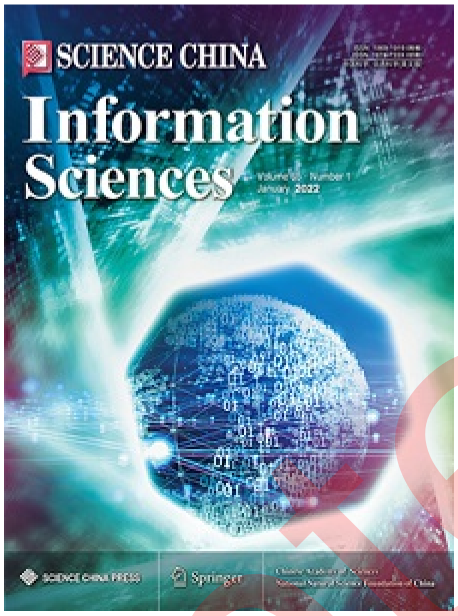


CToMP: A Cycle-task-oriented Memory Protection Scheme for Unmanned Systems

Journal:	*SCIENCE CHINA Information Sciences*
Manuscript ID	SCIS-2023-0048.R2
Manuscript Type:	Research Paper
Date Submitted by the Author:	08-Sep-2023
Complete List of Authors:	Ma, Chengyan; Xidian University School of Cyber Engineering Xi, Ning; Xidian University School of Cyber Engineering Lu, Di; Xidian University School of Computer Science and Technology Feng, Yebo; University of Oregon Ma, JianFeng; Xidian University School of Cyber Engineering
Keywords:	unmanned system, memory corruption attack, memory protection, system security, randomized memory address
Speciality:	Communication Network Security < Information & Communication Engineering

SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

CToMP: A Cycle-task-oriented Memory Protection Scheme for Unmanned Systems

Chengyan MA[1], Ning XI[1], Di LU[2*], Yebo FENG[3] & Jianfeng MA[1]

[1]*School of Cyber Engineering, Xidian University, Xi'an* 710071*, China;*
[2]*School of Computer Science and Technology, Xidian University, Xi'an* 710071*, China;*
[3]*University of Oregon, Eugene* 97403*, USA*

Abstract Memory corruption attacks (MCAs) refer to malicious behaviors of system intruders that modify the contents of a memory location to disrupt the normal operation of computing systems, causing leakage of sensitive data or perturbations to ongoing processes. Unlike general-purpose systems, unmanned systems cannot deploy complete security protection schemes, due to their limitations in size, cost and performance. MCAs in unmanned systems are particularly difficult to defend against. Furthermore, MCAs have diverse and unpredictable attack interfaces in unmanned systems, severely impacting digital and physical sectors. In this paper, we first generalize, model and taxonomize MCAs found in unmanned systems currently, laying the foundation for designing a portable and general defense approach. According to different attack mechanisms, we found that MCAs are mainly categorized into two types—*return2libc* and *return2shellcode*. To tackle *return2libc* attacks, we model the erratic operation of unmanned systems with cycles and then propose a cycle-task-oriented memory protection (CToMP) approach to protect control flows from tampering. To defend against *return2shellcode* attacks, we introduce a secure process stack with a randomized memory address by leveraging the memory pool to prevent `Shellcode` from being executed. Moreover, we discuss the mechanism by which CToMP resists the ROP attack, a novel variant of *return2libc* attacks. Finally, we implement CToMP on CUAV V5+ with Ardupilot and Crazyflie. The evaluation and security analysis results demonstrate that the proposed approach CToMP is resilient to various MCAs in unmanned systems with low footprints and system overhead.

Keywords unmanned system, memory corruption attack, memory protection, system security, randomized memory address

Citation

1 Introduction

Unmanned systems are embedded computing systems that monitor, respond to, or control an external environment through sensors, actuators, and other input/output interfaces [1]. Such systems must meet various constraints, such as timing, efficiency, security, etc., that are imposed on them by real-time behaviors of the external world they interface with. These unmanned systems wide ranging applications, such as search and rescue [2], agriculture [3], and autonomous vehicles. However, the increasing popularity of unmanned systems increases security concerns regarding them [4,5]. According to recent studies [6–8], current unmanned systems continue to possess a myriad of flaws that can be leveraged by malicious parties to launch attacks, affecting their proper operations, stealing private data of users, or even endangering public safety.

Among all the threats encountered by unmanned systems, software-oriented attacks are emerging and gradually becoming one of the most concerning. These attacks exploit software vulnerabilities within the unmanned system firmware to maliciously interfere with system operations, however, it has received attention only recently [9–11]. Particularly, **memory corruption attacks (MCAs)** [12], a special form of software-oriented attacks, are becoming increasingly rampant [13–15]. Unlike the segmented memory management in general computer systems, the user code in unmanned systems shares the same physical

* Corresponding author (email: dlu@xidian.edu.cn)

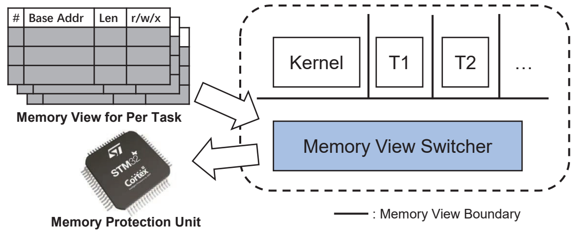


Figure 1 Architecture of MINION [14] for defending against MCAs. Here, each task has a memory view to indicate its accessible memory regions.

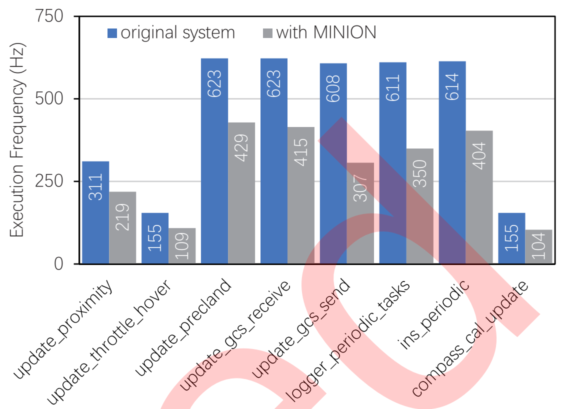


Figure 2 Effect of MINION on unmanned system performance.

memory with the kernel. This design enables the user code to directly access, invoke, or even tamper with critical kernel instructions. Furthermore, unmanned systems usually allow memory access from peripherals (e.g., ZigBee and WiFi), thereby allowing malicious parties to launch MCAs even wirelessly. For example, in UAV systems, attackers can launch MCAs to modify the return addresses of stack frames remotely, thereby seizing the control of drones through memory overflows. Recent studies [14, 16] have demonstrated the feasibility and risk of such MCAs in the real world. Therefore, designing a practical and effective methodology to defend against MCAs is vital for ensuring the reliability and stability of unmanned systems.

Multiple approaches have been proposed to tackle MCAs in unmanned systems. Their principal methodology is equipping unmanned systems with memory management mechanisms to isolate the kernel from the user code. Widely used techniques to achieve memory isolation include TrustZone [17] and Memory Protection Unit (MPU) [13, 14, 18]. Because the microcontroller units (MCUs) used by unmanned systems rarely have TrustZone, more approaches focusing on employing MPU are required to tackle MCAs (e.g., **MINION** [14]). According to multitasking and modular programming features in unmanned systems, these approaches first create a memory view for the user code of each task from the perspectives of code reachability, I/O interfaces, and in-memory data. This process can be done manually or semi-automatically using Low-Level Virtual Machine (LLVM) [19]. They then leverage MPU to securely switch memory views in task switchover, as shown in Figure 1. Although this methodology can limit MCAs to a certain extent, it has several drawbacks. First, by applying such approaches, programmers must consider memory range divisions in developments, which will confuse developers. Moreover, existing approaches can cause considerable system overhead, reducing the processing power of unmanned systems. We conducted some performance tests by implementing MINION, the most representative of these approaches, on a UAV based on CUAV V5+ hardware with firmware Ardupilot. We found that some low-priority tasks in MINION cannot reach the execution frequency in the original system because of the time and system overheads caused by the memory view switching and MPU configuration (Figure 2). This flaw is particularly fatal to unmanned systems with strict real-time requirements. For example, the execution frequency of tasks `update_gcs_send` and `ins_periodic` is reduced by half. This reduction caused a severe problem: we could not receive the Mavlink messages sent by the drone to the ground station in time, and the drone could not perceive its own acceleration and other states in real time. Eventually, the drone lost control and crashed. Moreover, such approaches only use MPU to protect the memory security of unmanned systems. However, because of the hardware performance limitation, the upper limit of the memory regions MPU can protect is 16; hence, these approaches lack extensibility in the face of complex unmanned systems.

