Commercial hypervisor-based task sandboxing mechanisms are unsecured? But we can fix it!

Dongdong Huo\textsuperscript{a,b}, Chen Cao\textsuperscript{c}, Peng Liu\textsuperscript{c}, Yazhe Wang\textsuperscript{a,b,*}, Mingxuan Li\textsuperscript{a,b}, Zhen Xu\textsuperscript{a,b}

\textsuperscript{a} Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
\textsuperscript{b} School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
\textsuperscript{c} The Pennsylvania State University, PA, USA

\section{Introduction}

Cyber–Physical–Social Systems (CPSS), which further integrate the social space with the Internet of Things (IoT), have emerged as the promising instrument in providing valuable information on personalized services \cite{1}. For better realizing these services, High Performance Computing and Communications (HPCC) technique, as a potential branch of CPSS, is innovating ways of processing large amounts of heterogeneous data \cite{2–4}. However, its security is not keeping up with the pace of such innovations \cite{5}. For instance, to avoid reducing the effectiveness, such technique also requires the veracity of these data to honestly describe the state of “things”, as discussed in recent literature \cite{6–8}. Though some explorations have noticed the importance of data security \cite{9}, they can just mitigate the threats from the perspective of cloud. Consequently, as one of the basic production facilities for generating data, IoT devices and their security issues are drawing great attention in industry and academia \cite{10–12}.

Nowadays, with the development of the IoT technique, Real-Time Operating Systems (RTOSs) are widely deployed to handle complex tasks. Most commercial RTOSs run on ARM Cortex-M processors, which have a low cost and low-energy consumption \cite{13}. However, these processors also have limited hardware-supported features, compared to x86 or Cortex-A processors. As an example, they do not have the typical Memory Management Unit (MMU), which supports virtual memory in the system. Thus, the basic isolation or sandboxing enhanced by virtual memory cannot be achieved. Generally, all tasks running on top of one RTOS share a flat memory space and they can access each other’s data without restriction. If one task gets compromised, the whole system is at risk. So, implementing task isolation or task sandboxing in RTOS on Cortex-M processors is required, but challenging.

Recent studies show an advance in implementing task sandboxing, which adopts a security extension, Memory Protection Unit (MPU), on Cortex-M processors. Although the MPU does not support virtual memory, it defines memory regions with different access permissions (read/write/execute) in privileged and unprivileged modes. Further, such permissions can be dynamically changed by MPU only in privileged mode. In consequence, by configuring whether a task can access certain memory regions when the task is executing, one RTOS can isolate tasks and provide task sandboxing.
In general, the state-of-the-art Cortex-M task sandboxing mechanisms can be classified into two classes: (Class A) privileged and unprivileged task sandboxing; (Class B) hypervisor-based task sandboxing. Class A mechanisms are enforced in FreeRTOS [14] and other similar monolithic RTOSs, which isolate tasks by making them run in separated privileged modes. Unlike Class A, the RTOSs such as Mbed, supporting hypervisor-based task sandboxing, puts all tasks in unprivileged mode, and the hypervisor is the only component in privileged mode, which is responsible for configuring the MPU. This particular design reduces the attack surfaces and enables the hypervisor to define different memory region maps for different tasks.

Although almost all existing works focus on Class A task-sandboxing [15–17], this paper researches Class B task-sandboxing, taking ARM Mbed uVisor [18] as an example of the RTOS with hypervisor. Class B task-sandboxing is also implemented by Mbed via the security partition manager (SPM) [19]. Though SPM is out of the scope of our proposed system prototype, its properties for ARMv7-M architecture have essentially no difference with uVisor. First, both uVisor and SPM rely on MPU and privileged execution for isolation. Second, they are both a self-contained privileged component which provides thread-level sandboxes (in SPM, a sandbox is denoted as a partition) for individual tasks by limiting access to memories. Third, they both make all tasks run in unprivileged mode. As a consequence, we prefer researching the open-sourced uVisor as SPM is currently close-sourced.

During the research, we found that Mbed uVisor, which supports hypervisor-based task sandboxing, is vulnerable, and the sandbox can be compromised by modifying some vital system-maintained data structures. For example, both OsEventObserver and osRtxInfo in Mbed can be accessed in unprivileged mode. By modifying certain fields in these data structures, a malicious task can access other tasks’ data and fail the task sandboxing.

In terms of how to defeat this attack, although the idea of “making it only accessible in privileged mode after initialization” seems reasonable, we found this idea would cause unacceptable denial-of-service, when applied to all the potentially-exploitable data structures. Taking osRtxInfo as an example, there is a period of time in which it must be accessible in unprivileged mode. For instance, the system would crash when a system task osRtxTimerThread running in unprivileged mode periodically searches MessageQueue in osRtxInfo. Hence, it is unlikely for Mbed uVisor to be defended from the attack discovered by us unless its task-sandboxing mechanism is improved.

To solve this problem, we propose to build a special protection domain to protect the system-maintained data structures. Its key security property is to enforce a strict access control rule to prevent accesses to these data structures in unprivileged mode. This proposal results in LIPS (Lightweight Intra-Mode Privilege Separation), which uses the MPU to divide memory space into the Internal domain and the External domain. Vital system-maintained data structures in the Internal domain are well protected by making it conditionally inaccessible in unprivileged mode. Meanwhile, all tasks’ data and code are placed in the External domain for normal operation. Additionally, we provide a secure switching mechanism which ensures that all tasks accessing the Internal domain must go through Domain Calls. Invoking a domain call raises an exception, and LIPS can identify the caller task based on certain register values when the exception is raised.

