MICROFT: Exploring and Mitigating Cross-state Control-flow Hijacking Attacks on ARM Cortex-M TrustZone

Zheyuan Ma, Xi Tan, Lukasz Ziarek, *Member, IEEE*, Ning Zhang, *Member, IEEE*, Shambhu Upadhyaya, *Fellow, IEEE*, Hongxin Hu, *Member, IEEE*, Ziming Zhao, *Member, IEEE*

***Abstract*—ARM Cortex-M is one of the most popular microcontroller architectures designed for deeply embedded and Internet of Things (IoT) applications. To facilitate efficient execution, it has some unique hardware optimization. Specifically, Cortex-M TrustZone has a fast state switch mechanism that allows direct control-flow transfer from the secure state program to the non-secure state userspace program. In this paper, we present MICROFT – exploring and mitigating cross-state control-flow hijacking attacks on ARM Cortex-M TrustZone. In particular, we first demonstrate how Cortex-M TrustZone's fast state switch mechanism can be exploited for arbitrary code execution with escalated privilege in the non-secure state by introducing a new exploitation technique, namely return-to-non-secure (ret2ns). We present the detailed methodology of ret2ns attacks in two representative cases and experimentally confirm the feasibility of four variants of attacks on two hardware platforms. To defend against ret2ns attacks, we design three address sanitizing mechanisms while imposing a negligible performance overhead of less than 0.1%. The first mechanism is a generic MPU-assisted address sanitizer, while the second and third mechanisms are more efficient software-fault isolation based approaches that assume the userspace and kernel space programs are placed in different and known memory regions.**

***Index Terms*—ARM Cortex-M, embedded systems security, exploitation techniques, TrustZone**

I. INTRODUCTION

ARM Cortex-M is the dominating 32-bit microcontroller architecture. In the 4th quarter of 2020 alone, 4.4 billion Cortex-M-based devices were shipped [1]. In contrast to microprocessors, like Cortex-A used in smartphones and laptops, Cortex-M does not include a memory management unit (MMU) and targets deeply embedded and Internet of Things (IoT) deployments. Example embedded and IoT products built on this architecture include (1) consumer devices like Fitbit Flex and Oculus VR; (2) electronic control units in vehicles; and (3) data communication subsystems in mobile phones, e.g., Bluetooth controllers. To facilitate efficient execution and low power consumption, Cortex-M has some unique hardware optimization that Cortex-A and other microprocessors do not offer. For instance, to provide faster interrupt handling, Cortex-M has a hardware-assisted stacking and unstacking mechanism that automatically stores and restores the contents of general registers to and from the stack.

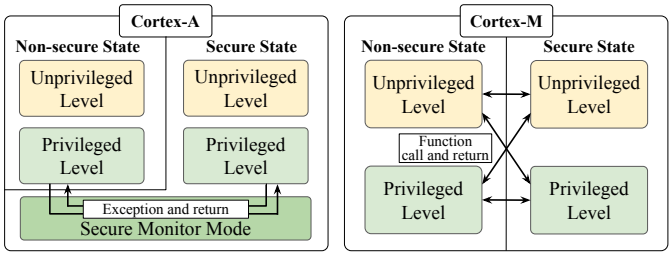


Fig. 1: State switches on Cortex-A TrustZone must go through the privileged secure monitor mode, whereas state switches on Cortex-M can occur between different secure and non-secure privilege levels.

ARM TrustZone is a hardware-assisted trusted execution environment (TEE) that splits system-on-chip resources between two execution *states*, non-secure and secure. Software running in the secure state can access all resources, whereas software in the non-secure state can only access non-secure resources. First introduced with Cortex-A [2], TrustZone has been recently extended to Cortex-M [3]–[5], but optimized for performance. Different from Cortex-A, which indicates the security state in the secure configuration register, the division of states in Cortex-M is based on memory regions. When running code in the secure memory, the processor is in the secure state. Otherwise, the processor is in the non-secure state. As shown in Figure 1, state switches on Cortex-A must go through a single entry point – the privileged secure monitor mode – via the secure monitor call instruction (`smc`), whereas state switches in Cortex-M can occur through function calls and returns, resulting in an unlimited number of entries between secure and non-secure privilege levels. While faster, the security implications of this state switch mechanism have not been thoroughly studied.