To fill this gap, herein, we propose an effective and efficient approach called cycle-task-oriented memory protection (CToMP) to protect unmanned systems from MCAs. Unlike existing approaches that isolate memory regions for each task, CToMP treats tasks executed within one cycle as a whole and focuses on protecting a few critical codes, variables, and registers. To reduce the system overhead and ensure the timeliness of tasks, CToMP releases the pressure of memory view switching and MPU configuration by

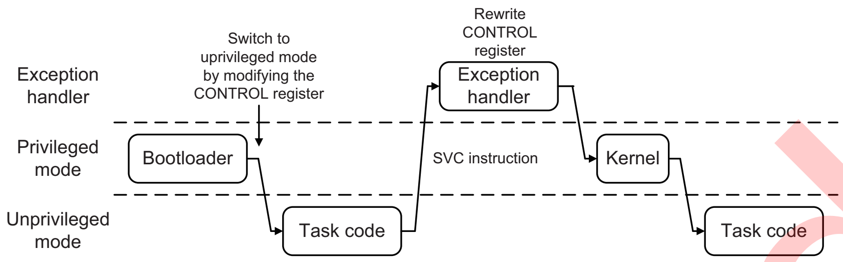


Figure 3 Execution level switching in unmanned systems. The system generally boots from privileged mode and then converts to unprivileged mode to execute tasks. If kernel instructions are required, the system will switch to privileged mode.

performing security operations and memory allocations right before starting each cycle rather than before beginning each task. To enhance the security and prevent the execution of injected `Shellcode`, CToMP dynamically assigns the process stack address and buffer addresses by randomizing memory allocations.

Compared with previously reported approaches to resist MCAs, CToMP makes the following contributions:

- To simplify the defense interface, we summarize the existing MCAs for unmanned systems and classify them into two categories (i.e., *return2libc* and *return2shellcode*) based on their execution of `Shellcode`.
- We establish a cycle-based operation model for unmanned systems as the foundation for our system design. To our knowledge, CToMP is a brand-new memory protection approach for unmanned systems. Moreover, we build a secure process stack based on randomized memory allocation to enhance system security.
- By evaluating CToMP on two unmanned platforms (CUAV V5+ with Ardupilot and Crazyflie), its effectiveness and efficiency are verified. Our approach defends against MCAs without compromising the real-time performance of unmanned systems.

The remainder of this paper is organized as follows. After describing the security model and motivations in §2, we present the design of CToMP in §3. Further, we evaluate CToMP in §4. Finally, we describe related work in §5 and conclude this paper in §6.

2 Security Model & Motivation

In this section, we elaborate on necessary background knowledge to help readers understand our system design. Later, to simplify the defense interface, we summarize and taxonomize existing MCAs against unmanned systems, classifying them into two categories (i.e., *return2libc* and *return2shellcode*). Finally, we demonstrate the motivations of our proposed approach.

2.1 Background

As for power consumption, hardware platforms of unmanned systems are still dominated by low-cost MCUs, such as STM32 series based on the ARM Cortex-M architecture. To ensure the availability of systems, the development of unmanned systems focuses more on realizing more applications with limited software and hardware resources rather than security protection that may take up more resources. However, these low-cost MCUs are also designed with security features, which have not been taken seriously. We will briefly introduce these features.

2.1.1 *Execution Levels*

Privileged and unprivileged modes are two execution levels of MCUs [20]. In privileged mode, the code can invoke all available instructions and access all resources (e.g., system registers, memory, and peripherals). In unprivileged mode, the code only has limited access to some resources. Any access to unpermitted registers, including MPU, SysTick timer, and Nested vectored interrupt controller (NVIC), will raise hardware exceptions. Therefore, the kernel can be placed in privileged mode, and user code related to the unmanned system task implementation can be placed in unprivileged mode.

The `CONTROL` register determines whether the code executes in privileged or unprivileged mode. Only in privileged mode can the code modify the `CONTROL` register, thereby changing the execution level to

Table 1 Summary of execution levels and stack use options.

Execution level	Used to execute	Stack used
Privileged	Kernel Exception handlers	Main stack
Unprivileged	Tasks	Process stack

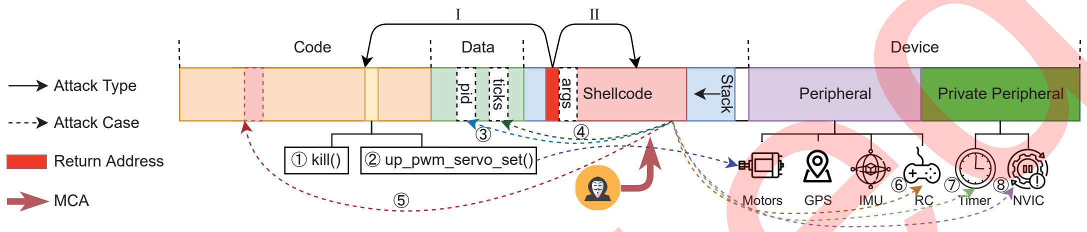


Figure 4 MCAs against unmanned systems are generally implemented by executing `Shellcode`, and their targets are distributed in the code, data and device areas of memory.

Table 2 Attack cases and their attack targets. We divide these attack cases into two types (i.e., *return2libc* and *return2shellcode*) according to the different implementation methods.

No.	Attack Case	Attack Type	Attack Target
①	Process Termination	return2libc	Reusing `kill()` function in the **Code** area.
②	Servo Operation	return2libc	Invoking `up_pwm_servo_set()` in the **Code** area with two abnormal args (`channel` and `value`) to change the speed of motors.
③	Control Parameter Attack	return2shellcode	Overwriting the PID parameters in the **Data** area.
④	Soft Timer Attack	return2shellcode	Overwriting `ticks` and `last_run` two count parameters in the **Data** area.
⑤	Memory Remapping	return2shellcode	Copying the malicious code to **FLASH** and replacing the existing function in the **Code** area.
⑥	RC Disturbance	return2shellcode	Modifying RC-related registers in the **Peripheral** area.
⑦	Hard Timer Attack	return2shellcode	Reloading the system timer value of `SYST_RVR` in the **Private Peripheral** area.
⑧	Interrupt Vector Overriding	return2shellcode	Overriding `NVIC` registers in the **Private Peripheral** area.

unprivileged mode. Correspondingly, in unprivileged mode, the code cannot change the execution level by directly modifying the `CONTROL` register. Instead, it must call the `SVC` instruction to switch the system to a handler mode, in which the code can change to privileged mode, as shown in Figure 3.

2.1.2 *Stack Pointer*

Another key point is that the code will use a separate descending stack in privileged and unprivileged mode, respectively. Consequently, MCUs must implement two stacks, the *main stack* and the *process stack*. Meanwhile, each stack has an independent stack pointer holding the address of the last stacked item in memory; we call them the `MSP` and `PSP`. When the code needs to invoke kernel operations, it typically switches to the *main stack*, and the tasks always use the *process stack*. We summarize these features in Table 1.

2.1.3 *Memory Protection Unit*

Cortex-M is the most commonly used ARM processor for embedded unmanned systems. MPU is a security kernel feature of the Cortex-M series MCUs [21]. It can set the properties and access permissions of different memory addresses by dividing the memory map into several regions, such as whether a certain address range is allowed to be executed, read, or written. Furthermore, MPU can isolate system resources and code by limiting access permissions in privileged and unprivileged modes. If the code accesses a memory region that MPU protects without permission, the processor will throw a fault exception, which can prevent illegal memory access.

2.2 Security Model

In MCAs attackers use memory vulnerabilities to inject `Shellcode` into the stack to achieve various malicious operations. As shown in Figure 4, `Shellcode` can be executed in two ways. Type I involves overwriting the return address in the stack to the existing function address in memory so that the device can perform some operations that are not allowed or illogical; it is called *return2libc*. Type II involves overwriting the return address in the stack with the starting address of `Shellcode` to execute the malicious behavior customized by the attacker. We name it *return2shellcode*. Additionally, MCAs for embedded unmanned systems are subdivided into several cases in Table 2 according to different `Shellcode` attack targets. Notably, as a variant of *return2libc* attack, the ROP attack [22, 23] has been widely discussed on x86 architecture systems. This attack executes instruction sequences, called *gadgets*, in the existing code area through the stack overflow vulnerability, and each gadget ends with a `ret` instruction. By chaining these gadgets together with `Shellcode`, an attacker can modify registers or some memory data. In the unmanned system scenario, the ROP attack is similar to attack case ② in Table 2.

Because of hardware and performance limitations, the existing embedded development platforms do not have memory isolation or address randomization and other security protection measures that can effectively avoid MCAs. Therefore, we believe that MCAs are ubiquitous in embedded unmanned systems. We assume the attackers have the following abilities to find and exploit memory vulnerabilities to perform attacks.

- Attackers can use Fuzzing [24–26] and other testing methods to discover memory vulnerabilities, such as stack overflow in the communication protocol encoding/decoding of the system remote control.
- Attackers can use static analysis tools [27–30] to reverse the firmware in embedded hardware and obtain the memory addresses of sensitive parameters, such as the PID parameters.
- Attackers can discover which key registers are accessed during the sensor reading and writing process through firmware Re-Hosting [31–34].

