Understanding MPU Usage in Microcontroller-based Systems in the Wild

Wei Zhou*†
School of Cyber Science and Engineering
Huazhong University of Science and Technology
Wuhan, China

Zhouqi Jiang*
School of Cyber Science and Engineering
Huazhong University of Science and Technology
Wuhan, China

Le Guan
School of Computing
University of Georgia
Athens, USA

***Abstract*—As more and more microcontroller-based embedded devices are connected to the Internet, as part of the Internet-of-Things (IoT), previously less tested (and insecure) devices are exposed to miscreants. To prevent them from being compromised, the memory protection unit (MPU), which is readily available on many of these devices, has the potential to play an important role in enforcing defense mechanisms. In this work, we comprehensively studied the MPU adoption in top operating systems for microcontrollers. Specifically, we investigate whether MPU is supported, how it is used, and whether the claimed security requirement has been effectively achieved by using it. We conclude that due to the added complexities, incompatibility, and fragmented programming interface, MPUs have not received wide adoption in real products. Moreover, although the MPU was developed for security purposes, it rarely fulfills its designed functionality and can be easily circumvented in many settings. We showcase concrete attacks to FreeRTOS and RIoT in this regard. Finally, we discussed fundamental causes to explain this situation. We hope our findings can inspire research on novel usage of MPU in microcontrollers.**

I. Introduction

Deeply embedded systems, which are powered by microcontroller units (MCUs), have long been used in closed environments, such as industrial plants and vehicle communication systems. The reliability and robustness of such systems were persistently tested in the past decades. However, these tests were conducted in a benign environment. That is, it is assumed that no adversary could actively penetrate the system. Unfortunately, this landscape has changed as more and more embedded devices are exposed to the Internet, where everyone can launch attacks remotely.

Since these systems are typically programmed using system programming languages such as C/C++, memory errors pose a great threat to their security, especially considering that many third-party libraries run at the same privilege level as that of the core program. As a security mechanism, ARM, a leading chip designer for MCU, proposed, designed, and implemented the memory protection unit (MPU) to protect their chips. The MPU is a low-cost security extension to ARM MCUs that safeguards certain sensitive memory regions in case a piece of code is compromised. Therefore, it is a promising mitigation technique to memory vulnerabilities. Other MCUs such as MSP430 FRAM followed this design and implemented similar hardware.

*Also with Hubei Key Laboratory of Distributed System, Hubei Engineering Research Center on Big Data Security.
†Corresponding author.

Workshop on Binary Analysis Research (BAR) 2023
3 March 2023, San Diego, CA, USA
ISBN 1-891562-84-3
https://dx.doi.org/10.14722/bar.2023.23007
www.ndss-symposium.org

In this work, we comprehensively investigated the usage of MPU in top operating systems (OSs) for microcontrollers. More specifically, we are interested in whether MPU is supported, how it is used, which kinds of security are claimed, and whether they are effectively achieved. To our surprise, we found MPU is seldom used in real products, although popular OSs largely support it. We attribute this to four reasons. First, MPU is typically a default off feature on OSs and needs the developers to explicitly enable it. Second, the added complexities and compatibility issues make the manufacturers reluctant to invest in R&D. Third, the introduced overhead on performance and resource consumption makes it unsuitable for mission-critical tasks. Finally, although MPU was mostly used for security purposes, it rarely fulfills its designed functionality and can be easily circumvented in many settings. We showcase a concrete attack to FreeRTOS and RIoT in this regard.

Looking forward, we discuss common pitfalls in using MPU and give recommendations on fixing the reported concerns. We also discuss how future generations of hardware (ARM TrustZone for MCUs) can make MCU-based systems more secure.

Responsible Disclosure. All the vulnerabilities described in this paper have been reported to the corresponding vendors with technical details. In particular, CVE-2021-43997 has been confirmed and patched by AWS on 11/12/2021[1].

II. Background

A. What Can MPUs Do?

ARM is a leader in the MCU market. Its Cortex-M series processors which are based on the ARM-V7M or ARM-V8M architecture consume a much lower silicon area and are highly optimized to be energy-efficient and responsive to interrupts, making them ideal for mission-critical tasks such as industrial applications. On the other hand, due to the cost- and power-efficient design, Cortex-M processors sacrifice

[1] https://www.freertos.org/security/security_updates.html.

security. For example, the memory management unit (MMU) is absent, which has been used to in many security solutions for commodity OSs [4].

