEventObserver and puts them in AUM RAM as a replica, named "Shadow Backup(SB)"

Step 3: Replace the Pointer with MF. RT2 replaces the pointer of void(*thread_destroy)(void) with MF in SB. To guarantee the normal operation of uVisor after control flow hijacking, RT2 must also keep the pointer of void(*thread_destroy)(void) and recall it at the end of MF.

Step 4: Hijack Control Flow and Modify Secure Number. Finally RT2 makes OsEventObs point to the SB, which is equivalent to replace the whole contents in OsEventObs but not to directly modify the original pointers. With the control flow of task terminating rewritten, every time the SVC dispatcher wants to seach "void(*thread_destroy)(void)" for task terminating, it will call MF first, and MF will gain the access to the secure number in privileged mode.

After the CTR, the task terminating flow will be reconfigured as Fig. 2(b): *(1) and (2)* are the same as usual. *(3)* The SVC dispatcher looks up the OsEventObs, whose contents have been replaced by that of SB. *(4)* The MF will be called and run in privileged mode, which can either read or write the secure number willfully. *(5)* MF then gives control back to uVisor to fininsh task terminating. In the end, if there is a task waiting to read the secure number, it can get a modified result. In fact, we can hijack not only the control flow in task terminating, but also task creating and task switching, as they are all implemented through context table.

[2]https://www.hex-rays.com/

3.2 Function Code Reuse Attack

3.2.1 **Proposition of FCR.** Besides the context table, other vital data in RTOS_Lib are vulnerable to RT's malicious modification, and can be used to compromise the control flow. For example, changing the global variables (e.g. mbedtls_md5_context) in a cryptographic function (e.g. md5.h) may influence the computation result[28]. To explore the implementation of all these attacks, a major prerequisite is that these vital global variables (e.g., context table) are stored in AUM RAM. So that a theoretical defense scheme is proposed to solve this issue: to put these global variables into OAPM RAM, and bring the variables back in AUM RAM for their assignment and initialization procedure. However this scheme needs to create another two SVC functions to frequently change the access permission of these variables. These functions can be used by FCR to counter this solution, and we will take the context table as an example to illustrate its vulnerability.

Vulnerability of the Solution. In Mbed, uVisor is distributed as a prelinked binary blob [32] and physically separated from RTOS_Libs. To use the memory functions provided by uVisor, context table must be initialized to point to these functions before using. As initializing this table depends on particular codes (i.e. rt_OsEventObserver.c) in RTOS_Libs, the table in OAPM RAM has to be put back to AUM RAM during initialization. Therefore additional SVC functions must be created to flexibly change the access permission of context table, including (1) Deprivi, which defines the MPU region to store the table in AUM RAM; (2) Upprivi, which redefines the region to place the table back to OAPM RAM.

Though the newly created SVC functions can be used to dynaically change the access permissions of context table, they are at the risk of "Function Code Reuse (FCR)". Next we will introduce a simple and useful way to exploit the buffer overflow vulnerablity to recall these functions.

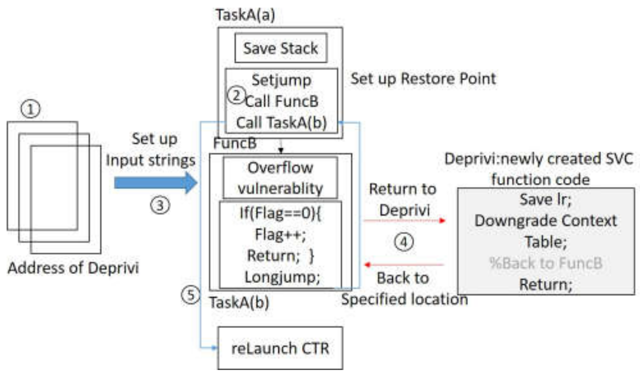


Figure 3: Steps of Function Code Reuse Attack

3.2.2 **Attack Details.** The goal of our attack is to re-expose context table and continue CTR. In this section we introduce FCR to make RTs reuse Deprivi. The details are shown in Fig. 3:

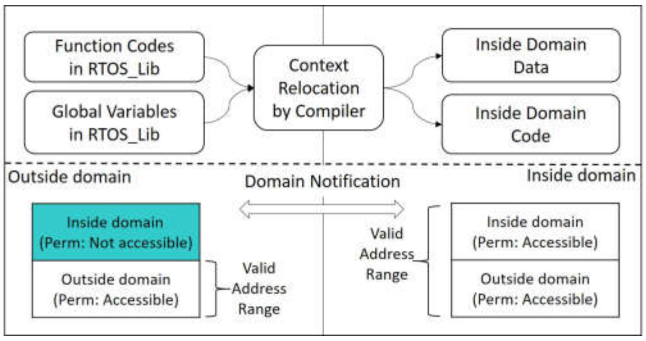
Step 1: Determine the Address of Deprivi. The attacker gets Entry Address of Deprivi first. We assume it can be easily got through a static analysis in the program.

Step 2: Create a FCR Environment. The attacker creates TaskA containing FuncB where its buffer overflow vulnerability makes sure the achievement of FCR in RTs. After Deprivi finishes, the original return address of FuncB is covered so that it cannot return to TaskA. Then the attacker sets a "Restore Point" for its return. Thus TaskA will be divided into two parts: TaskA (a) is responsible for saving the execute stack before FuncB executes and TaskA (b) is used to modify secure number via CTR attack.