Because `Shellcode` must be injected from outside the system, the vulnerabilities that can be easily exploited by attackers lie in the user code interacting with the outside rather than in the kernel responsible for interrupt management, task scheduling and other internal features. After discovering the vulnerabilities, attackers write `ShellCode` and tamper with the return address of the stack frame to achieve MCAs. Because all codes in an MCU share the same physical memory, an attacker who launches an attack from the user code can modify key parameters in the memory by injecting malicious instructions. In addition to attackers, ordinary users may also modify key parameters in the kernel because of misoperation, such as setting unreasonable PID parameters through MAVLink commands [35], which can also have disastrous consequences.

2.3 Finding of Drawbacks in Relevant Works

At present, the most effective solution to MCAs is to implement memory isolation between the kernel space and user code, which limits the range that malicious code can access in memory [13, 14, 18, 36]. These relevant works achieve memory isolation by building a memory view for each task, just like running a task in a **Sandbox**, and we summarize them as **task-oriented** solutions. Task-oriented memory isolation requires configuring MPU registers for the next task after one task ends, which is called memory view switching. Since MPU registers are accessible only in privileged mode, memory view switching must enter an exception handler and switch mode while the task runs in unprivileged mode. We summarize two stages in memory view switching as follows:

First, before task application execution, the following steps are required:

(1) The scheduler reads the task information list and decides which task must be executed next.

(2) The system initializes the process stack required by the task code and switches the memory view to configure the memory regions that the task can access by MPU.

(3) The system switches from the main stack to the process stack.

(4) The system switches the execution level from privileged mode to unprivileged mode.

Second, to switch from unprivileged mode to privileged mode after task completion, the following steps are required:

(1) The system enters `SVC` interrupt through a system call, and switches from unprivileged mode to privileged mode.

(2) Then, the system switches the stack pointer from `PSP` to `MSP`.

Table 3 Time cost of each step in the execution level and memory view switching. Since each task must configure different numbers of accessible memory regions, the time spent by MPU also differs.

Stage	MPU Configure	Stack Initialize and Switch
Time (μsec)	9-15	10
Stage	SVC System Call	Execution Level Switch
Time (μsec)	1	1

Through experiments, we also sorted out the time spent in each step, as shown in Table 3. However, when we applied this task-oriented solution to *Ardupilot*, an open-source unmanned system firmware, efficiency problems emerged. In the search for the cause, we found that although the task scheduling of unmanned systems is uncertain and irregular in the traditional concept, we can use a cycle-based model to describe it. Unmanned systems perform a certain number of tasks during each cycle. We will describe this model in detail in §3.1. Since at least seven tasks need executing in one cycle in Ardupilot, it takes at least 154μsec for frequent memory view switching between tasks. Consequently, some low-priority tasks cannot obtain the time in a cycle to execute, which affects unmanned system availability. Therefore, we consider whether we can use MPU to manage the memory access range of a cycle, which is similar to putting all tasks in a cycle into a sandbox rather than putting one task into one sandbox. This approach reduces the frequency of memory view switching, thereby improving system efficiency.

However, we need to prevent attackers from attacking tasks in the same cycle. When we simultaneously manage several tasks in a cycle with MPU, tasks do not securely access the code and data from each other because they share the same region of memory. We determine the static loading of the code as the reason why embedded unmanned systems are vulnerable to MCAs. This attribute allows attackers to easily analyze the call stack and memory addresses and inject `Shellcode` to cause damage. Therefore, we hope to implement random memory allocation, which is not widely used in embedded unmanned systems because of the lack of the Memory Management Unit (MMU), and improve the difficulty of memory analysis, so as to achieve the purpose of protecting user code and data in the same cycle.

3 Design

In this section, we elaborate on the design of CToMP. According to the use characteristics and requirements of unmanned systems, we have determined the following design goals:

G1 Security. Our design needs to be resistant to MCAs.

G2 Efficiency. After our design is added to unmanned systems, it only brings a minimum runtime overhead, and cannot affect the real-time requirement of task execution.

G3 Extensibility. Our design can adapt to different functional requirements in different scenarios rather than being fixed.

G4 Low Footprint. Due to the limitation of memory size in unmanned systems, our design cannot occupy too much memory space.

G5 Generality. Our design can be easily adapted to different unmanned devices without much additional development work.

Following these design goals, We first model the operation of unmanned systems in §3.1. Then, we introduce the overall system architecture in §3.2. In addition, we describe the structure of the secure process stack in §3.3 and the workflow of CToMP in §3.4.

3.1 Cycle-based Model of Unmanned Systems

Unmanned systems encapsulate simple functions such as sensor reading/writing and actuator execution into multiple tasks, and realize more complex remote control or autonomous operations through the cooperation between tasks. In the traditional concept, the unmanned system, as a real-time system, requires each task to be completed at a certain time, and these tasks are triggered uncertainly by the external environment. For example, we cannot predict when the unmanned system will encounter an

Table 4 RTOSes used by five kinds of unmanned system firmware, and their application scenarios.

Firmware	RTOS	Number of Tasks	Application Scenarios
Ardupilot	ChibiOS	49	UAVs, Rovers, Submarines, ...
PX4	Nuttx	27	Drones, VTOLs, Rovers, ...
FMT	RT-Thread	6	UAVs, Cars, Robots, ...
Paparazzi	ChibiOS	9	Rotorcrafts, Hybrids, Boats, ...
Crazyflie	FreeRTOS	37	Drones

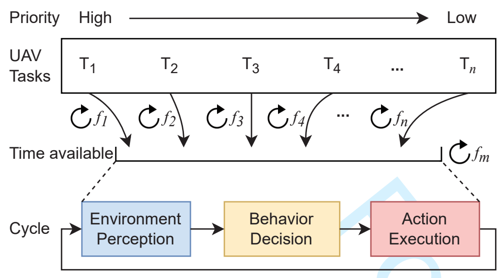


Figure 5 Cycle-based operation model of unmanned systems.

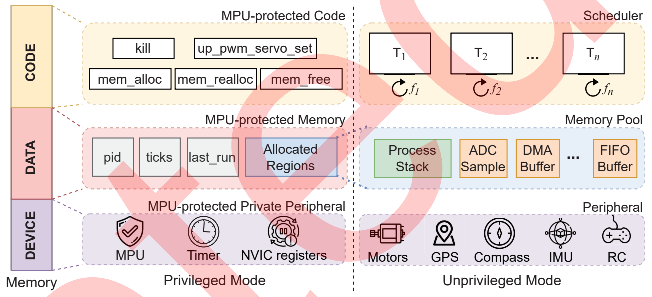


Figure 6 Overall architecture of CToMP.

obstacle and change its trajectory. Table 4 summarizes the results of our research on various unmanned system firmware applicable to drones, rovers, and boats. However, we found that although the number of tasks in these firmware varies, they all rely on the real-time operating system (RTOS) to implement task scheduling. Further, the scheduling algorithms in these RTOSes are composed of preemptive scheduling and round robin. This means, in practical design, unmanned systems only need to implement three stages of functions—**environment perception**, **behavior decision** and **action execution**, which are supported by tasks within a specified time period to meet the real-time requirement, and we call this time period a cycle. As shown in Figure 5, we regularize seemingly uncertain and irregular task executions in unmanned systems with cycle-based modeling, thereby laying the foundation for our system design. In detail, f_m in Figure 5 represents the cycle frequency, that is, how many cycles will be experienced in one second, and the longest available duration of one cycle is $1/f_m$ seconds. Then for task i, its frequency f_i is determined by the average cycle interval (ACI) at which it can be scheduled:

$$f_i = f_m/ACI_i \tag{1}$$

For example, in an unmanned system with a cycle frequency of 400Hz, a task is executed every two cycles on average, then the frequency of this task is 200Hz. What's more, each task is given a priority. In a cycle, high-priority tasks must be scheduled first, while low-priority tasks will use the remaining time as needed after the execution of high-priority tasks. This enables each task to meet its functional requirements as much as possible. Therefore, it can be obtained that the less time a cycle takes, the faster the cycle frequency becomes, and each task can be scheduled in a more timely manner, so the real-time requirement of the unmanned system is guaranteed. Conversely, if the single cycle time is long, the system reliability will be reduced.

3.2 System Architecture

The main goal of our system design is to prevent attackers from exploiting victim tasks with memory vulnerabilities to access kernel code or modify memory data. Specifically, our approach should be able to defend against each attack case in §2.2 effectively. As illustrated in Figure 6, in our system architecture, the code segment, data segment, and device address segment in the memory are separated by MPU into two environments: privileged mode and unprivileged mode. According to the assumptions in §2.2, as user code for all tasks is executed in unprivileged mode, the `Shellcode` injected by an attacker to exploit memory vulnerabilities usually occurs in unprivileged mode. Unlike relevant task-oriented approaches that require designing a memory view for each task and attack to defend against MCAs, our solution focuses on blocking two attack paths of `Shellcode` to secure the unmanned systems. We explain how this architecture meets the design goal **G1** *Security* as follows.

Defense against *return2libc*: Since functions located in privileged mode cannot be directly called by user code running in unprivileged mode, for attack cases ① and ②, we place two functions `kill` and `up_pwm_servo_set` in privileged mode. The `kill` is a kernel function used by RTOS in unmanned systems to terminate the task. The `up_pwm_servo_set` is a function used to control the execution of actuators in unmanned systems. Malicious calls to both of these functions can have catastrophic consequences, such as the crash of drones and changing the driving path of unmanned vehicles. Therefore we configure them with MPU not to be accessible in unprivileged mode to prevent `Shellcode` written by attackers from maliciously jumping to their function addresses.