As a stripped-down version of MMU, MPU enforces lightweight access control for MCUs. Specifically, an MPU provides a fixed number of hardware registers (e.g., eight in Cortex-M4), and enforces access control rules for the specified memory region. Concretely, a region can have individual memory access permissions (e.g., read/write/execution) and memory attributes (e.g., cacheability and shareability), depending on the privilege mode the code is running. In ARM Cortex-M MCUs, only two privilege levels are supported, corresponding to Ring 0 and Ring 3 in X86. If memory access violates the permissions programmed in MPU, the processor generates a MemManage fault (which is usually escalated into a HardFault). The MPU has been supported in mainstream ARM MCUs.

B. How to Program MPUs?

ARMv7-M. ARMv7-M is a widely adopted architecture in ARM MCU chips. It empowers the popular Cortex M0/M0+/M3/M4/M7 series processors. MPUs in ARMv7-M enforce access control based on regions (i.e., specific memory range). Depending on the implementation, 8-16 memory regions can be supported. The range, permission, and attribute of each memory region are programmed via the *Region Base Address Register* (RBAR) and *Region Attribute/Size Register* (RASR). Note that only privileged code can access these registers. Also, a region must be aligned. That is, a region must start at an address that is multiple of its size. MPU also imposes restrictions on the region size. More specifically, the size of a region must be (1) at least 32 bytes, and (2) a power of two. Consequently, an arbitrarily sized region must be emulated by an over-sized region (thus up to half of the region size is wasted) or be pieced together by multiple smaller regions (thus MPU registers are wasted). Large regions can be further divided into eight equally sized sub-regions which can be activated individually. However, these sub-regions all inherit the same permission settings specified by the parent region. This feature is achieved by configuring the *sub-region disable field* (SRD) of RASR. Regions can overlap and higher-numbered regions have precedence. Finally, there is a special region with the least priority that maps the entire system memory. It only takes effect in privileged mode. When this region is enabled, it sets the default access permissions for privileged mode execution. In other words, when memory access does not fall into any other regions, this default permission is enforced. In contrast, when memory access occurs in unprivileged mode, it must explicitly fall into a region. Otherwise, a fault is raised.

ARMv8-M. The ARMv8-M architecture is the successor of the popular ARMv7-M architecture. It empowers the Cortex-M23/M33/M35P series MCUs, which has improved MPU support. First, the number of MCU regions has increased. Second, the restrictions on the region alignment are relaxed. Third, the size restriction is removed because the ending address of a region can be directly specified in the Limit address. All of these improvements lead to more efficient usage of the system memory. A requested memory region that had to be pieced together by several smaller hardware regions can now be directly configured with only one hardware region. Naturally, legacy features that were designed for efficient usage of system memory, such as the sub-region mechanism and the overlapped region, have been abandoned.

III. MPU Usage in the Wild

Taking advantage of the MPU, MCU-based embedded systems can implement several memory-related security features, such as isolating critical services from untrusted code or detecting memory errors. To understand how MPUs are used in real MCU products, we performed a comprehensive survey, covering 30 top IoT operating systems (OSs) [34]. Some OSs (e.g., VxWorks) support both MCUs and traditional processors. We only consider their MCU variants. Some OSs (e.g., Windows 10 IoT) only support traditional processors. We did not include them in the table. Due to diverse terminologies of the same security feature used by IoT OSs, we summarize six unified names to refer to these commonly supported security features, which can be easily mapped to traditional memory isolation features.

- **Code Integrity Protection (CIP):** Code regions can be set as non-writable to prevent code injection and manipulation.
- **Data Execution Prevention (DEP):** Data regions like stack or heap can be set as non-executable to prevent buffer overflow exploitation. CIP and DEP map to W^X [38] or executable-space protection [35] in general computing platforms.
- **Coarse-grained StackGuard (CSG):** An inaccessible memory hole is placed at the stack boundary to detect stack overflows. Note that CSG can only detect overflows to the entire stack region, not confusing with traditional StackGuard to function-level stack frames.
- **Kernel Memory Isolation (KMI):** User mode (unprivileged) code cannot access any memory belonging to the kernel space without invoking system calls, which is similar to the user space and kernel space separation [37] in modern OSs.
- **User Task Memory Isolation (TMI):** User mode (unprivileged) tasks can only access its own memory except explicitly shared memory regions that belong to other tasks or kernel, which is similar to the process isolation [36] in modern modern OSs.
- **Peripherals Isolation (PI):** Peripheral access is restricted to tasks that actually need it.