Step 3: Set up Appropriate Input Strings for FuncB. The attacker makes a static analysis for the stack of FuncB and structures strings to overlap the return address of FuncB. Then he replaces the return address of FuncB with the Entry Address of Deprivi. Therefore Deprivi will be called by loading its Entry Address to pc register after FuncB returns. The address loaded to pc register must be modified according to instruction set modes.

Step 4: Return from Deprivi to TaskA. To avoid the program out of control, the attacker needs to appoint a designated loaction for Deprivi returning, which includes (1) Phase 1: returning to FuncB after Deprivi. Since the lr of a caller function will soon be saved by callee in its stack, the attacker can use a judgment sentence (i.e. if...else) before FuncB returns to Deprivi. So the lr will be written with the address of if in FuncB. (2) Phase 2: returning from FuncB to TaskA. Normally FuncB cannot return to TaskA because its original stack (i.e., return address) has been destroyed. To solve this problem, the attacker adds a setjmp [49] to make a Restore Point in TaskA(a), and a (longjmp) in FuncB to return to the Restore Point.

Step 5: Continue the CTR. The context table is downgraded to AUM RAM, TaskA (b) will repeat CTR attack to modify the secure number.



Function Codes in RTOS_Lib
Global Variables in RTOS_Lib
Context Relocation by Compiler
Inside Domain Data
Inside Domain Code
Outside domain
Domain Notification
Inside domain
Inside domain (Perm: Not accessible)
Outside domain (Perm: Accessible)
Valid Address Range
Valid Address Range
Inside domain (Perm: Accessible)
Outside domain (Perm: Accessible)

Figure 4: LIPS overview

4 LIPS DESIGN

4.1 Overview of LIPS

Fig. 4 describes our technique to implement the intra-mode privilege isolation. The two domains are isolated with different access permissions. The Outside domain obtains more restricted access to the memory, especially to Inside domain. Inside domain has less restricted access and can reach both Outside and Inside memory. This domain design is closely related to the Domain Notification Calls, which are designed only for STs to dynamically alter the permission of Inside domain. This notification mechanism is introduced to enable the exposure or concealment of memory for different domains.

4.2 Domain Isolation

Fig. 5 illustrates how we segment "the memory accessible in unprivileged mode" to achieve domain isolation, and this will not affect "the memory only accessible in privileged mode". The middle part is the original Mbed memory layout and the left and right parts show the memory perspective of the two domains.

Context Relocation. To build Inside domain, LIPS focuses on two basic elements in RTOS_Lib, which are the vital global variables and function codes (which take the form of a function). Crucial global variables are responsible for information transmitting among different tasks and Libs, and context table is a representative one which plays an important role in Mbed. Exokernel ensures each RT to directly call Lib functions in RTOS_Lib, however, crucial Libs (e.g. cryptographic methods) also need to be protected. For the remainder of this paper, the function codes and global variables placed in Inside domain will be called the Inside Domain Code and the Inside Domain Data separately. What's more, to solve the problem of MPU region limitation codes and global variables should be placed in separate continuous physical addresses, which can be easily achieved by the compiler [11]. Therefore we create new memory sections by marking Inside Domain Data and Inside Domain Code separately (see section 5.1 for Inside Domain).

4.2.1 Inside Domain Memory . Fig. 5 (left) describes memory layout of Inside domain. Inside Domain Code can call all the unprivileged code and access both Inside Domain Data and Outside Domain Data whose permissions are set to (U-R/W). LIPS also designs a flash region called the Notification

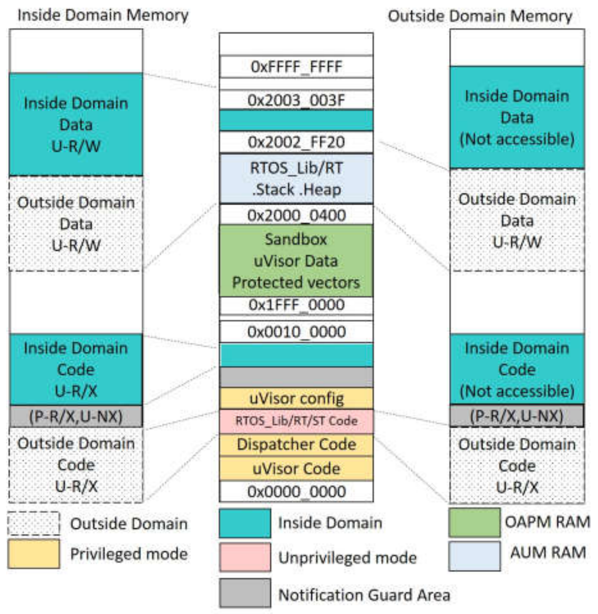


Figure 5: The physical address layouts for domains

Guard area (dark grey) set to (P-R/X,U-NX), where codes are nonexecutable in unprivileged mode. This area is used for the redirection to Reference Monitor, therefore codes in either Inside or Outside domains cannot modify it.

4.2.2 Outside Domain Memory . Fig. 5 (right) describes memory layout of Outside domain. Similar to Inside Domain Data and Inside Domain Code, the data and code in Outside domain are called Outside Domain Data and Outside Domain Code. Outside Domain Code, including RTOS_Lib, RTs and STs, can only modify Outside Domain Data in unprivileged mode. Outside Domain Data, which contain all data of RTs and RTOS_Lib, are set to (U-R/W) and range from 0x20004000-0x2002ff20. Inside domain is inaccessible (codes are nonexecutable and data are unreadable and unwritable) by Outside domain.