The inherent semantic gap between a secure state program and the non-secure state memory inevitably leads to confused deputy vulnerabilities [6], [7], where the secure state program can be tricked or exploited by a non-secure state program into misusing its authority or ability [8], [9]. Boomerang [10]

A preliminary version of this manuscript titled "Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense" was published in the Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC), San Francisco, USA 2023.

Zheyuan Ma and Xi Tan are the co-first authors. The corresponding author is Ziming Zhao. Zheyuan Ma, Xi Tan, and Ziming Zhao are also with the CactiLab.

Zheyuan Ma, Lukasz Ziarek, Shambhu Upadhyaya, and Hongxin Hu are with the Department of Computer Science and Engineering, University at Buffalo, USA. E-mail: {zheyuanm, lziarek, hongxinh}@buffalo.edu.

Xi Tan is with the Department of Computer Science, University of Colorado Colorado Springs, USA. E-mail: xtan4@uccs.edu.

Ziming Zhao is with the Khoury College of Computer Sciences, Northeastern University, USA. E-mail: z.zhao@northeastern.edu.

Ning Zhang is with the Department of Computer Science and Engineering, Washington University in St. Louis, USA. E-mail: zhang.ning@wustl.edu.

is a class of such vulnerabilities discovered on Cortex-A secure state programs, which allows malicious non-secure userspace applications to read and write the non-secure kernel memory by misleading the secure state program to do so on its behalf. Boomerang bugs could exist in any TEE implementation where the secure state program can access non-secure state memory, including Cortex-M TrustZone. Fortunately, boomerang does not directly lead to arbitrary code execution with escalated privilege.

In this paper, we present MICROFT – exploring and mitigating cross-state control-flow hijacking attacks on ARM Cortex-M TrustZone. Firstly, we report a new class of confused deputy attacks, namely return-to-non-secure (ret2ns), that exploit the fast state switch mechanism of the Cortex-M TrustZone. More dangerous than boomerang, ret2ns can lead to arbitrary code execution with escalated privilege in the non-secure state. Ret2ns is also a new type of return-to-user (ret2usr) attacks [11], [12] that redirect compromised secure state pointers to code residing in non-secure state userspace.

The *root cause* of the ret2ns vulnerability is the exploitation of the extensive state transition surface in Cortex-M TrustZone. This differs from the bypass of pointer sanitization seen in boomerang attacks and the exploitation of the shared kernel-user address space found in Unix-like systems in the ret2usr method. Ret2ns attacks affect all Cortex-M processors with TrustZone, including M23 [13], M33 [14], M35P [15], M55 [16], and M85 [17]. We also argue ret2ns vulnerabilities are likely to exist in any TEE implementations that allow direct control-flow transfers from secure state to non-secure userspace programs but keep executing at the privileged level.

Similar to ret2usr, which can be defeated by preventing arbitrary control-flow transfers and dereferences from kernel to userspace [11], ret2ns can be thwarted by preventing control-flow transfers from secure state to non-secure state userspace by using the privileged execute-never (PXN) feature. However, PXN is not available on most Cortex-M processors. Only the planned Cortex-M55 and M85 microcontrollers will support this feature [18]. To defeat ret2ns attacks on Cortex-M microcontrollers without PXN, we propose two address sanitizing mechanisms with negligible performance overhead. The first one utilizes the memory protection unit (MPU) in the non-secure state to add proper checks before the state transition happens from the secure state. The checks will get the current non-secure privilege level and examine the MPU setting of the destination address to determine whether privilege escalation is attempted. To further reduce the performance overhead, the second approach applies address masking but requires the non-secure userspace program and kernel space program to be placed in different and known ranges.

The contributions of this paper are as follows:

- We analyze and report a security design weakness in the fast state switch mechanism of Cortex-M TrustZone by introducing the concept of ret2ns attacks;
- We provide a detailed comparison of ret2ns with ret2usr and boomerang attacks, focusing on their threat models, effects of the attacks, and underlying causes;
- We present a detailed methodology for four variants of ret2ns attacks and incorporate comprehensive walk-through examples to illustrate two representative variants;
- We experimentally evaluate the effectiveness of ret2ns attacks on Cortex-M23 and M33 systems;
- We introduce three address sanitizing mechanisms: an MPU-assisted approach and two based on software-fault isolation, designed to mitigate ret2ns attacks with minimal runtime overhead;

The rest of this paper is organized as follows. Section II provides a background on the ARM Cortex-M architecture. The overview of the ret2ns attack, illustrated by two detailed example applications, is presented in Section III. Section IV presents two proposed defense strategies, including their effectiveness and performance evaluations. Section V reviews and compares relevant literature in the field. Section VI delves into the limitations and expanded discussion of the MICROFT study, and finally, Section VII concludes the paper.