For each OS, we studied whether MPU is used and how each of the six features is supported. In particular, a security feature can be mandatory, optional or unsupported. An optional feature can further be default-on or default-off with configurations. For open-source OSs, we mainly relied on manual review of the source code and documentation. Therefore, we could get the most accurate and comprehensive results. For propriety OSs, we can only resort to public resources such as official and third-party websites. Interestingly, FreeRTOS and embOS-MPU provides two separated versions with and without MPU support, and we list them as two different OSs. The results are summarized in Table I. In the following sections and Appendix A, we use some case studies to explain the table.

TABLE I. SUMMARY OF MPU USAGE AND ADOPTION ON MCU OSS

			MPU Usage					
	OS	MPU Support	CIP	DEP	KMI	TMI	CSG	PI
	Contiki [11]	None	-	-	-	-	-	-
	LiteOS [15]	None	-	-	-	-	-	-
	RT-Thread [22]	None	-	-	-	-	-	-
	OpenWrt [20]	None	-	-	-	-	-	-
	TinyOS [5]	None	-	-	-	-	-	-
Open-source	Mongoose OS [17]	None	-	-	-	-	-	-
	FreeRTOS [8]	None	-	-	-	-	-	-
	FreeRTOS-MPU [7]	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	-	-
	RIoT [21]	Optional	Default-off	-	-	-	Default-off	-
	Apache Mynewt [2]	None	-	-	-	-	-	-
	Zephyr [27]	Optional	Default-on	Default-on	Default-off	Default-off	Default-off	Default-off
	MbedOS [9]	Optional	Default-on	Default-on	-	-	-	-
	TizenRT [24]	Optional	Default-off	Default-off	Default-off	Default-off	Default-off	-
	CMSIS-Keil RTX [6]	Optional	Default-off	Default-off	-	Default-off	-	Default-off
	Azure RTOS ThreadX [10]	Optional	Default-off	Default-off	Default-off	-	-	-
	Micrium OS [16]	None	∅	∅	∅	∅	∅	∅
	Device OS [12]	None	∅	∅	∅	∅	∅	∅
	VxWorks [26]	None	∅	∅	∅	∅	∅	∅
Proprietary	embOS [14]	None	∅	∅	∅	∅	∅	∅
	embOS-MPU [13]	Mandatory	Mandatory	∅	Mandatory	Mandatory	∅	∅
	NXP MQX RTOS [18]	Optional	Default-on	∅	Default-on	Default-on	∅	∅
	Nucleus RTOS [19]	Optional	∅	∅	Default-on	Default-on	∅	∅
	SafeRTOS [23]	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	∅	∅

-: Not supported at the time of submission. ∅: No public resource available to firmly decide.

A. *Case Study: RIoT*

When all the MPU related protections are enabled (i.e., The macros of `MODULE_MPU_NOEXEC_RAM` and `MODULE_MPU_STACK_GUARD` are set), RIoT enforces DEP for the whole RAM and activates CSG using three MPU memory regions.

DEP: RIoT enables the MPU region number 0 (recall that lower number means lower priority) to cover the whole RAM region (from `0x20000000` to `0x3fffffff`) and set the permission as readable/writable but non-executable. Then, code sections are explicitly granted the execution privilege.

CSG: RIoT defines the permission of the last 32 bytes (the smallest MPU region) of the main stack as read-only via the MPU region number 1. Similarly, when switching to another task, RIoT configures the last 32 bytes of the target task stack as read-only via the MPU region number 1. With this 32-byte memory hole, exhausting the stack memory will trigger a MemManage fault.

B. *Case Study: FreeRTOS-MPU*

The basic FreeRTOS does not enable MPU. For the MPU-enabled version, called FreeRTOS-MPU, we list the supported security features in Table II.

CIP & DEP: The code sections are configured as read-only. User task stacks and kernel data regions as configured as non-executable. Accesses to unmapped regions trigger memory faults.

KMI: The privileged kernel APIs are located in the first part of the flash memory, which maps to the second MPU region. It is set to be read-only in privileged mode, and inaccessible in unprivileged mode. The kernel-maintained data (e.g., current control block) are located in a separated MPU region in RAM which is readable and writable in privileged mode. In addition, to expose kernel APIs (e.g., `vTaskDelay`) to unprivileged tasks, FreeRTOS-MPU takes advantage of software interrupt (i.e., `SVC`).