Defense against ROP: According to the description in §2.2, the ROP attack achieves the goal of executing malicious code by exploiting gadgets of existing code and concatenating them. But as we explained in defending against *return2libc* attacks, the code containing sensitive parameters has been protected by MPU, so attackers cannot jump to gadgets that can modify these parameters through `Shellcode`. Although attackers can still exploit gadgets located in unprivileged mode, these gadgets cannot access registers and data protected by MPU; hence, our architecture can still effectively resist the ROP attack.

Defense against *return2shellcode*: In this attack way, the attacker can execute more malicious instructions in `Shellcode` and destroy more memory data. Since all code in the MCU shares the same physical memory, in the absence of memory isolation, the attacker's targets include, but are not limited to, the PID parameters that control the unmanned system attitude, the count parameters responsible for task scheduling, and various peripherals such as sensors and system Timer. Here we use two solutions to protect them.

First, for attack cases ③, ⑦, and ⑧, the data and registers in these attack objects should be configured and fixed in memory when the unmanned system is initialized. Therefore, we only need to configure these parameters and registers to be unmodifiable in unprivileged mode with MPU after they are set. They can be effectively protected from being accessed by the attacker's `Shellcode`. At the same time, we configure the entire code segment to be non-writable with MPU, so that attack case ⑤ cannot overwrite the existing code with malicious instructions through `Shellcode`.

For attack case ④, `ticks` and `last_run`, these two count parameters are used for task scheduling, and they are changed in each cycle to ensure that the execution of each task can meet the functional requirements. Parameter `ticks` is the number of cycles that have passed, `last_run` is an array used to calculate the interval between the last executed cycle of each task and the current cycle. Therefore, we set the two parameters to be modifiable only in privileged mode. When the tasks of a cycle are executed, the system switches the execution level once, and then updates these two parameters after entering privileged mode, which can satisfy the task scheduling and parameter security.

Finally, for some code and data that must share the memory with other tasks, such as reading remote control variables from registers that can only run in unprivileged mode (attack case ⑥), we design another solution to protect their security. By analyzing the execution condition of *return2shellcode*, we found that the attacker needs to obtain the starting address of injected `Shellcode` to jump to the malicious code execution area by tampering with the return address in the stack. Due to the static loading code of MCUs, it is easy for an attacker to analyze the required memory address and implement malicious behavior. However, we noticed that using the two instructions `__set_PSP` and `__set_CONTROL(SP_PROCESS)` can specify an area in memory as the process stack used by tasks. If we randomize the stack in each cycle, it will be much more difficult for an attacker to analyze the starting address of `Shellcode`, which can effectively prevent the execution of `Shellcode`. Therefore, we design a **memory pool** in unprivileged mode to dynamically allocate a random address area for the stack used in each cycle.

In summary, CToMP can resist all attack cases and meet the design goal **G1** *Security*. At the same time, different from task-oriented solutions, we manage the memory access range of tasks in a cycle as a whole, so that the MPU configuration and execution level switching do not need to be performed between tasks, but only before the start of a cycle, so as to satisfy the design goal **G2** *Efficiency*. In addition, since MPU in MCUs can protect up to 16 memory regions, it is difficult for solutions that only rely on MPU to play a protective role when the functions of unmanned systems are gradually complex. Our solution can effectively prevent the execution of `Shellcode` by randomizing the process stack, and is not limited by the number of MPU-protected memory regions and other hardware resources, which meets the design goal **G3** *Extensibility*. What's more, we only add a memory pool area into the system architecture to

Algorithm 1 Allocation and release of memory regions for the secure process stack. Here, r_i is a unit in the structure array R.

```
1: function MEMALLOC($size$)
2:     if $allocated\_regions\_num$ >MAX_ALLOCATE_NUM then
3:         return NULL
4:     end if
5:     $start\_addr \leftarrow TRNG()$
6:     $end\_addr \leftarrow start\_addr + size - 1$
7:     for $i = 0 \rightarrow allocated\_regions\_num$ do
8:         if MAX($r_i.start\_addr, start\_addr$) < MIN($r_i.start\_addr + r_i.size - 1, end\_addr$) then
9:             return mem_realloc(size)
10:        end if
11:    end for
12:    $R \leftarrow R \cup$ New Region
13:    $allocated\_regions\_num++$
14:    return $start\_addr$
15: end function
16: function MEMREALLOC($size$)
17:    if $realloc\_times > 3$ then                                  ▷ Retry times can be set.
18:        return NULL
19:    else
20:        return mem_alloc(size)
21:    end if
22: end function
23: function MEMFREE($pointer$)
24:    for $i = 0 \rightarrow allocated\_regions\_num$ do
25:        if $r_i.pointer = pointer$ then
26:            Delete $r_i$ from $R$
27:            $allocated\_regions\_num--$
28:            return
29:        end if
30:    end for
31: end function
```

achieve dynamic memory allocation, and we will describe how the memory pool satisfies the design goal **G4** *Low Footprint* in §3.3. Last but not least, MPU, execution levels and the process stack are features supported by most unmanned system MCUs, so our system architecture can be easily adapted to other unmanned system firmware and RTOSes, and our design realizes the **G5** *Generality*.

3.3 Secure Process Stack

Unfortunately, due to the lack of MMU support, MCUs for unmanned systems do not support dynamic memory allocation, or can only allocate memory blocks with fixed addresses, which does not meet our needs for randomizing the process stack. For example, only three fixed regions are provided for memory allocation to choose from in ChibiOS, which supports the unmanned system firmware Ardupilot. In order to be compatible with the existing memory allocation solutions in unmanned systems, we designate an area in memory as a memory pool. We use the following structure array $R = \{r_0, r_1, \ldots, r_n\}$ to denote the usage of the memory pool.

```c
typedef struct allocated_region {
    void *pointer;
    uint32_t start_address;
    uint32_t size;
}allocated_region[MAX_ALLOCATE_NUM];
```

In this structure, we define a pointer to the allocated memory region, the start addresses of the region and the allocation size. At the same time, we also need to specify the maximum number of memory regions that can be allocated by defining the value of MAX_ALLOCATE_NUM. Three functions are designed to allocate, re-allocate and free memory:

```c
void *mem_alloc(uint32_t size);
void *mem_realloc(uint32_t size);
void mem_free(void *pointer);
```

As described in Algorithm 1, `mem_alloc` uses the True Random Number Generator (TRNG) [37] supported by ARM Cortex-M series MCUs to generate a start address of the memory area to be allocated. Then, when the number of allocated regions does not reach the maximum setting, the allocator needs to traverse all memory regions in the structure array `allocated_region` to check whether there is a conflict

Table 5 The five memory regions that need to be dynamically allocated in the unmanned system firmware Ardupilot, and their sizes.

Name	ADC sample	DMA buffer	RX bounce buffer
Size (Byte)	144	304	64
Name	TX bounce buffer	FIFO buffer	Process stack
Size (Byte)	64	112	1024

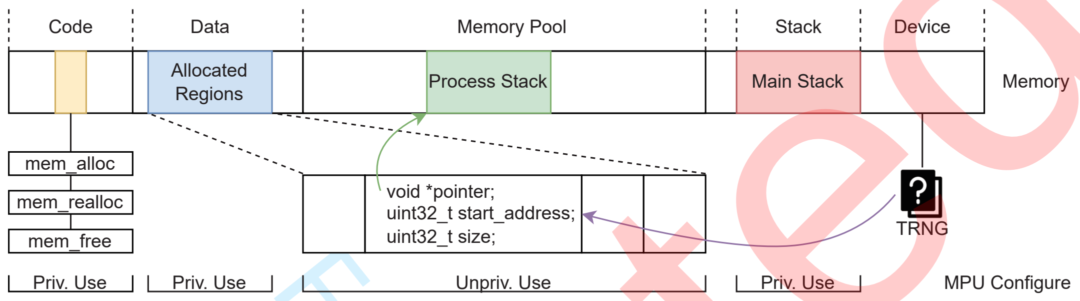


Figure 7 Secure process stack with randomised addresses. We used MPU to configure the permissions of each module in randomized memory allocation.

between the to-be-allocated region and the allocated regions. If there is no conflict, the allocator should save the region information into the structure array. Otherwise, it will find the allocatable region again by `mem_realloc`. To free the allocated memory, it just needs to call `mem_free` to delete the information of the pointer in the structure array.

In order to measure the space occupancy of the memory pool, we summarize the stack and buffers that use memory allocation and the size of regions they require in Table 5. It can be seen only five buffers and one stack need to dynamically allocate memory; the value of `MAX_ALLOCATE_NUM` should be set to 6, which means that the space occupied by the structure array `allocated_region` is 72 bytes. At the same time, the total memory space required for these buffers and the process stack is 1712 bytes. In order to ensure the randomness of the process stack space address, we set the size of the memory pool to 5632 bytes, which is about three times the size of the total required space. Since most MCUs have 1MB-2MB of Flash ROM and 192KB-512KB of SRAM, the memory pool we designed has a very small footprint and can meet the design goal **G4**.