4.3 Domain Notification Calls (DNCs)

Despite that DNCs are designed only for STs to perform domain switching for safety reason, with RTs and STs all running in unprivileged mode, DNCs can also be used by RTs (which is potentially untrusted). For this problem, in DNCs we design two safety components to fix it, as showed in Fig. 6. One is DNC Entries, which are stored in Notification Guard area. Their functions includes guaranteeing themselves as the entrance of Domain Notification Calls, and causing Cortex-M processors to froze registers when a task calls them. The other one is Reference Monitor, which runs in privileged mode and performs "Frozen-register-based" task role identification-either RT or ST. It also keeps the access control rule that only STs can perform domain switching.

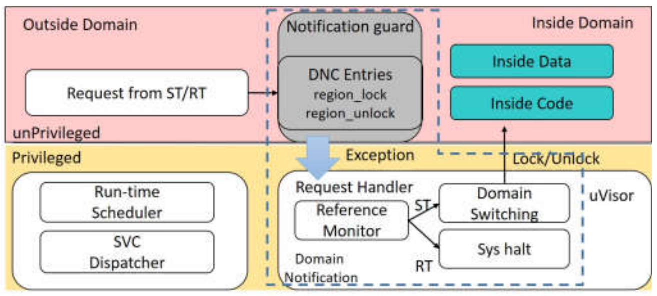


Figure 6: Domain Notification Call Architecture

4.3.1 Notification Guard Area and DNC Entries. DNCs take full advantage of the memory fault exception handling mechanism in Cortex-M processors (see section 2.1.2 for Exception Handling Process), and use MPU to set a NUM Flash area (which is nonexecutable in unprivileged mode) named Notification Guard area. And then, in Notifcation Guard area, DNCs create several void functions named DNC Entries, which correspond with different DNCs (to gain the permission to lock/unlock the Inside domain). Now, if either a RT or a ST want to call DNCs by accessing these entries, a memory access fault will arise. The registers of this task will soon be frozen, including the pc register which stores the value of DNC entry and sp register which stores the task's stack address. Then the control flow will be redirected to privileged Request Handler. In Cortex-M processors, the value of runtime pc register can not be forged, so that Request Handler can check the frozen pc register to make sure the entries are not bypassed. To be specific, Request Handler will not pass control to Reference Monitor until pc register points to one of the DNC Entries.

4.3.2 Frozen-register-based Access Control in Reference Monitor. LIPS further makes use of frozen sp register to realize task role identification. Before the system starts, uVisor provides two appointed RAM areas for STs and RTs respectively, which are fixed and separate from each other(see section 2.2 for AUM RAM and OAPM RAM). Based on this fact, Reference Monitor in Request Handler can identify the role of a task by the following rule: when a task accesses the DNC entries in notification guard area, the task's sp will be instantly saved (frozen). As the tasks sp must point to either of the two designated RAM areas, if the value of tasks sp register points to the RAM area which belongs to STs, it must be a ST, otherwise a RT. As Fig. 7 shows, with the help of the Notification Guard area, Reference Monitor can load the task registers and check the value of frozen sp. According to the result, the Monitor can identify the role of the task and determine to call sys halt or domain switching function.

This frozen-register based task-role access control method suits both of the situation when the task directly and indirectly (through FCR) calls DNC entries, as the task's sp register always points among RAM areas of tasks.

4.3.3 Domain Switching. When a ST calls the DNC, its input parameters will be stored in frozen registers (i.e., r0-r3) by Cortex-M processors, and obtained by the Reference Monitor

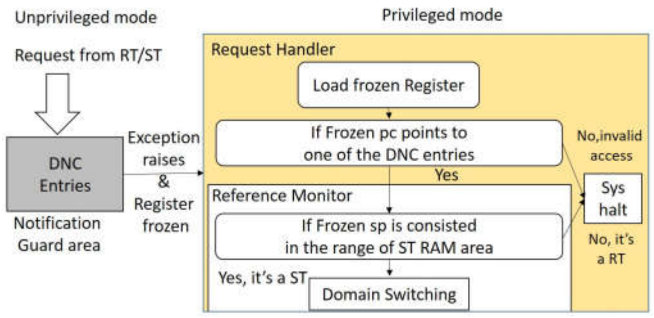


Figure 7: The Task Role identification

to control the switching bewteen Inside and Outside domains. According to the varies input parameters, the access permission of Inside domain can be dynamically changed. Table 1 illustrates the input opcode for each DNC entry. Currently we provide region_lock and region_unlock to change the permission of Inside Domain Code and data respectively. DNC Entries support 3 types of Region Code: 0x1 (for Inside Domain Code), 0x2 (for Inside Domain Data) and 0x3 (for Inside Domain Code and Data simultaneously). When a ST wants to access Inside Domain Data, it must call region_unlock to unlock domain protection. After that it can call region_lock to lock the domian.

Another way to lock/unlock the Inside domain is to create new lib functions which use SVC instructions to enter privileged mode, further to control MPU and finally to operate the Inside domain. However, this newly created lib functions are stored in RTOS_Lib so they can be executed by both RTs and STs, which means that the malicious RT can also use these functions to break down the domain protection through FCR attack or directly call them. (see section 3.2 for Function Code Reuse Attack).