TABLE II. MEMORY MAP OF FREERTOS-MPU ON ARMV7-M

Region No.	Range	Usage	Privilege Mode	Permission
0	flash_segement_start -flash_segemnt_end	Non-writable Code Segment	Privileged Unprivileged	`r-x` `r-x`
1	privileged_functions_start -privileged_functions_end	Kernel APIs Isolation	Privileged Unprivileged	`r-x` ∅
2	privileged_data_start -privileged_data_end	Kernel Data Isolation	Privileged Unprivileged	`rw-` ∅
3	0x40000000-0x5fffffff	Non-executable Peripherals	Privileged Unprivileged	`rw-` `rw-`
4	User Task Stack	User Task Stack Isolation	Privileged Unprivileged	`rw-` `rw-`

TABLE III. MEMORY MAP OF ZEPHYR ON ARMV7-M

Region No.	Range	Usage	Privilege Mode	Permission
0	Flash Region	Non-writable Code Segment	Privileged Unprivileged	`r-x` `r-x`
1	RAM Region	Kernel Object	Privileged Unprivileged	`rw-` ∅
2	Task Stack	Task Stack Isolation	Privileged Unprivileged	`rw-` `rw-`
3	Last 32 Bytes on Stack Bottom	Stack Guard	Privileged Unprivileged	`r--` ∅

TMI: Tasks can be created to run in either privileged mode or unprivileged (user) mode. A separated MPU region is reserved for each user-mode task and is inaccessible for any other tasks. By default, no memory is shared among user-mode tasks. To share any data, the user-mode task should explicitly use the queue or semaphore mechanisms provided by the kernel, which consume up to three user-definable MPU regions.

C. *Case Study: Zephyr*

Zephyr is among IoT OSs that make heavy use of MPU. It provides flexible protection based on users' needs. We summarize our findings in Table III.

CIP&DEP: The first MPU region maps to the flash memory which stores program text and read-only data. It is configured

TABLE IV. MEMORY MAP OF TIZEN ON ARMV7-M

Region No.	Range		Usage	Privilege Mode	Permission
0	Common Binary	Program Text	Non-writable Code Segment	Privileged	`r-x`
				Unprivileged	`r-x`
1		Read-only Data	Non-writable Data Segment	Privileged	`r--`
				Unprivileged	`r--`
2		Stack/Heap/Data	Common Stack/Heap/Data	Privileged	`rw-`
				Unprivileged	`rw-`
3	Apps Binary	Program Text	Non-writable Code Segment	Privileged	`r-x`
				Unprivileged	`r-x`
4		Read-only Data	Non-writable Data Segment	Privileged	`r--`
				Unprivileged	`r--`
5		Stack/Heap/Data	App Stack/Heap/Data	Privileged	`rw-`
				Unprivileged	`rw-`
6		Last 32 Bytes on Stack Bottom	Stack Guard	Privileged	`r--`
				Unprivileged	`r--`
7		User Task Stack	User Task Stack Isolation	Privileged	`rw-`
				Unprivileged	`rw-`

as read-only for all access. The SRAM region and any thread stack are non-executable. Accesses to unmapped regions trigger memory faults.

KMI: Zephyr makes sure that all kernel objects in SRAM (mapped in region #1) do not overlap with any user stacks (mapped in region #2). User tasks must invoke system calls to access kernel objects indirectly.

TMI&CSG: As mentioned before, Zephyr uses MPU region #2 to map task-specific stacks. Also, similar to RIoT, the last 32 bytes of each task's stack are mapped to MPU Region #3, which is configured as read-only for privileged mode and inaccessible for unprivileged mode. Therefore, during a task switch, the corresponding MPU registers need to be updated for the target task. With this mechanism, a task cannot access others' memory or exhaust its own allocated stack memory. To add flexibility, a user task can use the memory domain to gain access to additional memory. This memory region, which is implemented by MPU, is inheritable and can be assigned to multiple user or supervisor tasks.

PI: A peripheral driver instance is considered as a kernel object. Therefore, its range is pre-configured to be inaccessible by user tasks. To access peripherals, a user task must invoke system calls.

D. Case Study: Tizen

All the MPU-enabled security features in Tizen are optional. They can be activated by the following macros: `CONFIG_APP_BINARY_SEPARATION`, `CONFIG_OPTIMIZE_APP_RELOAD_TIME`, `CONFIG_SUPPORT_COMMON_BINARY` and `CONFIG_MPU_STACK_OVERFLOW_PROTECTION`. When fully enabled, the achieved security features are summarized in Table IV.

CIP&DEP: Tizen sets all the code, including user tasks in apps binaries and common libraries as read-only. The data regions of all tasks and common libraries are non-executable. Unmapped memory regions is default region which is configured to be accessible only in privileged mode.

KMI: The kernel resides in the default region, which can only be accessed in privileged mode. The user tasks communicate with the kernel via system calls.