In summary, our work makes the following contributions:

• We discover the vulnerabilities (i.e., each vulnerability corresponds to a particular system-maintained data structure) associated with Mbed uVisor, which supports hypervisor-based task sandboxing mechanism. When being exploited, these vulnerabilities can enable a malicious task to access other tasks’ data and fail the task sandboxing. Hence, data stored in such IoT devices are unsecure and can be easily forged, which may drastically reduce their credibility.

• To mitigate the attack, we propose LIPS, which provides lightweight intra-mode privilege separation by separating vital system-maintained data structures from all tasks in two isolated domains. Thus the security of data generated from such devices is ensured, with assistance on improving the data quality.

• With a series of experimental tests, we demonstrate the attacks and the protection enhanced by LIPS in Mbed uVisor, proving its effectiveness in expanding our understanding of safeguarding a typical type of hypervisor-based RTOSs.

The rest of this paper is organized as follows. Section 2 narrates the background. Section 3 presents motivational threat model. Section 4 explains the design of LIPS in detail. Implementation and evaluation of LIPS are described in Section 5 and Section 6, respectively. Section 7 discusses the related work. Section 8 concludes this paper.

2. Background information

In this section, the relevant background information such as the task sandboxing design, the ARM Cortex-M processor and Mbed OS are reviewed.

2.1. Task sandboxing design

Task Sandboxing between x86 and Cortex-M Processors. In order to emphasize distinctions in security capabilities, features of task sandboxing between two platforms are compared in this section. As Cortex-M processors lack MMU to provide virtual memory, the detailed implementations of their task sandboxing are different from those implemented in x86 or Cortex-A processors. For example, first, in x86 processors, a sandboxed process cannot directly execute other sandboxed processes’ codes, while in Cortex-M processors the sandboxed task can execute codes of other sandboxed tasks running in the same privileged mode. Second, in x86 processors, a sandboxed process cannot directly read/write other processes’ data. Nevertheless, such accessibility is not stationary in Cortex-M processors as the permission of data can be dynamically changed by MPU. Third, x86 processors support two-way sandbox isolation by isolating every sandboxed process running in the same privileged mode. Second, in x86 processors, sandboxed processes cannot access other processes’ memory spaces. However, without MMU, Cortex-M processors cannot achieve such isolation. For instance, tasks can index each other as they share a single memory space. These differences lead us to believe that the task sandboxing mechanism in Cortex-M processors is unique, because its running environment limits the deployment of solutions available in x86 or Cortex-A processors.

FreeRTOS task sandboxing. Besides uVisor, one of the most popular type of task sandboxing mechanisms is implemented by FreeRTOS and its variants. To be specific, FreeRTOS leverages MPU to design two kinds of memory region maps, which are for all privileged components and for unprivileged components separately. According to these maps, tasks are divided into unprivileged tasks and privileged tasks. Among them, both privileged tasks and lib codes of FreeRTOS run in privileged mode and their data (i.e., stack and context) are stored in particular RAM region which is set to be only accessible in privileged mode. While unprivileged tasks run in unprivileged mode with their data stored in RAM region which is accessible in unprivileged mode. Based on this design, FreeRTOS ensures that unprivileged tasks cannot directly access privileged tasks [14]. However, enormous privileged components may leave unpredictable attack surfaces for attackers, as shown by recent literature where unprivileged tasks can exploit the buffer overflow vulnerability to compromise the privileged OS service to manipulate the kernel [20].
normal tasks refer to tasks developed by the third-party developers, while system tasks refer to tasks implemented by the system originally, while sandboxes for different tasks. Fig. 1 shows the whole process of task memory protection. MPU divides the single memory space into several regions. Then this task can only access these regions, excluding the MPU configuration, and then configures MPU to allow its access to certain regions. Therefore, in order to do memory protection, developers have to create a delicate architecture, which can leverage MPU to support different peripherals into this single memory space. That is, a malicious task can also access different peripherals without restraint.

To counter with the missing of MMU, Cortex-M processors introduce Memory Protection Unit (MPU), which can only be configured in privileged mode. MPU can divide this single memory space into several regions and assign different access permissions for them. When one task accesses a region which it does not have related access permissions, an exception is raised. However, MPU is a lightweight memory protection component, which provides access control for only certain regions. For example, the MPU in NXP FRDM-K64F development board provides up to 12 regions. Therefore, in order to do memory protection, developers have to create a delicate architecture, which can leverage MPU to reduce the potential threat. Otherwise, vulnerabilities can be easily exploited to attack the system.

**Mbed OS and uVisor.** Mbed OS [21] is an embedded operating system designed by ARM for Cortex-M processors and targets Internet-of-Things (IoT) devices. It is free and open-source, which draws attention from manufacturers and has been deployed in plenty of IoT devices. The popularity is usually accompanied by the risk of attacks. Hence, security is considered and it is one of the important features in Mbed OS.