We open-source our project[1], which includes vulnerable code examples, proof-of-concept exploits, defense instrumentation, and evaluation results.

II. BACKGROUND

In this section, we present the background knowledge of the ARM Cortex-M architecture. In particular, we discuss its execution modes, the TrustZone extension, and the memory protection unit.

A. Cortex-M Execution Modes

All Cortex-M processors with the TrustZone extension have thread and handler execution modes [19]. They also have privileged (kernel space) and unprivileged (userspace) levels, which are orthogonal to the execution mode. The current mode and privilege level are determined by the combination of the interrupt program status register (`IPSR`) and the `CONTROL` register. `IPSR` is part of the program status register (`xPSR`). `IPSR` indicates the exception number and handler mode if not 0. If `IPSR` is 0, the processor is in the thread mode, and the `nPRIV` bit of `CONTROL` determines whether the state is unprivileged or not.

To switch the processor from privileged to unprivileged, software can simply change `CONTROL.nPRIV` to 1 using the `MSR` instruction. To switch from unprivileged to privileged, software makes a supervisor call (SVC) with the `svc` instruction. When a higher priority interrupt or exception occurs, the processor automatically pushes eight registers, including program status register (`xPSR`), program counter (`PC`), and link register (`LR`), to the current stack. Then, the processor generates a special exception return value named `EXC_RETURN` (`0xFFFFFF**`), stores it in `LR`, and executes the interrupt service routine (ISR), e.g., SVC handler. When an ISR exits and `EXC_RETURN` is copied to the `PC`, the processor

[1]https://github.com/CactiLab/ret2ns-Cortex-M-TrustZone

will automatically perform unstacking, which pops the eight registers off the stack. The hardware-assisted mechanism of stacking and unstacking makes it possible to develop interrupt handlers solely in C programming language.

B. Cortex-M TrustZone

TrustZone adds another orthogonal partitioning of states. With TrustZone, the processor has the secure and non-secure state. The secure and non-secure states are hardware separated, and the non-secure software can only access non-secure resources, including memories and memory-mapped peripherals, whereas secure software can access both secure and non-secure resources. The TrustZone for Cortex-A uses a dedicated monitor mode to handle the secure state transitions. The processor will be running at the secure state while in the monitor mode, and uses the `NS` bit in the secure configuration register (`SCR`) to define the operation state to which the CPU will switch after the monitor mode. Therefore, any cross-state transitions will go through the single entry point — secure monitor mode.

Different from Cortex-A, the division of secure and non-secure in Cortex-M is memory-map-based, and transitions between states take place automatically. If the processor is running code in the secure memory, it is in the secure state. Otherwise, the processor is in the non-secure state. The `nPRIV` bit of `CONTROL` is banked between two states. In the rest of the paper, we use `CONTROL_NS.nPRIV` to refer to the non-secure state copy and `CONTROL_S.nPRIV` for the secure state copy. The `IPSR` is not banked. With TrustZone, a memory region can be secure, non-secure callable (NSC), or non-secure. The NSC represents the entry point of the transition from the non-secure state to the secure state, which must start with an `sg` (secure gateway) instruction. The interstate branch instructions `blxns` (indirect call) and `bxns` (indirect jump) are used to switch from secure to non-secure state. When calling a non-secure function, the `blxns` instruction pushes the return program status register (`RETPSR`) and the return address onto the secure stack. The link register `LR` will also be updated to `FNC_RETURN` (`0xFEFFFFFF`). When a non-secure function returns, `FNC_RETURN` is loaded into `PC`, which triggers the processor to unstack the `RETPSR`, check the `IPSR`, and clear interruptions, then unstack the real return address from the secure stack. The state switching mechanism for the `bxns` instruction is more straightforward. If the target address is not `FNC_RETURN` or `EXC_RETURN` and the bit 0 of the `LR` register is 0, `bxns` branches to non-secure code directly.