On the other hand, in order to secure the memory allocation, we need to configure the three functions and the struct array `allocated_region` to be accessible only in privileged mode with MPU. Only the memory pool can be accessed by tasks in unprivileged mode. Due to the real-time requirement of unmanned systems, the data in these buffers that need to dynamically allocate memory are only valid for one cycle; that is, they will not be available in the next cycle. So we allocate memory for them at the beginning of a cycle and release them at the end of the cycle, which can effectively meet their usage requirements. This also means that we do not need to perform frequent execution level switching in order to securely allocate memory by MPU. As illustrated in Figure 7, we implement a secure process stack to resist `Shellcode` execution.

3.4 Workflow of CToMP

Our approach works upon cycle-based task modeling, where three stages of functions (i.e., environment perception, behavior decision, and action execution in one cycle) are embedded within. The workflow of CToMP corresponds to the three stages of functions in one cycle.

First, use MPU to configure hardware and software resources that can be accessed in one cycle. Then, a region is allocated in the memory pool for tasks of this cycle as a secure process stack. At the same time, we also need to allocate regions in the memory pool for other buffers. Next, we use the `__set_CONTROL` instruction to make the stack pointer point to the secure process stack and switch the execution level to unprivileged mode.

In unprivileged mode, each task is scheduled on demand based on the priority and two soft timing parameters (`ticks` and `last_run`) that are read-only in this mode. At this stage, the unmanned system uses various sensors and communication equipment to complete the perception of the surrounding

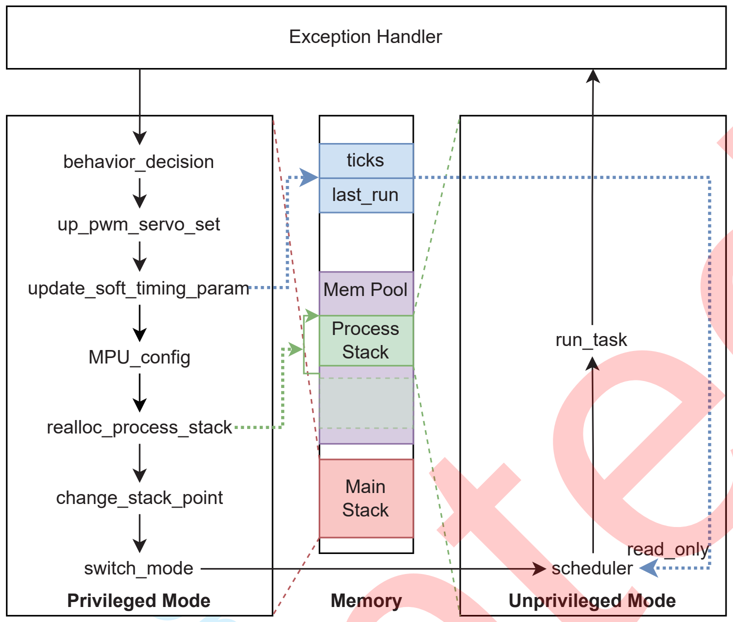


Figure 8 CToMP exection flow.

environment and its own state and receive control commands.

After tasks in a cycle are executed, the unmanned system needs to switch back to privileged mode to call some key functions protected by MPU, implement behavior decision and action execution, and allocate memory resources for the next cycle. Therefore, the user code is required to call the `SVC` instruction into the exception handler. In exception handling, switch the execution level back to privileged mode, and replace the running stack with the main stack after exiting the exception. In addition, we can also update soft timing parameters in this mode. We describe the above workflow with Figure 8.

According to the unmanned system model in §3.1, we conclude that one of the ways to ensure the real-time performance of unmanned systems is to shorten the running time of a cycle. Therefore, we clarify the advantages of CToMP in the design goal **G2** *Efficiency* by analyzing the computation cost of CToMP in one cycle and comparing it with the task-oriented approach. We assume that n tasks need to be executed in one cycle, and the task-oriented memory protection method uses MPU to limit m_i accessible areas for each task to prevent damage attacks caused by memory leaks. At the same time, since the tasks are all executed in unprivileged mode, and MPU can only be configured in privileged mode, each time a task is executed, it needs to go through the mode switch twice to configure MPU and stack for the next task before return to unprivileged mode. Therefore, in a task-oriented manner, the computation cost in one cycle is:

$$Time_{task-oriented} = \sum_{i=1}^{n} m_i T_{MPU} + n(2T_{switch} + T_{Stack} + T_{SVC}) \tag{2}$$

In Eq. (2), T_{MPU} denotes the time cost for MPU to configure a memory area, T_{Stack} denotes the time cost of the process stack initialization, T_{SVC} is the time consumed in one SVC call, and T_{switch} is the time cost of one mode switch. These indicators' values can be found in Table 3.

From the workflow of CToMP in Figure 8, it can be seen that since we regard all tasks in one cycle as a whole, CToMP mode only switches twice in one cycle. That is, switching to the unprivileged mode when entering a new cycle. After the cycle ends, switching to privileged mode to process sensitive data with an SVC call. What's more, due to only protecting key code and data, the memory regions configured by MPU pre-cycle are also fewer. We suppose these memory regions are z blocks. The computation cost of CToMP is as follows:

$$Time_{CToMP} = zT_{MPU} + 2T_{switch} + T_{Stack} + T_{SVC} \tag{3}$$

By comparing Eq. (2) and (3), we can obtain that the computation cost of CToMP in one cycle is much smaller than that of the task-oriented scheme, so CToMP meets design goal **G2**. More discussion on performance evaluation will be detailed in §4.2.2.

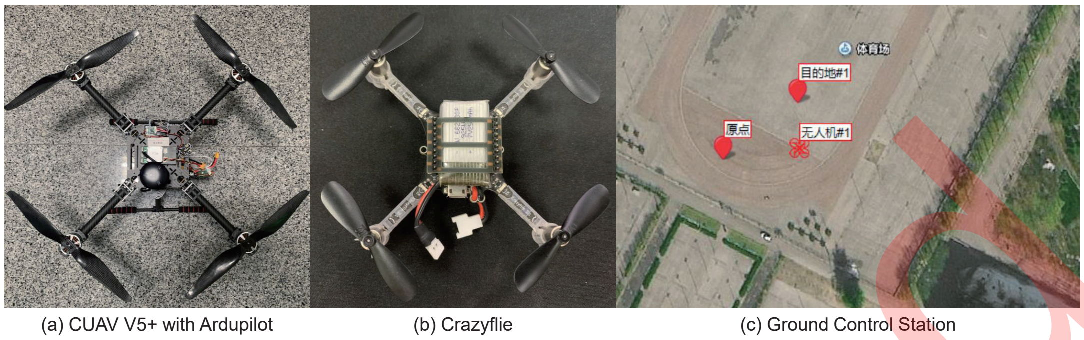


Figure 9 Experimental platforms and our ground control station. (a) CUAV V5+ with Ardupilot; (b) Crazyflie; (c) Ground Control Station.

4 Evaluation

In this section, we first introduce the implementation and configuration details. We then evaluate the proposed approach from the following perspectives:

- How effectively CToMP can compete against memory corruption attacks?
- What is the performance impact of CToMP on the unmanned system?

4.1 Implementation & Configuration

We implement CToMP on CUAV V5+, a typical unmanned aerial vehicle that is full compatibility with the Pixhawk project FMUv5 design standard. CUAV V5+ is equipped with an ARM 32-bit Cortex-M7 processor, a 512KB SRAM and a 2MB Flash memory where the code and data are stored. Similar to most unmanned systems designed based on the Pixhawk standard, CUAV V5+ supports the open source firmware, Ardupilot, which uses ChibiOS as the real-time operating system. We also chose an open source drone, *Crazyflie*, which is equipped with an ARM 32-bit Cortex-M4 CPU, a 196KB SRAM and a 1MB Flash memory and uses FreeRTOS as the operating system. We use Crazyflie as a supplementary experiment to verify the generality of our solution. Figure 9 shows our experiment platforms and a simple ground control station (GCS) to record the flight status of our UAVs so that we can analyze the experimental results.

4.2 Study Case: CUAV V5+ with Ardupilot

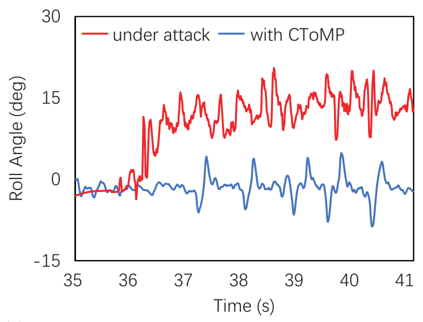
4.2.1 *Security Analysis*

Since unmanned systems are closely related to the physical world, attacks against them tend to cause damage and loss in the real world. Similarly, MCAs in Ardupilot mainly target the key code and data stored in the memory that can affect the stable flight of drones [7]. In the following, we describe several representative attacks in detail.