TMI&CSG: Tizen enforces TMI&CSG protection using MPU regions #7 and #6, similar to Zephyr. Besides, Tizen supports dynamically loading application binaries into the systems. To achieve TMI for them, MPU regions #3, #4 and #5 are used.

E. Case Study: Mbed OS

The Mbed OS is backed ARM, which complies with the *Platform Security Architecture* (PSA), an ARM initiative for security standard of IoT devices. However, due to hardware restriction, not every piece of hardware can fully support PSA, including the popular ARM-V7M devices. Due to the missing parts, Mbed OS for ARM-V7M only supports **CIP** for ROM and **DEP** for SRAM.

IV. COMMON PITFALLS IN USING MPU IN THE WILD

Although using MPU offers obvious security benefits, we found that MPU does not receive wide adoption in the wild. For example, although FreeRTOS provides a variant with MPU port, AWS mainly uses the non-MPU version in its products [3]. In this section, we shed light on the root cause of this situation by discussing common pitfalls of using MPU with real examples.

A. Weak Protection

While major IoT OSs adopt MPU to implement some security features as discussed in Section III, we found that these claimed features do not always fulfil their designed benefits. In certain scenarios, they can be easily bypassed. We discuss how an attacker can break **KMI** in FreeRTOS-MPU and invalidate MPU features in RIoT.

FreeRTOS kernel APIs are located in a region that can only be accessed in privileged mode. For compatibility, MPU-enabled FreeRTOS does not redesign its kernel APIs to reflect MPU integration. Rather, it wraps the existing kernel APIs with the `xPortRaisePrivilege` and `vPortResetPrivilege`, which raise and drop its privilege temporarily for kernel function invocations respectively. For example, as shown in listing 1 if a user mode task needs for delay for a certain time, it has to invoke `MPU_vTaskDelay` which uses the `xPortRaisePrivilege` function to switch to privileged mode with `SVC` software interrupt before invoking the real FreeRTOS kernel function `vTaskDelay`. On the completion of `vTaskDelay`, it needs to drop the privilege by invoking `vPortResetPrivilege`. However, since firmware binaries embed all the static compiled programs as well as FreeRTOS kernel and system call APIs, the function `xPortRaisePrivilege` can be easily located via reverse-engineering. Once an attacker launches a control flow hijacking attack (e.g., by exploiting a memory error) on any user mode tasks, the hijacked task can escalate privilege via invoking `xPortRaisePrivilege`, but never drop the privilege. With the escalated privilege, the hijacked task can access any resources on the device.

```c
void MPU_vTaskDelay(TickType_t xTicksToDelay){
    BaseType_t xRunningPrivileged =xPortRaisePrivilege();
    vTaskDelay(xTicksToDelay);
    vPortResetPrivilege(xRunningPrivileged);
}
BaseType_t xPortRaisePrivilege(void){
    BaseType_t xRunningPrivileged;
    xRunningPrivileged = portIS_PRIVILEGED();
    /*If the CPU is not privileged, raise privilege.*/
    if (xRunningPrivileged == pdFALSE){
        portRAISE_PRIVILEGE();
    }
    return xRunningPrivileged;
}
#define portRAISE_PRIVILEGE() __asm volatile ("svc␣%0␣\n"
      ::"i" (portSVC_RAISE_PRIVILEGE) : "memory");
void prvSVCHandler(uint32_t* pulParam){
    ...
    case portSVC_RAISE_PRIVILEGE:
        if ((ulPC >= _syscalls_flash_start_) && (ulPC <=
            _syscalls_flash_end_)){
            __asm {
                /*Obtain control value.*/
                mrs ulReg, control
                /*Set privilege bit.*/
                bic ulReg, #1
                /*Write back control value*/
                msr control, ulReg
            }
        }
    break;
    ...
}
```

Listing 1. Kernel wrapper in FreeRTOS-MPU

To verify our observation, we built a firmware with FreeRTOS-MPU. It has a stack overflow bug in a task. By exploiting it, we overwrote the return address on the stack with the address of the function `xPortRaisePrivilege`. As a result, the executing privilege was escalated. Combining more sophisticated ROP programming, we were able to run arbitrary code with elevated privilege. Since all other MPU protections like TMI and KMI are based on the privilege separation, any privilege escalation attack will render these security mechanisms useless.

On the other hand, some basic protections such as **CIP**, **DEP**, and stack guard (**SG**) do not need to assign different permissions for privileged and unprivileged modes. For example, RIoT sets the permission as non-executable for the whole RAM and read-only for stack guard range regardless of the privilege mode, and it runs the entire firmware under privileged mode. However, MPU control registers (e.g., `MPU_CTRL`) are located in the system peripheral region, which can be accessed by any privileged code. Even worse, systems often provide easy-to-use driver APIs for MPU configurations (e.g., `mpu_enable` and `mpu_disable` in RIoT). This means once an attacker is able to carry out a control flow hijacking attack, they can directly invoke these functions to completely disable MPU protections.