Mbed OS security is supported by uVisor, which offers data isolation among tasks. It always executes in the privileged mode, while all tasks are in unprivileged mode. This means that only uVisor can configure MPU. Furthermore, uVisor itself can access all data in the memory space by default. Essentially, uVisor is a supervisory kernel, configuring MPU for different data and peripherals, which can be accessed in unprivileged mode by a certain task. Each task in Mbed OS has a sandbox ID, which is associated with one MPU configuration. When a task is scheduled to run, uVisor looks up the related sandbox ID, retrieves the MPU configuration, and then configures MPU to allow its access to certain regions. Then this task can only access these regions, excluding other tasks' data or certain peripherals. In this way, uVisor provides sandboxes for different tasks. Fig. 1 shows the whole process of task switching with related data structures in Mbed uVisor. In Mbed uVisor, system tasks refer to tasks implemented by the system originally, while normal tasks refer to tasks developed by the third-party developers.

### 2.2. Cortex-M processor and Mbed OS

**ARM Cortex-M Processor.** To explore RTOS task sandboxing mechanisms, ARM Cortex-M processor family is a suitable subject as it is dominating the microcontroller market, due to the low cost and low power consumption [13]. It is based on ARM M-profile architecture, which is different from the classical ARM A-profile architecture in several aspects. For example, it only supports Thumb instructions for code density. Except that, the most significant difference is Cortex-M processors do not have Memory Management Unit (MMU). MMU supports virtual memory for tasks in a system, which can be utilized for isolation among tasks. Without MMU, different tasks share one flat memory space, which results in the risk of manipulating other tasks' data by a malicious task. Even worse, Cortex-M processors also map different peripherals into this single memory space. That is, a malicious task can also access different peripherals without restraint.

In Mbed OS, events are related with a set of callback function pointers, which are invoked when a certain event happens. For example, the data structure OsEventObserver stores function pointers related to task creation and switching, etc. When switching tasks, one of the function pointers called thread_switch is invoked and it essentially points to a static function implemented in uVisor, called thread_switch_transition. This data structure OsEventObserver is managed by system tasks and it is pointed by a specific pointer, osEventObjs. However, this pointer is not protected from normal tasks and can be manipulated. The process of exploiting this vulnerability is similar with system call table hijacking in the Linux kernel. First of all, a malicious task copies all contents of OsEventObserver to a new memory space to build a Shadow, and manipulate osEventObjs to point to this Shadow. The Shadow is proposed to deal with the situation where Mbed uses the keyword “const” to define the function pointers stored in OsEventObserver, making them unrewritable after its initialization, so normal task's direct modification to the pointers will cause a memory fault exception. Nevertheless, normal tasks has the permission to copy the contents (i.e., pointers) of OsEventObserver as a replica (i.e., Shadow), which will avoid such exception. In this Shadow, certain callback function pointers can be replaced by malicious function pointers. As an example, thread_switch can be replaced to point to an attacker-controlled function. Since all function pointers in OsEventObserver are invoked by SysTick Handler, which has not called EXC_RETURN and runs in privileged mode, they can configure MPU, access any data and peripherals mapped in the memory. That is, the attacker-controlled function breaks the isolation among tasks and fails uVisor, as shown in Fig. 2.

### 3. Motivational threat model

In this section, vulnerabilities in Mbed task sandboxing, the threat model and assumptions are presented to describe the motivational threat model of this paper.

#### 3.1. Vulnerabilities in Mbed uVisor

The goal of uVisor hypervisor is to achieve isolation among tasks, i.e., one task cannot access other tasks' data or certain peripherals. However, after scrutinizing the source code of Mbed OS and uVisor, we discovered two vulnerabilities in Mbed uVisor, which can be exploited to break such isolation among tasks. In the next few sections, we will elaborate both vulnerabilities and how to exploit them.

##### 3.1.1. OsEventObserver

In Mbed OS, events are related with a set of callback function pointers, which are invoked when a certain event happens. For example, the data structure OsEventObserver stores function pointers related to task creation and switching, etc. When switching tasks, one of the function pointers called thread_switch is invoked and it essentially points to a static function implemented in uVisor, called thread_switch_transition. This data structure OsEventObserver is managed by system tasks and it is pointed by a specific pointer, osEventObjs. However, this pointer is not protected from normal tasks and can be manipulated. The process of exploiting this vulnerability is similar with system call table hijacking in the Linux kernel. First of all, a malicious task copies all contents of OsEventObserver to a new memory space to build a Shadow, and manipulate osEventObjs to point to this Shadow. The Shadow is proposed to deal with the situation where Mbed uses the keyword “const” to define the function pointers stored in OsEventObserver, making them unrewritable after its initialization, so normal task's direct modification to the pointers will cause a memory fault exception. Nevertheless, normal tasks has the permission to copy the contents (i.e., pointers) of OsEventObserver as a replica (i.e., Shadow), which will avoid such exception. In this Shadow, certain callback function pointers can be replaced by malicious function pointers. As an example, thread_switch can be replaced to point to an attacker-controlled function. Since all function pointers in OsEventObserver are invoked by SysTick Handler, which has not called EXC_RETURN and runs in privileged mode, they can configure MPU, access any data and peripherals mapped in the memory. That is, the attacker-controlled function breaks the isolation among tasks and fails uVisor, as shown in Fig. 2.
4. Design of LIPS

In this section, the design of LIPS (i.e., domain isolation and domain calls) is discussed, which demonstrates the protection provided by LIPS.

3.2. osRtxInfo

Another vulnerability is also related with one critical data structure, osRtxInfo. This data structure is responsible for storing OS runtime information, including message queue, timer, ready-to-run tasks, etc. In this data structure, one field, sandbox_id, which is used by uVisor to match the correct MPU configuration for a task, can be accessed by all tasks. Therefore, one malicious task can change its own sandbox_id to other tasks’ and deceive uVisor into configuring other tasks’ MPU configuration. In the end, with the related access permission, the malicious task can access other tasks’ data or peripherals.