5 LIPS IMPLEMENTATION

LIPS is incorporated into ARM Mbed uVisor, with Mbed5.4 and uVisor1.0. Most of our code is written in C. We also make some modifications to the compiler link script. The prototype is implemented on FRDM-K64F evaluation board, which is based on a Cortex-M4 core with 1024KB Flash, 256KB SRAM and a 120Mhz clock.

5.1 Inside and Outside Domains

Due of the MPU region limitation, either Inside Domain Code or Data need be recompiled in a contiguous range of address. We adopt the original uVisor memory design as a template. As Table 2 shows, regions from 0 to 4 are used as the uVisor defined protection regions, following the uVisor pre-defined memory permissions [38]. LIPS uses 3 additional regions for Inside domain. Region 9 is used as Notification Guard area, whose permission is set to (P-R/X, U-NX) and Regions 10 and 11 are used to store Inside Domain Code and data. Despite the settings of these regions (9 to 11), the template can still be changed to meet the system requirements. For example, if we want to arrange more Vital data and Vital code to Inside domain, we can enlarge regions 11 and 10.

Table 1: DNC Entries and the input parameters

Region Code	Opcode	Meaning	Entry Name
0x1	0x2	Data/User: R	region_lock
0x1	0x3	Data/User: R/W	region_unlock
0x1	0x4	Data/User: Not accessible	region_lock
0x2	0x11	Code/User: Not accessible	region_lock
0x2	0x12	Code/User: R	region_lock
0x2	0x13	Code/User: R/X	region_unlock
0x3	0x21	Code/User: R/X; Data/User: R/W	region_unlock
0x3	0x22	Code/User: R; Data/User: R	region_lock
0x3	0x23	Code,Data/User: Not accessible	region_lock

Meanwhile, the size of the compiled binary is highly related to the size of Inside Domain Code, because Inside Domain Code will occupy additional Flash area, whose unused part will be filled with 0xff and cannot be omitted. So region 10 must not contain too much unused area for the reduction of binary size. Regions 5 to 8 are left for later use of the system.

To build up regions 9 to 11, we modify the linking script to add three new sections: 1) "protected_code" section, to mark Inside Domain Code; 2) "protected_data" section, marking Inside Domain Data and 3) "Notification_guard" section, to mark Notification Guard area. Some init codes are added to uVisor initialization process to use the new sections for template regions establishment [38].

5.2 Domain Notification Calls

To accomplish the Domain Notification Calls, we implement DNC Entries and Request Handler. DNC Entries (i.e., region_lock and region_unlock), the entrance to DCN and initialized as void functions, are gathered through the link script and stored in the Notification Guard area, and all the access of tasks to these entries will cause a memory access fault. The registers running in unprivileged mode, including pc, lr, sp and r0-r3, will be saved. Then the Request Handler will check these registers in privileged mode to identify the role of the calling task.

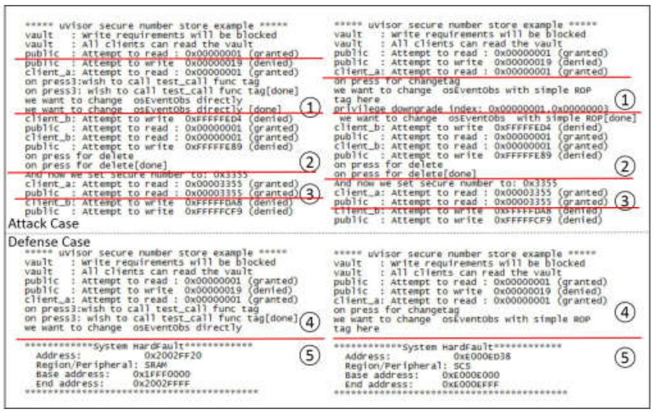
If the task passes the role identification, it is thought to be a ST and Domain Switching will happen. The input parameters of DNC will be restored from frozen r0 and r1. When the Request Handler finishies its work, it will return to DNC Entries which are pointed by the pc register of the calling task. Since the entries are in Notification Guard area, the access will raise a memory access fault again and lead the system to infinite calling loops. To handle this problem, Request Handler replaces the value of frozen pc with frozen lr to skip repeatedly calling DNC Entries. The details of Request Handler are shown in appendix C.

6 EVALUATION

In this section, we evaluate LIPS in five perspectives, the effectiveness of LIPS, the impact on task scheduling, performance overhead, the portability across different platforms and the security analysis. We first implement the proof-of-concept of both CRT and FCR, and then LIPS, to prove that LIPS can defend both CRT and FCR in real-world IoT applications. Next we set up the experiment to verify original task scheduling is rarely affected by LIPS. Then we measure

Table 2: LIPS region template.