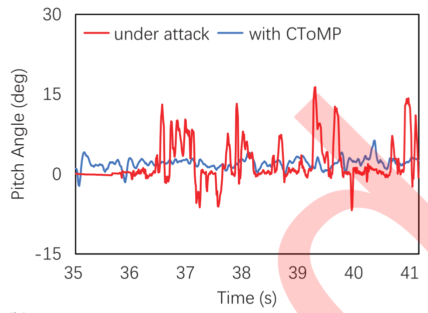
Process Termination. ChibiOS, the real-time operating system in Ardupilot, provides some POSIX-like interfaces. The `kill` function can directly terminate the execution of tasks. We modified the return address in the stack used by the vulnerable user code, and terminated `fast_loop`, the core task of Ardupilot, by calling `kill` through *return2libc*. However, under the protection of CToMP , `kill` cannot be called outside of privileged mode, which makes it impossible for attackers to use the process stack in unprivileged mode to perform malicious jumps.

Servo Operation. The function of `up_pwm_servo_set` is to output the PWM waveform to control the motor speed of drones. Similar to the previous attack, this attack changed the return address to `up_pwm_servo_set` address, and passed two illegal parameters (`channel` and `value`) to this function, which caused motors rotating abnormally. Same as the previous one, CToMP makes `up_pwm_servo_set` inaccessible in unprivileged mode, thus preventing the attack.

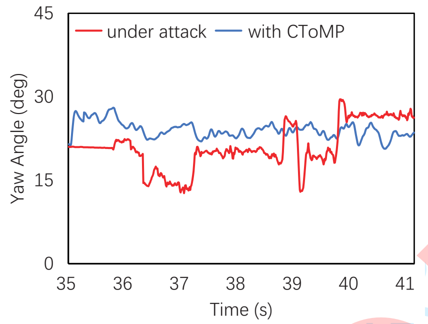
Control Parameter Attack. PID controller is the most widely used automatic controller. It has the advantages of simple principle, easy implementation, and wide application. There are three important parameters `pid_rate_roll`, `pid_rate_pitch`, `pid_rate_yaw` in Ardupilot, which respectively control the roll, pitch and yaw angle of the drone aircraft attitude. After the aircraft debugging is completed, these three parameters will be fixed and written into the memory and will not change; slight changes in



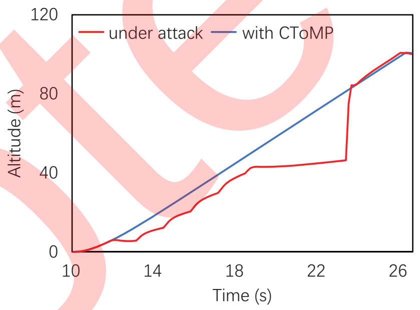
(a) Roll angle values under control parameter attack



(b) Pitch angle values under control parameter attack



(c) Yaw angle values under control parameter attack



(d) Altitude values under soft timer Attack

Figure 10 Under different attacks, protected system sensor output compared to the unprotected system.

parameters will affect the stability of flight. In our experiment, the attack overwrote the PID control parameters by injecting malicious code while the drone was hovering in the air. The drone began to sway back and forth, left and right, gradually deviating from the original position. In our architecture, we configure these three parameters to be **read-only** using MPU, protecting them from tampering, then the attack will no longer work. Figure 10(a) - Figure 10(c) record the data of three angles output by IMU. It is obvious that under the protection of CToMP , the output of IMU is more stable.

Soft Timer Attack. The scheduler is an important component of unmanned systems, and it is responsible for the on-demand execution of various functional tasks. We interfered with the execution frequency of task `update_altitude` by modifying the two soft timer parameters `ticks` and `last_run` in the scheduler. Figure 10(d) shows the flight altitude recorded by the GCS in both cases when the drone is attacked and protected by CToMP . It can be seen when an attack occurs, although we increased the remote control (RC) throttle to make the drone start rising, the change of altitude is not reflected on the GCS in real-time, which undoubtedly interferes with the driver's control of the drone, bringing a certain risk. We effectively avoid this attack by placing scheduling-related data in privileged mode that malicious code cannot access.

Memory Remapping. ARM Cortex-M series MCUs allow users to perform a hot-patching operation. This means the new program can be written into the Flash ROM, not through `JTAG` or `USART0`, but through `USB`, `RS232`, wireless transmission and other interfaces that are still preserved in the external environment after hardware packaging. The attacker can replace the existing function in Flash with the malicious code in `Shellcode` from the vulnerable user code, and the work of the program update should only be the responsibility of **Bootloader**. Therefore, in our architecture, after Bootloader starts the system, the entire code segment is configured as unwritable by MPU, making this attack ineffective.

RC Disturbance. Most tasks of unmanned systems are to sense the surrounding environment and receive control commands. In order to isolate from the kernel, we put the user code of these tasks into unprivileged mode. For the cause of facilitating the management of these sensors and communication devices, ARM Cortex-M series MCUs provide **MMIO** technology, which can map the registers of these devices to the peripheral area in memory. This allows tasks to mutually access data from other devices, which facilitates sensor data fusion, but attackers can exploit vulnerable code to attack some sensors across tasks. For example, by modifying the channel value of RC registers in `Shellcode`, the attacker

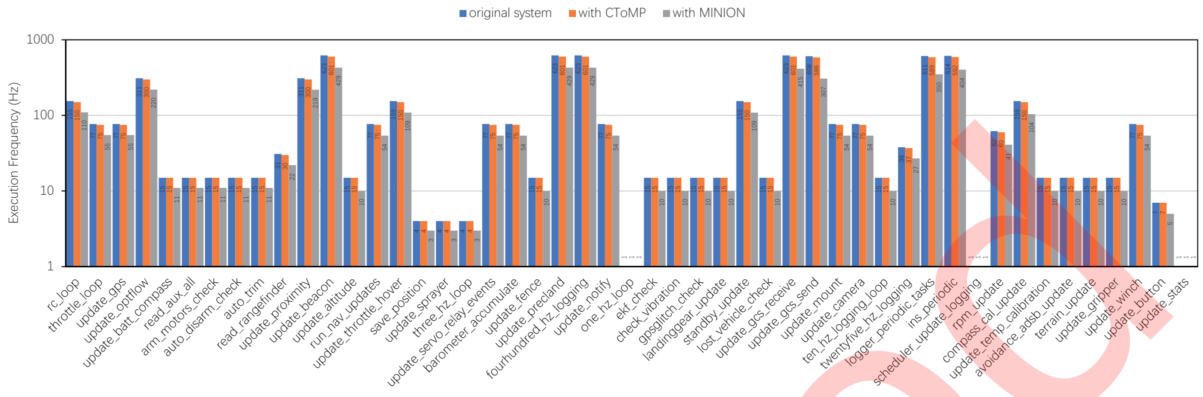


Figure 11 Log-based the execution frequency of tasks with CToMP and with MINION and baseline in original system Ardupilot.

can even take control of the unmanned system. In our architecture, due to the support of the randomized process stack, it is very difficult for an attacker to find the starting address of `Shellcode` injected through the vulnerable code, and cannot perform malicious operations.

Hard Timer Attack. All ARM Cortex-M series MCUs have a 24-bit system timer, **SysTick**. This timer counts down from the reload value (`SYST_RVR`) to zero, providing the unmanned system with a microsecond-accurate system clock. Through this clock, the unmanned system can count the execution time of tasks in each cycle and schedule tasks efficiently. In the absence of memory isolation, an attacker can slow down the system time by overwriting the value of `SYST_RVR`, which will seriously affect the task scheduling of unmanned systems. Since SysTick is a private peripheral in MCUs, its `SYST_RVR` can only be reloaded in privileged mode. In our architecture, vulnerable user code in unprivileged mode cannot be exploited to attack the hard timer.

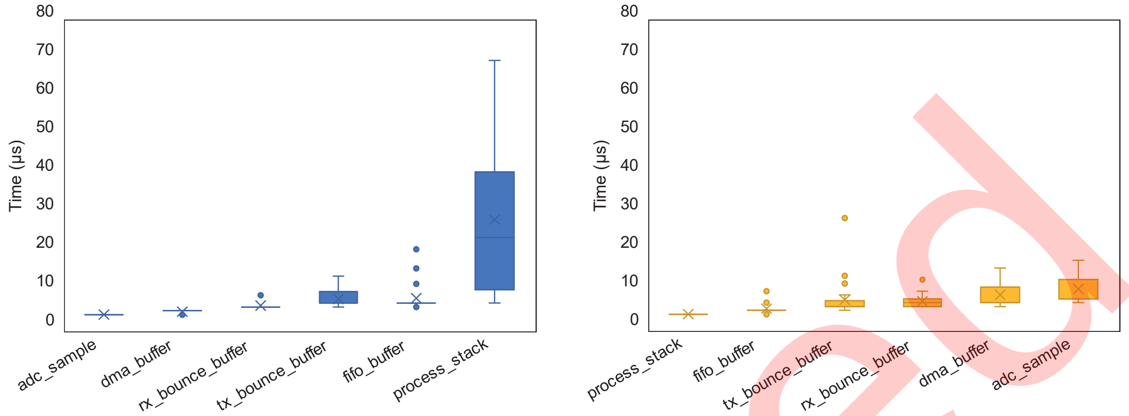
Interrupt Vector Overriding. The **NVIC** supports 1 to 240 interrupts for each task in unmanned systems. Tasks can be configured with a priority in the range of 0-255, and this information is stored in a vector table. By replacing the priorities in the original vector table and increasing the priority of some tasks, the attack can make some unimportant tasks always be called in each cycle, which will inevitably waste software and hardware resources. However, like the hard timer attack, the NVIC is also a private peripheral, so this attack cannot be implemented in our architecture.