The fundamental cause of both vulnerabilities is that certain critical data structures and related pointers in Mbed uVisor are exposed to normal tasks. So, normal tasks can manipulate them to break the isolation enhanced by Mbed uVisor. Although we only elaborate two vulnerabilities above, the exposure can be extended to other critical data structures and related pointers, which leads to more attacks.

3.2. Threat model and assumptions

LIPS emphasizes protecting the critical data structures and related pointers in Mbed OS. We assume the implementation of Mbed OS and uVisor is trusted, but certain critical data structures and related pointers are exposed and can be accessed by all tasks. System tasks are usually developed with Mbed OS and we also assume they are trusted. However, normal tasks are developed by third-party developers and may have vulnerabilities, such as buffer overflow, which can be exploited by attackers to gain control of related tasks. This assumption is aligned with the idea of Mbed uVisor, that the normal task is not trusted. Each normal task in Mbed OS can be executed in unprivileged mode. This exception is caught by the request handler, which deals with requests to invoke internal domain code or modify internal domain data. In addition, any unprivileged components (e.g., tasks, libs) cannot change the permission of the guard area as they have no permissions to control MPU. Hence code placed in this area cannot be modified in unprivileged mode.

Further, there is a guard area between the code of internal domain and external domain. It is set as non-executable in unprivileged mode. That is, a memory exception would be raised if it is accessed in unprivileged mode. This exception is caught by the request handler, which deals with requests to invoke internal domain code or modify internal domain data. In addition, any unprivileged components (e.g., tasks, libs) cannot change the permission of the guard area as they have no permissions to control MPU. Hence code placed in this area cannot be modified in unprivileged mode.

4.1. Overview

LIPS is complementary to Mbed uVisor. It divides data and code into two domains, i.e., internal domain and external domain, with different access permissions. All tasks’ data and code are in the external domain, secured by Mbed uVisor with different boxes. The critical system data structures and code, e.g., osRtxInfo and its pointer, are in the internal domain. They cannot be directly accessed by tasks, which must go through Domain Calls. Domain calls check whether certain tasks can access the code or data in the internal domain, and change the access permissions of these code or data based on the result. On the other hand, domain calls also make sure the access permissions are resumed back later. The overview of LIPS is shown in Fig. 3.

4.2. Domain isolation

Fig. 4 depicts the domain layout and access permissions used in the domain isolation. The memory space on both sides illustrates two domains’ default memory layout and access permissions, while the memory space in the middle illustrates how two domains are mapped to the flat memory space in the system. By default, the code and data in the internal domain are set to be not accessible in unprivileged mode and accessible in privileged mode. By contrast, external domain can be accessed without restriction in any mode.

Further, there is a guard area between the code of internal domain and external domain. It is set as non-executable in unprivileged mode. That is, a memory exception would be raised if it is accessed in unprivileged mode. This exception is caught by the request handler, which deals with requests to invoke internal domain code or modify internal domain data. In addition, any unprivileged components (e.g., tasks, libs) cannot change the permission of the guard area as they have no permissions to control MPU. Hence code placed in this area cannot be modified in unprivileged mode.

4.3. Domain calls

All accesses to the code and data in the internal domain should go through Domain calls. Each domain call checks whether one task is a system task or a normal task. If it is a normal task, the domain call raises an access fault and directly returns back to the task without further operation. If it is a system task, the domain call changes the internal domain’s access permission and that task can access the domain. Domain call mechanism consists of two main components, i.e., guard area and request handler.

Guard area and Request handler. To complete domain switching, tasks should execute particular instructions to trigger the process of controlling MPU. Perhaps designing specific SVC instructions may achieve the similar effect. However, this solution will leave extra...
interfaces, which still lay in the address space accessible by normal tasks. Thus, it cannot prevent a malicious normal task from reusing the interfaces to recall these SVC instructions. So we design guard area with entries to forbid such code reuse vulnerability.

As denoted above, guard area is an area between internal domain code and external domain code. There are several domain call entries inside this area. Invoking domain calls is actually invoking these domain call entries. Because guard area is set as non-executable in unprivileged mode, invoking domain call entries raises a memory fault exception. For each exception, Cortex-M processor has an exception handler and all register values are frozen before entering into the exception handler. Therefore, a request handler, which handles this memory fault handler, can obtain all register values and identify the role of tasks invoking domain call entries based on these values. Notice that, all exception handlers execute in privileged mode. So, the request handler can modify the configuration of MPU, i.e., change the access permissions of certain internal domain region. If the task is a system task, the internal domain’s access permission will be changed based on the register values. Otherwise, the access permission remains the same and the system would directly return to the task. Fig. 5 illustrates how to access internal domain code and data.

**Identifying tasks.** Tasks are identified in the request handler to determine whether to allow the access to the internal domain. This identification is based on the frozen register values in the processor, when the memory fault exception is raised. Specifically, the register value of sp is used. As presented in background information, each normal task in the system has a fixed stack range, which results in a fixed value range of sp register values for different tasks. Based on these values, the request handler can identify whether one task is a system task or a normal task.

**Domain Call Entry.** Each domain call entry in the guard area cannot be really executed by tasks in unprivileged mode, because of the configuration of the non-executable permission. They are essentially functions, which can only be invoked by the request handler in privileged mode. In particular, the frozen register pc value is used to determine which domain call entry should be executed. Furthermore, the register r0-r1 values are used to determine the accessed internal domain region and its access permission.