B. *Incomplete Protection*

As shown in Table I, only a few MCU OSs support MPU-enabled security features, among which many are default-off. Furthermore, due to the hardware limitation of MPU, we found existing protections are too coarse-grained to be really useful in real world.

First, most systems only focus on memory protection and standard peripherals are not protected by default. Although there may be a few remaining MPU regions that can be configured individually by each user mode task, they are not very suitable for protecting small peripheral regions scattered in the address space due to the alignment problem and the limited number of MPU regions, as mentioned in Section II. For instance, the memory range for the Audio peripheral on MPS2+ FPGA prototyping system broad (Cortex-M4 AN386) is 0x40024000-0x40024FFF (16 Bytes).

Second, developers have the real-world need to assign separate access rights to interrupt handlers [29]. However, this is very challenging since ARM Nested Vector Interrupt Controller (NVIC) registers for interrupt line configurations are located in the system peripheral region, which can only be accessed in privileged mode.

Furthermore, although the memory regions that are not covered by MPU regions only allows privileged access, it enables all access permissions including read, write, and execute, which also has potential hazards. Assuming that an attacker exploits a stack overflow in a task stack, they cannot execute the malicious code in its stack due to MPU stack isolation. However, they can put malicious code in these un-mapped regions beforehand, and use a stack overflow attack to redirect the control flow to malicious code in these regions.

MPUs cannot resist hardware attacks. For example, they do not restrict peripherals as master, allowing them to access all memory (e.g., via DMA). If the attacker is able to establish access via JTAG or in the case of micro-probing, MPU cannot help at all.

C. *High Overhead*

We also found leveraging MPU to realize kernel or task memory isolation (**KMI** and **TMI**) incurs too much overhead in many real-world application scenarios. This is because each invocation to kernel API has to go through a full privilege mode switch. Since kernel APIs are frequently invoked, this poses a significant impact on real-time performance. Our experiment shows that one thousand privilege switch of FreeRTOS-MPU system takes 3.5ms in average on MPS2+ FPGA prototyping system broad (Cortex-M4 AN386) with 25MHZ CPU clock frequency. In a previous research [31], a similar result was obtained. Such high overhead rules out MPU's application in many scenarios having strict real-time constraints [31].

To realize **TMI**, the MPU regions need to be re-configured for different tasks and applications. For example, Tizen has to reset MPU regions #3-7 during the application switch and #6 and #7 during the task switch, which will also cause delays.

D. *Conflicting with Existing System Design*

To integrate MPU-based protection, some existing mechanisms have to be re-designed or even become incompatible. For example, user mode tasks of FreeRTOS-MPU cannot use dynamic queues because there is no shared memory between any two tasks as we mentioned in Section III. To overcome this, a task has to allocate memory statically and shares it with the peers by configuring an MPU region. Note that each peer needs the same MPU region for each queue. This applies to semaphores too, which is a special kind of queue (queue length is one). If a task needs multiple queues/semaphores or anther additional memory regions (e.g., a task may also need to write USB data to a DMA-mapped buffer in RAM), MPU resources would soon become exhausted. Similarly, Zephyr and Tizen also suffer from the same issue.

E. *Fragmented Programming Interface*

Other than the standard MPU offered by ARM, chip vendors tend to design proprietary MPUs to distinguish them from others. The customized MPUs—while mostly providing better security guarantees—impose a steep learning curve for developers and may discourage them from adopting MPUs or even lead to programming errors. For example, many of NXP's Kinetis series MCUs discard ARM MPU and integrate NXP's proprietary MPU called sysMPU, Compared to ARM MPUs which can only restrict the permission of CPU, sysMPU can restrict the permission of every memory reference generated by each bus master including peripherals. However, in our experiments, when sysMPU is enabled, by default, all peripherals cannot directly access RAM (i.e., via DMA) and this causes firmware hangs in many cases. To properly use it, developers have to correctly configure sysMPU, which requires a deep knowledge of sysMPU design. More likely, when developers face difficulties, they may simply give up and disable sysMPU as we have witnessed in many NXP demos.

Summary: IoT devices are low-cost energy-efficient devices. If more transistors are reserved for complex hardware security features, not only the price of SoC could be raised accordingly, but also increased power consumption rules out many applications in which thermal design power (TDP) concerns. Due to the reasons discussed earlier and many complaints we found in developer forums (e.g., [30]), only 30% of manufacturers implement one or more MCU security features in their products [1].