4.2.2 *Performance Evaluation*

In order to measure the impact of new security features on the real-time performance of unmanned systems, some relevant works in recent years are based on the time of task execution. They believe that as long as tasks are completed within the maximum executable time, it will not affect the real-time performance of unmanned systems. However, according to our analysis of some existing unmanned system firmware, we found that the initial-setup maximum executable time of these tasks is inaccurate, and they will change dynamically during the operation of unmanned systems to improve the scheduling efficiency. This is why we found that the relevant works have problems with some complex unmanned systems, such as Ardupilot. The maximum executable time is not an accurate measure of whether the functionality of the task is affected.

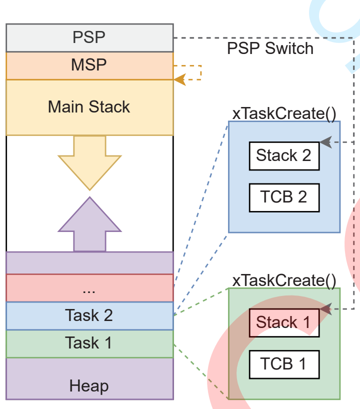
According to the operation model of unmanned systems in §2.1, we believe that a task needs to achieve a certain execution frequency to meet its functional requirement. Failure to reach the execution frequency will affect the real-time performance of unmanned systems, such as perception delay and command delay, which will directly affect the safety of unmanned systems. Therefore, we choose task execution frequency as an indicator to measure the performance impact of security features on unmanned systems. As demonstrated in Figure 11, we can see that our approach brings little system overhead. None of the task execution frequency is affected under the protection of CToMP , enabling the protected system to reach the same operation efficiency as the original system In the framework of relevant work **MINION**, many low-priority tasks are affected by the frequent configuration of MPU and cannot be executed in a cycle, and the desired execution frequency cannot be achieved. Therefore, the effect of CToMP on the efficiency of Ardupilot is completely acceptable.

At the same time, we noticed that randomizing memory allocation can lead to memory fragmentation



(a) From small to large. (b) From large to small.

Figure 12 Runtime Overhead of two kinds of memory allocation solutions.



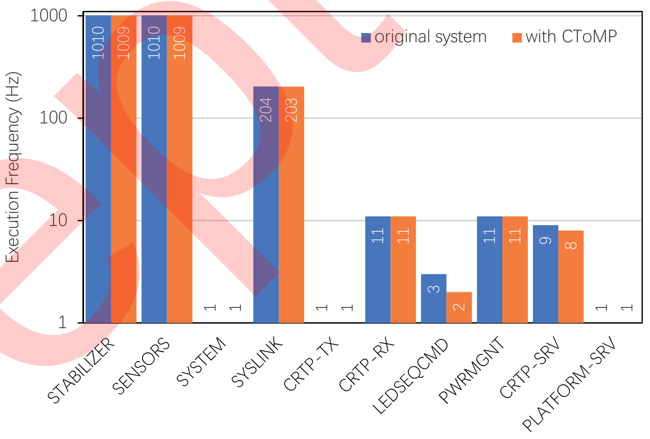


Figure 13 Task Control Block (TCB) in FreeRTOS.

Figure 14 Log-based the execution frequency of tasks with CToMP and baseline in original system Crazyflie.

issues. This causes a large number of address conflicts in memory allocation, and multiple memory reallocations will consume more time. We tried the simplest solution, which is to arrange the memory regions that need to be allocated in order of size. We found that allocating memory from large to small (Figure 12(b)) can improve the efficiency by 26.5% than allocating memory from small to large (Figure 12(a)). The time of memory allocation also can be accepted.

4.3 Study Case: Crazyflie

To illustrate the generality of CToMP , we validate it on another unmanned system, Crazyflie. Since Crzayflie uses the relatively low-end STM32F405 as the MCU, it needs more efficient security functions. There are two main missions in Crazyflie, `STABILIZER` and `SENSORS`. `STABILIZER` is mainly responsible for the attitude control of drones, while `SENSORS` is used to collect data from sensors such as gyroscopes and accelerometers. At the same time, there is also a Crazy RealTime Protocol (CRTP) service for data communication with the control side.

Similar to Ardupilot, attackers can exploit vulnerabilities[1)2)] in the firmware to inject `Shellcode`. Therefore, in CToMP , we use MPU to put `STABILIZER` into privileged mode, which is isolated from other user codes that can interact with the outside world. This can effectively protect the attitude control of drones from being interfered with the malicious code. Unlike Ardupilot, where multiple tasks

1) https://forum.bitcraze.io/viewtopic.php?t=2063
2) https://forum.bitcraze.io/viewtopic.php?t=4923

can use the same process stack, the operating system FreeRTOS in Crazyflie uses `xTaskCreate` to allocate a fixed stack for each task in RAM, and makes the PSP point to the corresponding stack when the task is scheduled, as illustrated in Figure 13. This eliminates the need to build a memory pool, but only needs to randomly change the stack address of each task in RAM in each cycle. We also tested the effect of CToMP on the real-time performance of Crazyflie, as shown in Figure 14. In the experiments, the execution frequency of Crazyflie tasks was not affected by CToMP .

5 Related Work

Since the unmanned system is a typical embedded system, the security research of embedded systems also inspires our research.

MCAs in Embedded Systems. Although the existing technology strictly checks the integrity of the embedded software, monitoring methods for executing embedded system programs are not widely used because of resource constraints [38]; hence, some attacks can cause memory corruption during runtime, such as stack/buffer overflow attacks and code reuse attacks [39]. In particular, memory overflow attacks have been one of the most mainstream methods to date [40–42], because in the most commonly used embedded program language C\C++, some functions dealing with buffer data lack a boundary detection mechanism, e.g., `strcpy()`, or because of unavoidable programmer negligence. These attacks have always been difficult to solve, and several vulnerabilities related to memory overflow attacks are reported in **CVE** every day. These vulnerabilities are distributed in the firmware of embedded devices that have been released, such as routers of various brands (TP-LINK[3)][4)], NETGEAR[5)]), webcams[6)], and even in the ARM official dependency library, which contains an unsafe function `encode_ise()`[7)]. These vulnerabilities allow unauthenticated attackers to remotely execute arbitrary code, causing severe losses. Furthermore, although few vulnerability reports are available for the UAV real-time system, after analyzing 596 bugs submitted on *github.com* in two types of open-source flight control software, Ardupilot and PX4, Wang et al. [43] found that hackers can exploit some bugs to launch security attacks. In Hooper and Tian's research [16], a buffer overflow bug was used to force a small commercial drone *Parrot* to land without cracking the Wi-Fi password used for control communication. Consequently, designing a general memory protection scheme is necessary in embedded real-time systems.

Memory Protection in Embedded Systems. To resist the damage caused by MCAs, the most effective approach is to establish a memory management mechanism to partition the memory usage region of the embedded system. Some works [44–48] use MMU in general-purpose computing systems for reference to dividing the embedded system memory into blocks and realize dynamic memory allocation. However, these schemes have considerable limitations in actual engineering applications. For example, `malloc` function is also implemented in Ardupilot, but it only provides three fixed regions for memory allocation.

In addition, there are some frameworks [9, 13–15] that implement access control to embedded system memory from the perspective of authority management. MINION [14] and M2MON [15] are two memory protection architectures based on STM32 series chips and are applied to Ardupilot. They use MPU to limit the memory-accessible range of tasks' user code that may be hacked by unauthorized users and protect sensitive data from malicious modification through memory corruption vulnerabilities. However, with the update of unmanned systems, the functions supported by drones gradually increased, and the performance of these task-oriented solutions declined. In some other low-cost platforms, such as Arduino Yun based on the AVR architecture, Sergio et al. [9] proposed to use an XOR-based encryption and a liner PRNG to protect the confidentiality of private metadata. However, this work is not universal and representative because of the platform specificity of this solution. Additionally, we also noticed that to prevent code-reuse attacks (*return2libc* attacks), [9] and [49] used Address Space Layout Randomization (ASLR) technology. However, their approaches can only randomize the code address once during firmware burning or program startup. In contrast, CToMP randomizes the stack address when each cycle begins so that the stack address will be different in each cycle, considerably increasing the difficulty of analyzing memory corruption vulnerabilities for attackers. Hence, it prevents *return2shellcode* attacks.

3) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44632
4) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-25074
5) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45524
6) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33549
7) https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44331

ARM TrustZone Security. The initial stage of TrustZone [50] is a security architecture proposed for high-performance Cortex-A processors. Recently, low-power Cortex-M33 series MCUs have also begun to support TrustZone [51], and STM32L5 [52] is an earlier chip to cover this feature, but it is not widely used. However, similar to the switching of execution levels in our system architecture, TrustZone technology in Cortex-M33 divides the secure world and normal world through memory mapping and uses an exception handler to achieve transitions [53, 54]. In the future, we can adapt CToMP into the secure world of TrustZone to complete the management of memory resources in the trusted environment.