What is more, LIPS provides two domain call entries, i.e., region_lock and region_unlock. They support nine types of access pattern, as illustrated in Table 1. Each type has different region codes and opcodes. To be specific, region code denotes which region is accessed, i.e., 0 × 1 for data, 0 × 2 for code, and 0 × 3 for code and data. Alternatively, opcode denotes the access permission for different regions.

<table>
<thead>
<tr>
<th>Region code</th>
<th>Opcode</th>
<th>Access permission</th>
<th>Domain call entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 × 1: Data</td>
<td>0 × 3</td>
<td>NP X R/W</td>
<td>region_unlock</td>
</tr>
<tr>
<td>0 × 4:</td>
<td>0 × 11</td>
<td>NP</td>
<td>region_lock</td>
</tr>
<tr>
<td>0 × 2: Code</td>
<td>0 × 12</td>
<td>NP X R/W</td>
<td>region_lock</td>
</tr>
<tr>
<td>0 × 13</td>
<td>0 × 21</td>
<td>NP X R/R/W</td>
<td>region_unlock</td>
</tr>
</tbody>
</table>

**Algorithm 1: Request Handler.**

1. if EXC_FROM_PSP (lr) & EXC_FROM_NP (lr) then
2. 
3. else
4. return lr % not our condition, return
5. end if
6. pc = vmpu_unpriv_uint32_read (sp+(6×4)) % get the value of frozen runtime pc in unprivileged mode
7. if pc points to one of the Domain Calls then
8. if sp is in the range of a Secure Box then
9. ACCESS_FAULT_BY_NXPS % Failed, Normal Tasks require Domain Calls
10. RETURN_TO_CALLER_TASK
11. else
12. Restore r0 r1 from sp[0] and sp[1] % A System Task requires Domain Calls
13. switch (pc)
14. case 0x000040420: %entry point of lock
15. Call region_lock (r0, r1)
16. break
17. case 0x000040480: %entry point of unlock
18. Call region_unlock (r0, r1)
19. break
20. default: POSITION_FAULT
21. break
22. end switch
23. end if
24. else
25. return lr
26. end if
27. vmpu_unpriv_uint32_write ((sp+(6×4)), ((ulint32_t*)sp)[5]) %use lr to replace pc in task stack, so the program can avoid triggering exception again and again after Exception Handler returns
28. return lr;

Table 1

The Internal Domain Region’s Access Permission, before and after invoking Domain Call Entry with different codes (* denotes Any Permission, NP denotes No Permission. R, W and X denotes read, write and execute permission respectively.)

5. Implementation

We developed the prototype of LIPS based on Mbed OS (5.9) with uVisor (1.0). LIPS was implemented on a real development board, i.e., NXP FRDM-K64F with a Cortex-M4 32-bit core, Memory Protection Unit, 1024 KB Flash memory, and 256 KB Static RAM. Furthermore, LIPS adds 267 lines of C code and 20 lines of linker script code to uVisor for realizing its two core functionalities: region assignment and domain calls.

5.1. Region assignment in LIPS

MPU is configured based on regions. That is, the access permission can only be applied for a region in the memory space. However, the number of regions that can be configured in MPU is limited. So, LIPS has to be implemented with this restriction. Specifically, LIPS puts internal domain code and data in two regions respectively and each

---

region has a contiguous range of addresses. Table 2 shows how LIPS assigns different regions in the memory space. Among them, regions 0–4 are the default regions used by uVisor, following the predefined task sandbox design [18]. Regions 9–11 are assigned to guard area, internal domain code and data for LIPS. In particular, region 9 is for guard area with the non-executable permission in unprivileged mode and readable/executable permissions in privileged mode. Regions 10 and 11 are for internal domain code and data, and their access permissions are dynamic and can be changed by the request handler in privileged mode. Although the sizes of regions 9–11 are fixed in Table 2, they can be enlarged according to different requirements. For example, if more code needs to be put into internal domain code area, LIPS can be easily configured to meet the requirement. Furthermore, regions 5–8 are not used in LIPS at present. They are reserved for future use if needed.

To utilize regions 9–11 in Mbed OS with uVisor, we modify the linker script to add three new sections. Section protected_code and protected_data are used for internal domain code and data respectively. Section guard_area is for guard area. The related init code is also incorporated into the uVisor initialization process for using these new sections to establish the regions.

5.2. Domain calls

Domain calls are implemented by the guard area and the request handler. Currently, Domain Call Entries inside the guard area consist of two functions, i.e., region_lock and region_unlock. They can only be executed in privileged mode. When they are invoked in unprivileged mode, a memory fault exception is raised and several register values are frozen in the stack of a task, including (1) pc, storing the address of the next instruction. (2) lr, storing the return address. (3) ap, storing the stack address of the task. And finally (4) r0 ~ r3, storing function parameters. When entering the request handler, it will restore register values from the stack to identify the role of tasks, invoke specific domain call entries, as depicted in Listing 1. Specifically, the request handler identifies the role of a task according to the value of its frozen sp. If this value is inside the range of a secure box, this task is a normal task and the request handler will refuse the request. Otherwise, it is a system task and the following operation can continue. Then, it invokes the domain call entries based on pc. If the value is 0 × 00004020, it invokes region_lock. If the value is 0 × 00004080, it invokes region_unlock. The register r0 and r1 are used by two domain call entries to change access permissions for different internal domain parts. In the end of the request handler, it replaces the value of pc with the original lr to return to the task. The access permissions of the internal domain are also resumed back to the original one after the task invokes domain call entry, region_lock.