Region Num	Permissions	Stat Addr	Size	Protects
0	P-R/W	0x0000_0000	4GB	Background Region
1	P-R/X, U-R/X	0x0000_0000	Code size	Code Section
2	P-R/W, U-R/W	0x2000_0400	Outside RAM size	Outside RAM
3	P-R/W, U-R/W	0x1FFF_0000+(Box N Start offset)	Box N Data Size	Box N Stack
4	P-R/W, U-R/W	0x1FFF_0000+(Box N Start offset)+(Box N Stack Size)	Box N Context Size	Box N Context
9	P-R/X, U-NX	0x0002_7000	1KB	Notification Guard
10	P-R/X, U-Varies	0x0002_7420	4KB	Inside Domain Code
11	P-R/W, U-Varies	0x2002_F020	4KB	Inside Domain Data



```
***** uvisor secure number store example *****
vault    : write requirements will be blocked
vault    : All clients can read the vault
public   : Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0x00000019 (denied)
client_a: Attempt to read : 0x00000001 (granted)
on press3:wish to call test_call func tag
on press3: wish to call test_call func tag[done]
we want to change  osEventObs directly
we want to change  osEventObs directly [done]
client_b: Attempt to write  0xFFFFFED4 (denied)
public   : Attempt to read : 0x00000001 (granted)
client_b: Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0xFFFFFE89 (denied)
on press for delete
on press for delete[done]
And now we set secure number to: 0x3355
client_a: Attempt to read : 0x00003355 (granted)
public   : Attempt to read : 0x00003355 (granted)
client_b: Attempt to write  0xFFFFFDA8 (denied)
public   : Attempt to write  0xFFFFFCF9 (denied)
```

```
***** uvisor secure number store example *****
vault    : write requirements will be blocked
vault    : All clients can read the vault
public   : Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0x00000019 (denied)
client_a: Attempt to read : 0x00000001 (granted)
on press for changetag
we want to change  osEventObs with simple ROP
tag here
privilege downgrade index: 0x00000001,0x00000003
 we want to change  osEventObs  with simple ROP[done]
client_b: Attempt to write  0xFFFFFED4 (denied)
public   : Attempt to read : 0x00000001 (granted)
client_b: Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0xFFFFFE89 (denied)
on press for delete
on press for delete[done]
And now we set secure number to: 0x3355
client_a: Attempt to read : 0x00003355 (granted)
public   : Attempt to read : 0x00003355 (granted)
client_b: Attempt to write  0xFFFFFDA8 (denied)
public   : Attempt to write  0xFFFFFCF9 (denied)
```

```
***** uvisor secure number store example *****
vault    : write requirements will be blocked
vault    : All clients can read the vault
public   : Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0x00000019 (denied)
client_a: Attempt to read : 0x00000001 (granted)
on press3:wish to call test_call func tag
on press3: wish to call test_call func tag[done]
we want to change  osEventObs directly
************System HardFault************
  Address:            0x2002FF20
  Region/Peripheral: SRAM
  Base address:     0x1FFF0000
  End address:      0x2002FFFF
****************************************
```

```
***** uvisor secure number store example *****
vault    : write requirements will be blocked
vault    : All clients can read the vault
public   : Attempt to read : 0x00000001 (granted)
public   : Attempt to write  0x00000019 (denied)
client_a: Attempt to read : 0x00000001 (granted)
on press for changetag
we want to change  osEventObs with simple ROP
tag here
************System HardFault************
  Address:            0xE000ED38
  Region/Peripheral: SCS
  Base address:     0xE000E000
  End address:      0xE000EFFF
****************************************
```

Figure 8: Effectiveness against CTR and FCR

the performance overhead on a real IoT device. Then the compatibility of LIPS is tested in different platforms of various versions of Mbed. At last we summarize the security guarantees provided by LIPS.

Several different kinds of binaries are evaluated in our test, including (1) unmodified baseline, (2) setting Inside/Outside domain, Domain Switching through SVC (theoretical design) and (3) setting Inside/Outside domain, with Domain Notification Calls (LIPS design). All binaries are obtained by default configuration of Mbed compiling command. Mbed system in our evaluation ranges from versions 5.4 to 5.9, which are fully supported by uVisor version stable (1.0). The main system clock in our test platform (i.e., NXP FRDM-K64F) is statically set to 120MHz.

6.1 Effectiveness Evaluation

The official example "secure number store"[3] is adopted to evaluate the effectiveness of LIPS, which is quite suitable for our test with the purpose to store sensitive data in a ST (called secure_number) and prevent it from unauthorized modification of other tasks (incluing other STs and RTs). Based on the example, we slightly change the program that "secure_number" blocks all the modification request from other STs (client_a and client_b) and RTs, only with reading permissions left. Next we will show how to use a untrusted RT to change the sensitive data in secure_number.

Fig. 8 illustrates our effectiveness experiment. The sensitive data is set 0x00000001 in "secure_number", and all RTs can't modify it. Attack Case (left) shows how CTR attack works in the following phases. *Phase 1:* an attacker can copy the contents of OsEventObserver to build the "DNC Backup (SB)". Then he replaces the pointer "void (*thread_destroy)(void *context)" with a malicious function in SB and makes OsEventObs point to the SB(see section 3.1 for Context Table Remapping Attack). *Phase 2:* a button interrupt is registered to trigger a task terminating request. After receiving the request, the SVC dispatcher actually calls the malicious function in SB. This function owns the privileged permission to modify the sensitive data. *Phase 3:* the target data has been successfully changed to 0x00003355. *Phase 4 and 5:* with the introduction of LIPS, performing CTR attack will raise a system hard fault. This is because RTs have no permission to copy OsEventObserver, which is already in Inside domain (0x2002FF20, inaccessible from Outside domain).

Fig. 8 Attack Case (right) describes how FCR attack can be used to call "Deprivi" for downgrading context table to "the RAM which is accessible in unprivileged mode". The details will show as the following phases: *Phase 1:* By carefully filling up the stack, the attacker can reuse Deprivi and Uprivi through a buffer overflow vulnerability in a bug RT (see section 3 for Function Code Reuse Attack). *Phase 2 and 3:* the table downgraded, the attacker can continue with CTR attack and maliciously modify sensitive data. *Phase 4 and 5:* The entries of Deprivi and Uprivi will be marked and recompiled into Notification Guard area. The access to the entries will be redireted to the Reference Monitor to make sure RTs can't perform domain switching.