6 Conclusion

With the widespread use of unmanned systems, the security issues they conceal have gradually gained people's attention. In this paper, we seek to tackle memory corruption attacks (MCAs), which inject malicious code through memory vulnerabilities and tamper with critical kernel instructions or data in memory.

To achieve this goal, we propose a cycle-task-oriented memory protection (CToMP) approach. By analyzing and testing with various typical attack interfaces, we found that CToMP is resilient to different types of MCAs, and it will not affect the efficiency of unmanned systems. To summarize, CToMP is an efficient and dependable memory protection mechanism that can meet the requirements of unmanned systems for velocity, practicality, and reliability simultaneously. Our source code is available on GitHub: https://github.com/xidian-uav/uav_memory_isolation.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Key Program 62232013), the Fundamental Research Funds for the Central Universities (Nos. ZYTS23202 and YJSJ23007), and the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92267204). We also thank anonymous reviewers and editors for their comments and guidance.

References

1 Stankovic J A. Real-Time and Embedded Systems. ACM Comput Surv, 1996, 28: 205-208
2 Tomic T, Schmid K, Lutz P, et al. Toward a Fully Autonomous UAV: Research Platform for Indoor and Outdoor Urban Search and Rescue. IEEE Robot Autom Mag, 2012, 19: 46-56
3 Messina G, Modica G. Applications of UAV Thermal Imagery in Precision Agriculture: State of the Art and Future Research Outlook. Remote Sensing, 2020, 12: 9
4 Chai H X, Zhang G X, Zhou J L, et al. A short review of security-aware techniques in real-time embedded systems. J Circuit Syst Comp, 2019, 28: 2
5 Zhi Y Y, Fu Z J, Sun X M, et al. Security and privacy issues of UAV: a survey. Mobile Netw Appl, 2020, 25: 95-101
6 Leccadito M, Bakker T, Klenke R, et al. A survey on securing UAS cyber physical systems. IEEE Aero El Sys Mag, 2018, 33: 22-32
7 Fei F, Tu Z, Yu R, et al. Cross-Layer Retrofitting of UAVs Against Cyber-Physical Attacks. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018
8 Xiao M B, Wang X D, Yang G S. Cross-Layer Design for the Security of Wireless Sensor Networks. In: 2006 6th World Congress on Intelligent Control and Automation, 2006. 104-108
9 Pastrana S, Tapiador J, Suarez-Tangil G, et al. AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2016. 58-77
10 Niesler C, Surminski S, Davi L. HERA: Hotpatching of Embedded Real-time Applications. In: 28th Annual Network and Distributed System Security Symposium (NDSS), 2021
11 Bai J, Li T, Lu K J, et al. Static Detection of Unsafe DMA Accesses in Device Drivers. In: 30th USENIX Security Symposium (USENIX Security 21), 2021. 1629-1645
12 Regalado D, Harris S, Harper A, et al. Gray Hat Hacking The Ethical Hacker's Handbook, Fourth Edition. McGraw-Hill Education, 2015
13 Koeberl P, Schulz S, Sadeghi A, et al. TrustLite: A Security Architecture for Tiny Embedded Devices. In: Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14), 2014
14 Kim C H, Kim T, Choi H, et al. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In: 25th Annual Network and Distributed System Security Symposium (NDSS), 2018
15 Khan A, Kim H, Lee B, et al. M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles. In: 30th USENIX Security Symposium (USENIX Security 21), 2021
16 Hooper M, Tian Y F, Zhou R X, et al. Securing commercial WiFi-based UAVs from common security attacks. In: 2016 IEEE Military Communications Conference (MILCOM 2016), 2016. 1213-1218
17 Wang J W, Li A, Li H R, et al. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In: 2022 IEEE Symposium on Security and Privacy (SP), 2022. 352-369
18 Hardin T, Scott R, Proctor P, et al. Application Memory Isolation on Ultra-Low-Power MCUs. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018. 127-132
19 Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO), 2004. 75-86
20 STMicroelectronics. STM32F7 Series and STM32H7 Series Cortex®-M7 processor Programming Manual. 2019
21 STMicroelectronics. Introduction to STM32 microcontrollers security. 2021

22 Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 07), 2007. 552–561
23 Roemer R, Buchanan E, Shacham H, et al. Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans Inf Syst Secur, 2012, 15: 1
24 Mouzarani M, Sadeghiyan B, Zolfaghari M. Smart fuzzing method for detecting stack-based buffer overflow in binary codes. IET Softw, 2016, 10: 96-107
25 Rawat S, Mounier L. Offset-Aware Mutation Based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011. 531-533
26 Mouzarani M, Sadeghiyan B, Zolfaghari M. A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes. In: 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), 2015. 42-49
27 Redini N, Machiry A, Wang R Y, et al. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In: 2020 IEEE Symposium on Security and Privacy (SP), 2020
28 Costin A, Zarras A, Francillon A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16), 2016. 437–448
29 David Y, Partush N, Yahav E. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. SIGPLAN Not, 2018, 53: 392–404
30 Qasem A, Shirani P, Debbabi M, et al. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. ACM Comput Surv, 2021, 54: 2
31 Wright C, Moeglein W A, Bagchi S, et al. Challenges in Firmware Re-Hosting, Emulation, and Analysis. ACM Comput Surv, 2021, 54: 1
32 Clements A A, Gustafson E, Scharnowski T, et al. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In: 29th USENIX Security Symposium (USENIX Security 20), 2020
33 Gustafson E, Muench M, Spensky C, et al. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), 2019
34 Johnson E, Bland M, Zhu Y F, et al. Jetset: Targeted Firmware Rehosting for Embedded Systems. In: 30th USENIX Security Symposium (USENIX Security 21), 2021
35 Han R D, Yang C, Ma S Q, et al. Control Parameters Considered Harmful: Detecting Range Specification Bugs in Drone Configuration Modules via Learning-Guided Search. In: Proceedings of the 44th International Conference on Software Engineering (ICSE '22), 2022. 462–473
36 Kim H, Lee J, Pratama D, et al. RIMI: Instruction-Level Memory Isolation for Embedded Systems on RISC-V. In: Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20), 2020
37 STMicroelectronics. STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs Reference Manual. 2018
38 Wang W K, Liu M Y, Du P, et al. An Architectural-Enhanced Secure Embedded System with a Novel Hybrid Search Scheme. In: 2017 International Conference on Software Security and Assurance (ICSSA), 2017. 116-120
39 Das S, Zhang W, Liu Y. A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems. IEEE Trans VLSI Syst, 2016, 24: 3193-3207
40 Mullen G, Meany L. Assessment of Buffer Overflow Based Attacks On an IoT Operating System. In: 2019 Global IoT Summit (GIoTS), 2019. 1-6
41 Rajendran G, Nivash R. Security in the Embedded System: Attacks and Countermeasures. In: Proceedings of International Conference on Recent Trends in Computing, Communication & Networking Technologies (ICRTCCNT), 2019
42 Xu B, Wang W K, Hao Q, et al. A Security Design for the Detecting of Buffer Overflow Attacks in IoT Device. IEEE Access, 2018, 6: 72862-72869
43 Wang D H, Li S Q, Xiao G P, et al. An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. 20-31
44 Bai L S, Yang L, Dick R P. MEMMU: Memory Expansion for MMU-Less Embedded Systems. ACM Trans Embed Comput Syst, 2009, 8: 3
45 Bukkapatnam K, Prashant, Rekha C K, et al. Smart Memory Management (SaMM) For Embedded Systems without MMU. In: International Conference on Recent Advancements in Engineering and Management (ICRAEM-2020), 2020. 3
46 Cheng X H, Gong Y M, Wang X Z. Study of Embedded Operating System Memory Management. In: 2009 First International Workshop on Education Technology and Computer Science, 2009. 962-965
47 Deligiannis I, Kornaros G. Adaptive memory management scheme for MMU-less embedded systems. In: 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), 2016. 1-8
48 Yu Y H, Wang J Z, Sun T Y. A Novel Defragmemtable Memory Allocating Schema for MMU-Less Embedded System. In: Advances in Intelligent Systems and Applications - Volume 2, 2013
49 Salehi M, Hughes D, Crispo B. MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks. In: 2019 IEEE Conference on Dependable and Secure Computing (DSC), 2019. 1-8
50 ARM Ltd. ARM Security Technology - Building a Secure System using TrustZone Technology. 2009
51 ARM Ltd. Introduction to the ARMv8-M architecture. Version 2.0. 2017
52 STMicroelectronics. STM32L552xx and STM32L562xx advanced Arm®-based 32-bit MCUs Reference Manual. 2020
53 Azab A M, Ning P, Shah J, et al. Hypervision Across Worlds: Real-Time Kernel Protection from the ARM TrustZone Secure World. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14), 2014. 90–102
54 Pinto S, Santos N. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Comput Surv, 2019, 51: 6