6. Evaluation

LIPS was evaluated from three angles, i.e., effectiveness, efficiency, and portability. First, we performed the attacks illustrated in Section 3 with and without LIPS. The results show that LIPS can succeed in defending the system from these attacks. Second, LIPS has few impact on task scheduling and little performance overhead. Third, LIPS can be implemented in these boards with MPU enabled.

6.1. Attack prevention

In this section, we depict how LIPS prevents the aforementioned attacks. The official example secure_number_store from uVisor is used here. It stores a number in a box and this number can only be written with LIPS. Specifically, we leverage OsEventObserver to manipulate the number. The number is originally set as 0 × 00000001, with only read permission. After that, a malicious task copies the contents of OsEventObserver and builds a Shadow. In this Shadow, the function pointer thread_destroy is modified to point to a function in this task, which manipulates the number above the pointer OsEventObs

---

Table 2
LIPS region assignment.

<table>
<thead>
<tr>
<th>Region num</th>
<th>Access permission</th>
<th>Start address</th>
<th>Size</th>
<th>Region name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P-R/W</td>
<td>0 × 0000 0000</td>
<td>4 GB</td>
<td>Background region</td>
</tr>
<tr>
<td>1</td>
<td>P-R/X, U/R/X</td>
<td>0 × 0000 0000</td>
<td>Code size</td>
<td>Code section</td>
</tr>
<tr>
<td>2</td>
<td>P-R/W, U/R/W/NX</td>
<td>0 × 2000 0400</td>
<td>SRAM size</td>
<td>SRAM</td>
</tr>
<tr>
<td>3</td>
<td>P-R/W, U/Varies</td>
<td>0 × 1FFF 0000 + Secure box</td>
<td>Secure box I-N stack size + Secure box stack area</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P-R/W, U/Varies</td>
<td>0 × 1FFF 0000 + Secure box 1 start offset</td>
<td>Secure box I-N context size + Secure box context area</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P-R/X, U/NX</td>
<td>0 × 0002 7000</td>
<td>1 KB</td>
<td>Guard area</td>
</tr>
<tr>
<td>10</td>
<td>P-R/X, U/Varies</td>
<td>0 × 0002 7420</td>
<td>4 KB</td>
<td>Internal domain code</td>
</tr>
<tr>
<td>11</td>
<td>P-R/W, U/Varies</td>
<td>0 × 2002 F20</td>
<td>4 KB</td>
<td>Internal domain data</td>
</tr>
</tbody>
</table>

---

Fig. 6. Effectiveness evaluation.

---

pointing to the original OsEventObserver is also changed to point to this Shadow. In order to call this task itself, it also registers a button event to trigger task deleting operation, which would invoke the function thread_destroy. Because this function pointer is changed to the function inside this task, which changes the number, the target number is finally changed after a button is pressed.

With LIPS, the attack is prevented in ➊, i.e., modifying the pointer OsEventObserve, which leads to a System HardFault. As a result, the consequent steps can be prevented.

Further, we expand our evaluation with more IoT applications deployed with Mbed uVisor. They demonstrate the use of different features in uVisor (secure LED blinking, secure multi-thread processing, debug information printing and secure AES cipher services). Besides secure number store, uVisor IRQ blinky and Threaded blinky are also official applications provided with development board and written by ARM, and others (i.e., NuMaker_mbed_uVisor and uvisor_debug_fault) are from the individual developer. In addition to the aforementioned data vulnerabilities, we also check if a specified system functionality (i.e., the function osRegisterForOsEvents for registering OsEventObserver) could be reused by a malicious normal task. The results are shown in Table 3, all BaseLine of IoT applications are not only unable to ensure the security of critical system data (e.g., OsEventObserver and osRtxInfo), but also failing to prevent reusing the function osRegisterForOsEvents. This finding is reasonable because Mbed executes tasks in flat memory model, which exposes the address of function osRtxInfo). In fact, the cause of these attacks relies on particular system data structures (i.e., osRtxInfo and OsEventObserver in context switching), whose vulnerability is an endogenous system problem and will not disappear by simply changing applications. Therefore we believe the vulnerabilities reported in this paper are widely seen in IoT applications as long as they use Mbed uVisor to enforce the task sandboxing. By comparison, with the secure enhancement by LIPS, such vulnerabilities will be restricted and mitigated, which eventually convinces us to ensure LIPS' effectiveness.

### 6.2. Performance analysis

**Overhead on Task Scheduler.** Mbed OS adopts the round-robin architecture for the task scheduler, which makes sure that each task is assigned a time slot for running in the processor. Once the assigned time slot is consumed, one task is scheduled out of the processor and another task runs. This task scheduling mechanism has a potential threat to LIPS. For example, there are two tasks A (system task) and B (normal task). In one time slot, task A is accessing the internal domain and before it invokes the domain call entry region_lock to lock the internal domain, it is scheduled out of the processor. That is, the internal domain can be accessed by task B, which leads to a potential attack. Therefore, the task scheduler in Mbed OS is patched by LIPS to make sure that, the internal domain is locked again before task scheduling. Because the overhead happens when the system schedules the system task out and schedules the normal task in, we only measured this process with and without LIPS. The result shows that without LIPS, the task scheduling consumes 4.44 ms and 4.46 ms with LIPS. So, the patch from LIPS only adds 0.4% extra latency.