In fact, the context table vulnerability is a system flaw in Mbed's Exokernel design and these applications won't affect any internal design in uVisor. Therefore all the features provided by LIPS will work well. This makes us believe that LIPS can serve most of the Mbed daily application scenarios.

6.2 Impact on Task Scheduling

Mbed adopts a Round-Robin scheduler, which employs time-sharing, giving each task a time slot. Once the time slot expires, the scheduler will force tasks out of the processor by storing the stack. To best test the impact of our technology on task scheduling, we will focus on the worst case.

Supposing the following situation: there are two tasks scheduled by the kernel, TaskA (ST) and TaskB (RT). However task scheduling happens before TaskA finishes accessing the resources in Inside domain, TaskA has no time to lock the Inside domain. When TaskB takes control, it can directly

[3]https://github.com/ARMmbed/mbed-os-example-uvisor-number-store/

Table 3: Impact on task scheduling.

Item	Task scheduling between Tasks (ms)	
	RT - RT	ST - RT
Without LIPS	1.495	4.44
With LIPS	1.514	4.46
Overhead	1.3% up	0.4% up

Table 4: Performance overhead on a sweeping robot.

Robot Stage	Normal (s)	Number of Access Inside domain (s)			
		0	1	2	3
WC	0.44	0.44	0.444	0.448	0.452
DC	10	10	10.004	10.008	10.012

Table 5: Time consumption to perform Domain Notification through eihter Reference Monitor or SVC.

Benchmark	BA (ms)	Direct SVC (ms)			DNC (ms)		
		SC	SD	CD	SC	SD	CD
ah..press	78.96	81.42	81.44	81.45	83.17	83.18	83.2
ah..ont64	93.71	95.17	96.19	96.2	97.93	97.93	97.95
Bu..sort	85.89	88.38	88.37	88.39	90.1	90.11	90.14
cnt	101.6	104.07	104.07	104.1	105.81	105.82	105.84
Compress	81.6	84.07	84.07	84.1	85.81	85.82	85.84
Cover	118.48	120.95	120.95	120.98	122.69	122.7	122.72
Ud	93.84	96.32	96.314	96.34	98.05	98.06	98.08
Tarai	81.44	83.91	83.91	83.94	85.65	85.66	85.68
Qsort	84.24	86.71	86.72	86.74	88.45	88.46	88.48
Lcdnum	84.4	86.87	86.88	86.9	88.62	88.62	88.64
DNC-Cost		2.38	2.48	2.49	4.21	4.22	4.24

access Inside domain. To handle this situation, we patch the scheduler so that it can help recognize TaskA and lock the Inside domain. However the locking process will cause some overhead and we will check if this overhead can be ignored.

Table 3 illustrates our experiments to measure the overhead on task scheduling. In Mbed two types of task scheduling exist, RT-RT and ST-RT. The locking process only happens in ST-RT, during which the permission of the sandbox memory needs to be reconfigured so that the Inside domain can be locked at the same time. Therefore the average extra latency is only 0.4%, which is within the time window of the deadline. Other than task scheduling in ST-RT, task scheduling in RT-RT does not need locking process so its overhead is only cost by extra task identification code in the scheduler, which is also tolerable.

6.3 Performance Evaluation

To evaluate the performance on overall system, we deploy LIPS on a carefully simulated sweeping robot. The typical working flow of the robot can be divided into two stages. In first stage the robot analyse the control command from the remote host, called Waiting for Command (WC). In the second stage the robot will work until it finishes cleaning, called Doing the Cleaning (DC). The total time of DC depends on the size of cleaning area, so we make the robot clean 0.5 square meters. To measure the overhead in both stages, we manually set the numbers of their access to Inside domain. The results are shown in Table 4, although the access number is set to 3, the final benchmark scores reach 0.91 % (during WC) and 0.4 % (during DC) performance overhead on average, shwoing the feasibility of the prototype. Due to the linear growth of the score, we can conclude that the time of the WC will be expressed in this form: Time (WC)= 0.44+0.004n (s), where n represents the number of requests for Inside domain during the execution of WC. Simultaneously DC can be expressed as Time (DC)= 10+0.004n (s).

We select 10 of BEEBs benchmarks [48] at random to measure the average execution time of Donmain Notification through Reference Monitor and SVC. Each benchmark performs a full domain access operations, including unlocking, accessing and locking the Inside domain. As Table 5 shows, columns are the time consumption of (1) original benchmarking (BA), (2) accessing Inside Domain Code (SC), (3) accessing Inside data (SD) and (4) accessing both Inside Domain Code and data (CD). We can draw conclusions that: (1) The time of domain switching to access Inside Domain and Code are basically the same. (2) The average overhead of domain switching will be 2470us by SVC, and 4210us by DNC. (3) Though taking shorter time, the SVC-based domain switching lacks the task role identification.

6.4 Portability Evaluation

Table 6 gives the compatibility evaluation on various platforms. STM32F429Discovery[16] and Giant Gecko[19] both support Mbed and uVisor. Despite only 8 MPU regions in these two boards, their design rules of MPUs endow the overlapped MPU regions with the permission of higher numbered one, different from those applied in our prototype which allow overlapping permissions. So we reuse some regions to realize the core techniques in LIPS to finish transplantation. Though LPC 1768[41], Nucleo-32[46] and Nucleo-64[37] are available for Mbed, our technique cannot run on these devices for their lack of MPU. The FreeRTOS-based Mico Evalution board[43], which does not support uVisor, is also not feasible. In fact, uVisor Guides [37] claims it can be ported to not only Mbed-enabled platforms but also CMSIS RTOSes by simply adding some platform specific codes. Since LIPS exerts the maximum effect in uVisor, the good portability of our technique is ensured.