V. Suggestions

A. Minimizing Pitfalls

Developers can raise the bar to bypass MPU protections for attacker by following the suggestions below. As is demonstrated in Section IV-A, inappropriate implementation of system calls can be leveraged by attackers to break **KMI**. Thus, developers should pay more attention to functions involved in privilege escalations like system calls. To be specific, additional caller checks should be performed before system call invocations, and the kernel should make sure the privilege is dropped after system calls. Besides, some MPU-enable OSs just run the whole system at the privileged mode like RIoT. Therefore, a single vulnerability could grant attacker the capability to reconfigure MPUs, thereby disabling the desired security features. We recommend to drop privilege immediately after MPU configuration.

The ARMv7 architecture allows MPU regions to overlap, and higher-numbered regions always have precedence. Thus, during a task switching, in addition to re-configuring the MPU regions, system should make sure that unused MPU regions are disabled. It can prevent leaking permission from the previous task to the next one.

B. Region Usage Optimization

A limitation with MPU is that only a few number of regions are supported (e.g., only eight regions for Cortex-M0+/M3/M4). Creatively using the *sub-region disable field* (SRD) of `BASR` can overcome this limitation to some extent. As mentioned in Section II, each memory region can be divided into eight sub-regions, which can be enabled/disabled individually. Suppose a developer needs to allocate a 5KB region and a 3KB region for two tasks. Without sub-regions, the developer has to configure two regions of 8KB and 4KB separately. There is a waste of 3KB and 1KB of memory space correspondingly. With sub-regions, the same 8KB region can be shared between the two tasks. Specifically, when running task one, MPU is configured so that the highest three sub-regions of the 8KB region are disabled. When running task two, MPU is configured so that the lowest two sub-regions of the 8KB region are disabled. In this way, the 8KB memory block is reused by the two tasks without wasting any memory.

The *default ARMv7-M address map* [25] defines the default memory access permissions and attributes of memory regions when MPU is not present. Therefore, it can be used to reduce MPU region consumption. Specifically, the `IACCVIOL` field of *Memory Management Fault Status Register* (MMFSR) indicates whether an Execute Never (XN) fault has been triggered when memory access violates the permissions of the default memory map (e.g., non-executable for standard and system peripheral regions). If we can leverage this feature to set XN, an MPU region can be saved.

Compiler-based approaches can intelligently figure out an optimized MPU configuration for certain tasks [31], [28]. For example, MINION [31] uses k-means clustering to group memory sections having similar access permissions together to minimize the number of required MPU regions.

C. New Hardware Features

Although ARMv8-M provides a more powerful MPU design, it cannot solve all the problems. In the long term, we expect a redesigned architecture that fundamentally addresses the illustrated insecurity and inflexibility in a lightweight way. Along this direction, hardware-based solutions have been proposed [32], [33]. The most representative work is TrustLite[32]. It is a radical hardware redesign that efficiently implements many novel security primitives (e.g., execution-aware MPU, secure loader) for embedded devices. In addition, the ARMv8-M architecture has extended TrustZone technology to Cortex-M series processors, allowing for TEE to be deployed in MCUs. For example, Mbed OS already implemented PSA which provides the root-of-trust service and infrastructure for developing secure IoT applications.

VI. Conclusion

In this work, we investigate the MPU usage of popular cyber-physical systems in the wild. To our surprise, we found that MPU as a ready-to-use security feature for protecting microcontroller is rarely used in real-world products. We studied the source code of MCU OSs in an attempt to find explanations for this situation and eventually identified four reasons. Unfortunately, some of them are fundamental and not remedial in a short term. We give recommendations for better use of MPU and hope our findings can inspire research on novel usage of MPU and new hardware retrofitting for MCUs.

Acknowledgment

We thank all the anonymous reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China (Grant No.62202188), The National Key R&D Program of China (Grant No.2022YFB31033400), and Cisco Research.

REFERENCES

[1] "2019 Embedded Markets Study," https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, 2019.

[2] "Apache Mynewt," https://github.com/apache/mynewt-core, 2020.

[3] "FreeRTOS AWS Reference Integrations," https://github.com/aws/amazon-freertos, 2020.

[4] "Memory management unit," https://en.wikipedia.org/wiki/Memory_management_unit, 2020.

[5] "TinyOS," https://github.com/tinyos/tinyos-main, 2020.