**Overhead on the Real System.** To evaluate the performance overhead on the real system, we deployed LIPS on a simulated sweeping robot. The work flow of this robot has two stages, i.e., Waiting for Commands (WC) and Doing Cleaning (DC). The first stage is to wait for the commands from the user, while the second stage is to do the cleaning. The time for the second stage depends on the size of the cleaning area. So, in order to decrease the time, we set the area as 0.5 square meter. To measure the overhead in both stages, we place the code of command parsing and floor sweeping in Internal domain. The results show that the actual execution time and the times of accessing the code in internal domain have a distinctly positive correlation. To be specific, Time (WC)=0.44+0.004n (s), where n represents the number of requests for internal domain during the execution of WC. Simultaneously, DC can be expressed as Time (DC)= 10+0.004n (s). In this test, the access number is set to 1 (means codes in internal domain just need one time of execution), the final benchmark scores reach 0.3% (during WC) and 0.01% (during DC) performance overhead on average, showing the feasibility of the prototype.

### 6.3. Portability analysis

Table 4 shows the portability of LIPS on various platforms. STM32-F429Discovery\(^3\) and Giant Gecko\(^9\) both support Mbed OS and uVisor. Though they only have 8 MPU regions, it is still convenient to deploy the core components in LIPS as Mbed uVisor requires only 5 regions, as shown in Table 2. In addition, if a developer wants to define the permission of a specified memory space, he is suggested to use sub-regions (which are available in this type of MPU) first, instead of assigning a new MPU region. While our prototype does not support sub-regions, it has to assign new MPU regions to control the permission of a specified memory space. Further, LPC 1768,\(^10\) Nucleo-32\(^11\) and Nucleo-64\(^12\) are available for Mbed, but LIPS cannot run on these devices because of the lack of uVisor. The Micro Evaluation board,\(^13\) which executes FreeRTOS and does not support uVisor, is also not feasible. In fact, the uvisor Guide\(^22\) claims that uvisor can be ported to not only Mbed-enabled platforms but also CMSIS RTOS, so the portability of LIPS is also ensured.

### 7. Related work

Schemes against control-flow attacks in embedded systems. As the vulnerabilities reported in this paper break the task sandboxing by compromising the privileged control-flow paths, schemes aiming to mitigate such attacks are retrieved first. For example, Control-Flow Integrity (CFI)\(^[23–27]\) and Code Pointers Integrity (CPI)\(^[28]\)

---

are efficient ways of solving such issue, while they may both focus on desktop systems. For CFI, performance and precision are two key factors affecting its implementation in real-world systems. For instance, since all indirect controls need to be tracked, the overhead of the CFI is pretty high. However, though a fine-grained CFI research illustrates that it can reduce the overhead [26], most precise CFI still remains contingent. CFI CaRe [25] protects the CFI metadata by leveraging TrustZone-M security extension. It introduces the interrupt-aware CFI to make sure the integrity of the exception handler. In addition, most of explorations in CFI still depend on the virtual memory [28] or specific compiler features [29]. Our technique builds protection domain for vital system-maintained data structures, preventing the flow of the task switching from being hijacked.

**Protection domains for embedded systems.** Due to the resource limitation, it seems a challenge work to build protection domains for Cortex-M processors. To the best of our knowledge, one of the most relevant explorations is Hips [30]. It 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. Meanwhile, Microstache [31] achieves similar isolation through a specialized hardware mechanism and new process abstraction on x86-64 platform. These solutions require either specific registers (e.g., TxSZ) or powerful peripherals (e.g., MMU), which are not currently supported in Cortex-M processors. On the other hand, with rudimentary memory protection support (i.e., MPU) found in IoT devices, LIPS builds separate domains, making it easier for the deployment.

**Other IoT security frameworks.** It has been well established from a variety of studies that particular frameworks are being proposed [14, 29, 32–36] to discuss how to secure IoT devices. Among them, FreeRTOS [14] designs coarse-grained isolation by separating tasks in different privileges. EPOXY [29] instruments particular instructions to redirect them to run in privileged mode. Therefore, the rest unprivileged program cannot use these instructions to access the critical hardware resources. But the high frequency of the instruction overlay will cause a huge slowdown. ACES [35] is a LLVM-compatible compiler which automatically infers and enforces inter-component isolation for bare-metal systems. With this scheme, a single task may be cut into small compartments so that it would affect the task scheduling. In addition, with identifying the reachable memory regions of realtime tasks, MINION [36] changes the regions for each task through context switching process. However, such approaches are not seen in industry yet. Though providing effective isolation, TrustLite [33], TyTan [32] and Sancus [34] all require specific modifications to the circuit of processors, which show poor portability. ARM TrustZone [37] is available in all spectrums of Cortex-A and the coming Cortex-M processors which are based on ARMv8-M [38]. While Armv8-M based IoT devices do not support this feature.