6.5 Security Analysis

Throughout section 4, we discussed in detail how LIPS achieves domain isolation, Frozen-register-based access control and domain switching. In this section, we first summarize how these features fulfill the required security guarantees. Afterwards, we discuss how LIPS prevents other possible attack scenarios.

Security Guarantees: According Section 4, LIPS provides two principal security guarantees. First, it guarantees that malicious RTs cannot break the domain isolation. CTR attack is well protected by this property. Second, it guarantees the access control that only STs can execute Domain Notification Calls to access Inside Domain, and block the request from RTs. This property can defend the FCR attack.

Section 4.2 shows how LIPS uses the MPU to provide the domain isolation. The Inside Domain design can protect not only the context table, but also other vital global variables

Table 6: The portability of LIPS crossing different platforms.

NO.	Platform	Architecture	Mbed(ver 5.4-5.9)	uVisor(ver 1.0)	LIPS	remark
1	FRDM-K64F	Cortex-M4, 12 MPU regions	fit	fit	fit	soft/hardware support
2	STM32F429Discovery	Cortex-M4, 8 MPU regions	fit	fit	fit	soft/hardware support
3	Giant Gecko	Cortex-M3, 8 MPU regions	fit	fit	fit	soft/hardware support
4	LPC 1768	Cortex-M3, No MPU	fit	not fit	not fit	lack uVisor support
5	STM32L031 Nucleo-32	Cortex-M0, No MPU	fit	not fit	not fit	lack uVisor support
6	STM32L053 Nucleo-64	Cortex-M0, No MPU	fit	not fit	not fit	lack uVisor support
7	Mico Evalution board	Cortex-M3, 8 MPU regions	not fit	not fit	not fit	FreeRTOS

and function codes (e.g. decryption functions) in RTOS_Lib. The memory latout defined by LIPS prevents the Outside Domain from accessing the isolated Inside Domain. Moreover, as LIPS is embedded in uVisor and their data are unmodifiable, RTs cannot compromise uVisor to use privileged instructions to control the MPU to revoke this protection.

Section 4.3 shows how Domain Notification Calls take advantage of DNC entries, Notification Guard area and froze-task-register based access control. These mechanisms combined guarantee that the Inside Domain is accessible only when the ST takes control. They also guarantees that this only happens at a specific entry point (i.e., DNC entries). Hence, RTs cannot tamper with domain notification process to break the isolation.

Stack safety of Inside Domain Code: There is a security issue that if the run-time scheduler performs task switching when a task calls Inside Domain Code, its local stack will be left in Outside domain and face the danger of being compromised by other RTs. But this will not affect our Inside domain design as Inside domain is only for STs, and RTs can not access the Inside domain. If above issue happens to a ST, its stack will be sandboxed in OAPM RAM. So the stack security of STs is ensured.

Code Snippet Reuse: To reuse DNC, in addition to FCR, a possible way of attacks can be done by reusing some code snippet in a ST. As DNC is designed for ST, it will be in some ST's function code, the reusing of which can indirectly call DNC entries. Even so, RT's sp pointer is still in AUM RAM so that the judgement of Reference Monitor won't be changed. In fact, due to software and hardware restraints, in RTOS all the attacks repeatedly and sophisticatedly modify the stack and registers but soon will lose control that programs would go out and RTOS would crash. We will take the attack of "reusing some code snippet in a ST" as an example, to illustrate the unpredictable system influence on RTOS. The details are showed in Appendix A.

7 RELATED WORK

There have been lots of defending technique in desktop systems, such as Control-Flow Integrity (CFI)[1, 26, 44, 47, 55, 57] and protections of code pointers (CCFI,CPI)[24, 40]. Performance and precision are two key factors affecting the implementation of CFI in real system. Since all instructions need to be tracked, the overhead of the CFI will be pretty high. However a fine-grained CFI research illustrates that it can reduce the overhead[55]. But the security of the most precise CFI still remains contingent. ECFI[2] designs a novel and PLC-compatible CFI mechanism for real-time PLCs and considers the runtime operation of the PLC as the highest priority. LMP[20] leverages Intels Memory Protection Extensions (MPX) to make backwards-edge CFI both secure and efficient. Our techniques build protection domain for context table. Therefore the control flow in Exokernel cannot be hijacked by CTR. CPI[24] depends on the virtual memory in MMU, whose feature is not available on Cortex-M processors. We provide the Reference Monitor, checking the sp of tasks whose legal access to DNC will be assured.

Building protection domains in Cortex-M processors is a relatively new problem, as most of the current researches are focusing on the Cortex-A processors. To the best of our knowledge, the most notable work recently to realize this technique is Hilps[10], which utilizes TxSZ hardware field on AArch64 to achieve an intra-mode privilege isolation mechanism that memory in the same privilege will be divided into two separate domains. Microstache[42] achieves efficient intra-process isolation through a specialized hardware mechanism and new process abstraction on x86-64 platform. However these solutions require either special registers (TxSZ) or powerful peripherals (MMU), which are beyond Cortex-M processors for constrained resources. LIPS uses MPU to divide "the memory accessible in unprivileged mode" into separate domains, which only causes a small amount of overheads.