C. Cortex-M Memory Protection Unit

The memory protection unit (MPU) is a programmable unit inside a Cortex-M processor that monitors all memory accesses, including instruction fetches and data accesses, over software-designated regions. The permission for an MPU region is determined by the access permission field and the eXecute-Never (XN) bit, which determines whether execution is permitted when read is permitted. The access permission field has four possible values: (i) read-only by any privilege level; (ii) read-only by privileged; (iii) read/write by any privilege level; and (iv) read/write by privileged. In the ARMv8.1-M architecture, the planned Cortex-M55 and M85 introduce the Privileged eXecute-Never attribute (PXN), which allows an MPU region containing the userspace application or library to be marked as unprivileged-execution-only. When TrustZone is implemented, MPU is banked between the two states. The security state can use the test target alternate domain instruction (`tta`) to retrieve the access permissions of a non-secure state address.

III. The ret2ns Attacks

In this section, we will first discuss the system and threat model and provide an overview of the ret2ns attack. Next, we will delve into two detailed attack variants, namely the handler-mode-originated and `bxns` case, as well as the thread-mode-originated and `blxns` case.

A. System and Threat Model

Our work focuses on Cortex-M microcontrollers with TrustZone extension running both secure and non-secure state software. On the hardware front, such systems lack an MMU and other security features, e.g., PXN. On the non-secure state software front, we assume these systems either run (1) a real-time operating system (RTOS), e.g., FreeRTOS [20], with userspace and kernel modules, where the kernel services execute in the handler mode and are dispatched in the SVC handler; (2) a security-enhanced bare-metal system that supports privilege separation. For example, EPOXY [21] and ACES [22] identify operations requiring privileged execution in bare-metal systems and modify the systems to only execute those operations in the privileged thread mode. The attack does not utilize any bugs in the kernel module of the non-secure state, but we assume a confined vulnerability in the secure state firmware or library, e.g., ARM TF-M [23]. Note that ret2ns vulnerabilities are likely to exist in any TEE applications that allow direct control-flow transfers from secure state to non-secure state userspace programs while still executing at the privileged level.

We assume a userspace attacker in the non-secure state who seeks to elevate privileges by exploiting a memory corruption vulnerability in the secure state. The attacker does not need physical access to the system, but rather remotely controls the input to a userspace program that interacts with the secure state program, by going through proper non-secure state system calls via the supervisor call instruction (`svc`). The vulnerability can be in either userspace or kernel space of the secure state. The attacker is only able to corrupt a single code pointer used by a `bxns` or `blxns` instruction in the secure state program, and we do not assume the attacker can corrupt any other code pointer, e.g., those used by `bx` or `blx`, in the secure state. After all, the attacker's goal is not to execute arbitrary code in the secure state, and with the constricted vulnerability, they cannot hijack the control flow in the secure state either. Overall, the adversarial capabilities we presume are similar to those needed for carrying out ret2usr/ret2dir attacks on x86 or Boomerang attacks on Cortex-A.

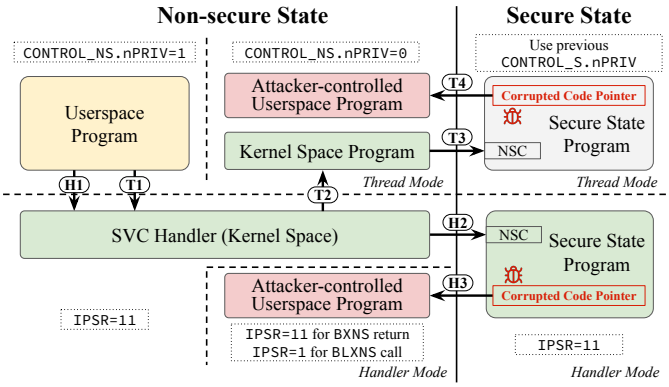


Fig. 2: Overview of ret2ns attacks. A secure state code pointer used by `bxns` or `blxns` is corrupted and redirected to an attacker-controlled userspace program in the non-secure state. After the state switch, the attacker-controlled userspace program executes at the privileged level.