**Ensuring data security from the edge.** Besides protecting vital data on aspects of devices, the issue of the protection from the field of edge is also a hot topic. The past decade has seen the rapid development of Cyber–Physical Systems (CPS) in improving intelligent human living environments [39]. Cyber–Physical–Social Systems extend CPS to combine social space of IoT and act as an important role to improve personalized services through quality prediction [40, 41], intelligent recommendation [42, 43], etc. As such services often rely on big data, the huge value make the security of these data more important. For instance, focusing on data confidentiality, Li et al. [44] use encryption techniques (online/offline signatures) to enforce the security of users’ data remotely. Moreover, Yang et al. [45] provide a research on aspects of data integrity, which requires the accuracy and consistency of users’ data. In addition, ABKS-UR [46] proposes a secure data search scheme which enables a user to securely search over encrypted data through keywords. Zhou et al. [47] explore particular ways to improve the accuracy and reduce the false rate in anomaly detection for industrial big data. At last, to solve the interactions between users and entities in multiple trust domains, researches [48] on access control systems in cloud computing environment is important for a wide range of scientific and industrial processes as well. Though still focusing on the veracity of data, above-mentioned schemes are for handling the overall situation, paying less attention to potential vulnerabilities which may compromise the data in an individual IoT device.

### 8. Conclusion

This paper takes ARM Mbed uVvisor as an example to argue that existing hypervisor-based task sandboxing mechanisms, which are applied to provide task isolation for data security, do not address the problem that particular system-maintained data structure can be exploited to enable a malicious task to fail the task sandbox and access other tasks’ data. Therefore the forged data would drastically reduce the effectiveness of a service. The research has also shown that a novel scheme with a special protection domain can be designed to protect the system-maintained data structures and improve the security of data stored in each sandbox, which further mitigates such effectiveness reduction. So we design LIPS that provides lightweight intra-mode privilege separation by separating such data structures from all tasks in two isolated domains. Experimental results not only reveal the influence of the vulnerabilities, but also prove the efficiency of the associated protection, which has small runtime overheads and good portability.

### Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to [https://doi.org/10.1016/j.sysarc.2021.102114](https://doi.org/10.1016/j.sysarc.2021.102114).

## Acknowledgments

The authors would like to thank the anonymous reviewers for their critical suggestions that greatly improved the paper quality. This work is supported by the National Key R&D Program of China (No. 2019YFB1706000).

## References


<table>
<thead>
<tr>
<th>NO.</th>
<th>Platform</th>
<th>Architecture</th>
<th>Mbed(ver 5.4-5.9)</th>
<th>uVisor(ver 1.0)</th>
<th>LIPS</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FRDM-K64F</td>
<td>Cortex-M4, 12 MPU regions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>soft/hardware support</td>
</tr>
<tr>
<td>2</td>
<td>STM32F429Discovery</td>
<td>Cortex-M4, 8 MPU regions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>software support</td>
</tr>
<tr>
<td>3</td>
<td>Giant Gecko</td>
<td>Cortex-M3, 8 MPU regions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>soft/hardware support</td>
</tr>
<tr>
<td>4</td>
<td>LPC 1768</td>
<td>Cortex-M3, No MPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>lack uVisor support</td>
</tr>
<tr>
<td>5</td>
<td>STM32L031 Nucleo-32</td>
<td>Cortex-M0, No MPU</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>lack uVisor support</td>
</tr>
<tr>
<td>6</td>
<td>STM32L053 Nucleo-64</td>
<td>Cortex-M0, No MPU</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>lack uVisor support</td>
</tr>
<tr>
<td>7</td>
<td>Micro Evaluation board</td>
<td>Cortex-M3, 8 MPU regions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>FreeRTOS</td>
</tr>
</tbody>
</table>
D. Huo et al.
Journal of Systems Architecture 116 (2021) 102114


Dongdong Huo received his M.S. degree from the University of Science and Technology of China, in 2014. He is currently completing his Ph.D. degree in the University of Chinese Academy of Sciences. He is an assistant professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research focuses on system security.

Chen Cao received the B.S. degree from the China University of Mining and Technology, and the Ph.D. degree from the University of Chinese Academy of Sciences. He was a postdoctoral fellow with the Cyber Security Lab, The College of Information Sciences and Technology, The Pennsylvania State University. He is an assistant professor now in the Pennsylvania State University, The Behrend College. His research interests are in operating system design and implementation and system security.
Peng Liu (M’99) received the B.S. and M.S. degrees from the University of Science and Technology of China, Hefei, China, and the Ph.D. degree from George Mason University, Fairfax, VA, USA, in 1999. He is a Professor of information sciences and technology, the Founding Director of the Center for Cyber-Security, Information Privacy, and Trust, and the Founding Director of the Cyber Security Laboratory, Pennsylvania State University, State College, PA, USA. He has authored or co-authored a monograph and over 260 refereed technical papers. His research has been sponsored by the U.S. National Science Foundation, ARO, AFOSR, DARPA, DHS, DOE, AFRL, NSA, TTC, CISCO, and HP. His current research interests include computer and network security. Dr. Liu has served on over 100 Program Committees and reviewed papers for numerous journals.

Yazhe Wang received the B.S. and M.S. degrees from the Shandong University, and the Ph.D. degree from the Institute of Software, Chinese Academy of Science, in 2010. He is the senior engineer of cybersecurity, principal investigator of the 5th laboratory with the Institute of Information Engineering, Chinese Academy of Sciences. His research focuses on IoT security, AI security, blockchain security, edge computing security, network and system security.

Mingxuan Li received his B.S. degree from the Zhengzhou University, in 2016. He is currently completing his Ph.D. degree in the University of Chinese Academy of Sciences. His research focuses on system security and blockchain.

Zhen Xu received his Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, in 2005. He is the professorate senior engineer of cybersecurity, director of the 5th laboratory with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests are cloud computing security, trusted computing, mobile internet security, network and system security.