Focusing on embedded systems, some frameworks[7, 9, 11, 22, 23, 45] are proposed to build memory isolation. FreeRTOS[7] achieves coarse-graid isolation by separating tasks in different privileges. EPOXY [12] introduces the "privilege overlay" mechanism in which all the marked instructions are redirected to run in privileged mode. Therefore the unprivileged bare program cannot directly access the critical hardware resources. But the high frequency for instruction overlay will cause a huge slowdown. ACES [11] uses a LLVM-based compiler to automatically infer and enforce inter-component isolation on bare-metal systems. A thread will be cut into many small components so that it may affect the effectiveness of task scheduling. MINION [22] uses LLVM and dynamic analysis to infer thread-level compartments and uses the OS context switch to change compartments. However this technology puts all the RTOS in "the memory accessible in unprivileged mode", which is not fit for the Exokernel design. TrustLite[23], TyTan[9] and Sancus[45] provide effective isolation solution, but they all require specific modifications to the circuit of processors, which show poor portability. ARM

TrustZone[27] is available in all spectrums of Cortex-A and the coming Cortex-M processors which are based on ARMv8-M[31]. ARMv7-M based IoT devices are still widely adopted in industry while they do not contain this feature.

8 DISCUSSION

Vulnerability harmfulness. In the previous chapters we firstly exploit the vulnerability of context table to uncovered the vulnerabilities of ARM Mbed task sandboxing via uVisor by crafting two attacks-Context Table Remapping (CTR) attack and Function Code Reuse (FCR) attack. The developers or system maintainers may utilize these vulnerabilities to reserve some spy interfaces in RTs, through calling the interfaces they can break the sandbox when necessarily and store ST's data or change IoT devices' operating mode.

Compiler-based Domain Solutions. In some studies, domain establishment can be achieved by modifying the compiler. The llvm seems to be a good solution because it's convenient to develop new technique by adding additional compiling passes[11, 12, 22], which is not supported by uVisor for now. To use it one must rewrite all the core code including uVisor, Exokernel and RTOS_Libs so that the posibility of making error will be much higher. While our technique only requires slight modifications to Mbed and uVisor, which presents both practicality and flexibility.

9 CONCLUSION

After all these years, both the attack and defense mechanism targeting x86 systems have developed to be accomplished with more and more complicate and cumbersome steps, accompanied with the evolvements of OS and hardware[25][53]. In comparison, Cortex-M based RTOSes are not only lightweight but also low-level in hardware, which make it much more direct and easier for RTOSes to be attacked and also to defend.

In this paper, we takes ARM Mbed as an example to research the task sandboxing techniques in Exokernel-based RTOS, and specifically craft the new attack to illustrate the severity of the vulnerability in task management. The experimental results show that exploiting the vulnerabilities attackers can hijack the privileged control flow and break the sandbox protection.

To make up for the new vulnerabilities, we propose LIPS, which uses Inside/Outside domains and Domain Notificaion Calls to enfore the task sandboxing technique in Mbed uVisor. A new round of experimental results prove that LIPS not only can effectively defend the widely existing CTR and FCR, but also has small runtime overheads and good portability.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 4.

[2] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*. 437–448. https://doi.org/10.1145/3134600.3134618

[3] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 2–13. https://doi.org/10.1145/1168857.1168860

[4] alexeyk13. 2019. RExOS - Realtime Exokernel Operating System. https://github.com/alexeyk13/rexos.

[5] Jorge Alfonso, Nuria Sánchez, José Manuel Menéndez, and Emilio Cacheiro. 2015. Cooperative ITS communications architecture: the FOTsis project approach and beyond. *IET Intelligent Transport Systems* 9, 6 (2015), 591–598.

[6] H Arasteh, V Hosseinnezhad, V Loia, A Tommasetti, O Troisi, M Shafie-Khah, and P Siano. 2016. Iot-based smart cities: a survey. In *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*. IEEE, 1–6.

[7] Amazon Web Services (AWS). 2017. FreeRTOS MPU. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[8] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2010. Rootkits on smart phones: attacks, implications and opportunities. In *Eleventh Workshop on Mobile Computing Systems and Applications, HotMobile '10, Annapolis, Maryland, USA, February 22-23, 2010*. 49–54. https://doi.org/10.1145/1734583.1734596

[9] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[10] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. 2017. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM.. In *NDSS*.

[11] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic Compartments for Embedded Systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 65–82.

[12] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–303.

[13] CVE-2018-16603. 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16603, 2018.

[14] CVE-2018-9989. 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9989, 2018.

[15] Denning, Peter J. 1970. Virtual memory. *ACM Computing Surveys (CSUR)* 2, 3 (1970), 153–189.

[16] DISCO-F429ZI. 2017. The STM32F429 Discovery kit. https://os.mbed.com/platforms/ST-Discovery-F429ZI/.

[17] Dawson R Engler. 1998. *The Exokernel operating system architecture*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[18] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. 251–266. https://doi.org/10.1145/224056.224076

[19] EFM32 Giant Gecko. 2017. Silicon Labs' EFM32 Giant Gecko. https://os.mbed.com/platforms/EFM32-Giant-Gecko/.

[20] Wei Huang, Zhen Huang, Dhaval Miyani, and David Lie. 2016. LMP: light-weighted memory protection with hardware assistance. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. 460–470. http://dl.acm.org/citation.cfm?id=2991089