[6] "CMSIS-Zone," https://www.keil.com/pack/doc/CMSIS/Zone/html/index.html, 2021.

[7] "FreeRTOS: Memory Protection Unit (MPU) Support," https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html, 2021.

[8] "FreeRTOS: Real-time operating system for microcontrollers," https://www.freertos.org/, 2021.

[9] "Mbed: Rapid IoT device development," https://os.mbed.com/, 2021.

[10] "Azure RTOS ThreadX," https://github.com/azure-rtos/threadx, 2022.

[11] "Contiki: The Open Source Operating System for the Internet of Things," http://www.contiki-os.org/, 2022.

[12] "Device OS," https://docs.particle.io/getting-started/device-os/introduction-to-device-os/, 2022.

[13] "embOS-MPU — Comprehensive memory protection," https://www.segger.com/products/rtos/embos/editions/embos-mpu/embos-mpu-basic-concepts/, 2022.

[14] "embOS – The Leading RTOS (Real Time Operating System)," https://www.segger.com/products/rtos/embos/, 2022.

[15] "Huawei LiteOS," https://github.com/LiteOS/LiteOS, 2022.

[16] "Micrium OS," https://www.silabs.com/developers/micrium-os, 2022.

[17] "Mongoose OS," https://mongoose-os.com/, 2022.

[18] "MQX Software Solutions," https://www.nxp.com/design/software/embedded-software/mqx-software-solutions:MQX_HOME, 2022.

[19] "Nucleus RTOS," https://www.mentor.com/embedded-software/nucleus/, 2022.

[20] "OpenWrt," https://openwrt.org/start, 2022.

[21] "RIoT," https://github.com/RIOT-OS/RIOT, 2022.

[22] "RT-Thread," https://www.rt-thread.org/, 2022.

[23] "SAFERTOS CORE Overview," https://www.highintegritysystems.com/safertos-core/, 2022.

[24] "Samsung TizenRT," https://github.com/Samsung/TizenRT, 2022.

[25] "System address map," https://developer.arm.com/documentation/ddi0489/c/programmers-model/system-address-map, 2022, last accessed: 2022-05-01.

[26] "VXWORKS," https://www.windriver.com/products/vxworks/, 2022.

[27] "Zephyr Project," https://docs.zephyrproject.org/latest/introduction/index.html, 2022.

[28] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic Compartments for Embedded Systems," in *27th USENIX Security*, 2018.

[29] N. Community, "MPU advanced features?" https://community.nxp.com/message/1144517?commentID=1144517#comment-1144517, 2019.

[30] S. Community, "FAQ: Ethernet not working on STM32H7x3," https://community.st.com/s/article/FAQ-Ethernet-not-working-on-STM32H7x3, 2018.

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

[32] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[33] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers," in *RAID*. Springer, 2017.

[34] Slashdot., "Best Real-Time Operating Systems (RTOS) of 2023," https://slashdot.org/software/iot-operating-systems/, 2023.

[35] Wikipedia, "Executable space protection," https://en.wikipedia.org/wiki/Executable_space_protection#Windows, 2022, last accessed: 2023-02-01.

[36] Wikipedia, "Process isolation," https://en.wikipedia.org/wiki/Process_isolation, 2022, last accessed: 2023-02-01.

[37] Wikipedia, "User space and kernel space," https://en.wikipedia.org/wiki/User_space_and_kernel_space, 2022, last accessed: 2023-02-01.

[38] Wikipedia, "W^X," https://en.wikipedia.org/wiki/W%5EX, 2022, last accessed: 2023-02-01.

APPENDIX

We detail more case studies of MPU usage in open-source IoT OSs (summarized in Table I) as follows.

A. *Case Study: Keil RTX*

Keil RTX can leverage CMSIS-Zone [6] to isolate tasks and peripherals from each other (i.e., **TMI** and **PI** are supported). An important concept in RTX is *execution zone*, in which users can configure the access permissions (i.e., readable, writable, and executable) to a bundle of memory regions and peripherals (i.e., **CIP** and **DEP** are also supported). This ensures that even if a task stack is corrupted, the error cannot tamper with the data and peripherals of other tasks. However, Keil RTX does not offer isolation of normal user tasks and the kernel (i.e., **KMI** is unsupported), so that code and data of the kernel cannot be entirely opaque to user tasks.

B. *Case Study: Azure RTOS ThreadX*

Similar to Tizen, ThreadX is able to dynamically load modules, and each module includes several tasks and can be configured as an isolated part with others. However, the isolation is only implemented at the module level, not the task isolation. Therefore, **TMI**, **PI** and **CSG** are unsupported.