B. Attack Overview

Figure 2 shows how ret2ns attacks work at a high level. Based on the non-secure execution mode that an attack originates from, we break ret2ns attacks into two categories: handler-mode-originated and thread-mode-originated attacks. Attacks in the former category are more likely to happen in RTOSes, whereas attacks in the latter category are likely to occur in security-enhanced bare-metal systems that support privilege separation. Attacks in either category can be further attributed to an indirect branch case using `bxns` or an indirect call case using `blxns`, resulting in four variants of ret2ns attacks.

In the handler-mode-originated attacks, a userspace program takes the attacker's input and makes a supervisor call (**H1**), so the processor enters the handler mode and `IPSR` is updated to 11 (the interrupt number of SVC). The SVC handler in turn calls a non-secure callable (NSC) function (**H2**), and the processor switches to the secure state. Because `IPSR` is shared between the secure and non-secure state, the secure state program keeps executing in the handler mode with privilege. In a legitimate control path, when the secure handler program exits back to the non-secure state using `bxns`, the control returns to the SVC handler. However, if the `bxns` instruction uses a corrupted code pointer as the destination, the processor can return to any location, e.g., userspace program (**H3**), in the non-secure state and keep executing it in the handler mode with privilege due to the unchanged `IPSR`. Another attack path exists when a secure handler program makes an indirect function call (`blxns`) with a corrupted code pointer (**H3**). In this case, `IPSR` updates to the value of 1 after this call to mask the identity of the secure exception service [19].

In the thread-mode-originated attacks, the unprivileged program uses an SVC call to escalate the non-secure privilege level with `CONTROL_NS.nPRIV` cleared (**T1**), after which a privileged program in the thread mode executes (**T2**). The privileged program takes the attacker's input and in turn calls an NSC function in the secure state (**T3**). The NSC function will call the secure state program, which eventually returns the control to the non-secure state (using `bxns`) or calls a non-secure callback function (using `blxns`). When a memory corruption vulnerability in the secure state program leads to a corrupted code pointer, the control flow will transfer to an attacker-controlled program in the non-secure state (**T4**). Since the non-secure state has `CONTROL_NS.nPRIV` cleared, the attacker-controlled program will keep executing in the privileged thread mode.

C. The Handler-mode-originated and bxns Case

We first demonstrate a detailed attack walk-through of the handler-mode-originated and `bxns` case, the source code and attack steps of which are shown in Listing 1 and Figure 3. This example represents a secure display function that can be implemented in any RTOS running in the non-secure state and a firmware running in the secure state. In this example, the non-secure state cannot control the LCD display because the LCD peripheral registers are only memory-mapped to the secure state address space. Instead, it uses the display service provided by the secure state.

When a non-secure state userspace program wants to print a message on the LCD, it calls the userspace library function *print_LCD()*, which makes an SVC call with number 0 to enter the handler mode, i.e., `IPSR=11` (① and Listing 1a line 8-9). The physical address of the user-supplied message is passed to the SVC handler in `R0`. The *SVC_Handler* parses the request and dispatches it to the secure state by calling the non-secure callable function *print_LCD_nsc()* (Listing 1a line 19) defined in the secure state (②). Because `IPSR` is not banked during the state switch, the secure state keeps executing in the handler mode with `IPSR=11`. The NSC function has an attribute of *cmse_nonsecure_entry*, so the ARMClang compiler knows to emit (1) a secure gate and branch instruction for this function in the non-secure callable memory region, and (2) a `bxns` instruction instead of the regular `bx` for the function return.

In ③, *print_LCD_nsc()* checks whether the LCD is ready by calling the corresponding driver function (Listing 1b line 8). If the LCD is ready (④), *print_LCD_nsc()* concatenates the user message to some timestamp and system status information and calls the driver function to print the message. Because *print_LCD_nsc()* is not a leaf function, its `LR` value is spilled to the stack. If the user-supplied message is long enough to overwrite the local variable *buf* (⑤), the saved `LR` value in the stack frame of *print_LCD_nsc()* can be corrupted, e.g., changed to the address of *attacker_controlled()* in the non-secure state. Note that the exploited memory corruption vulnerabilities do not have to exist in the non-secure callable function, they can also exist in functions called by the non-secure callable functions. In other words, any memory corruption vulnerability, e.g., format string, that can lead to the corruption of the saved `LR` value on the stack frame of a non-secure callable function can be exploited. When *print_LCD_nsc()* returns to the non-secure state using `bxns` (⑥), the processor keeps executing from the corrupted return address, e.g., *attacker_controlled*, in the handler mode.

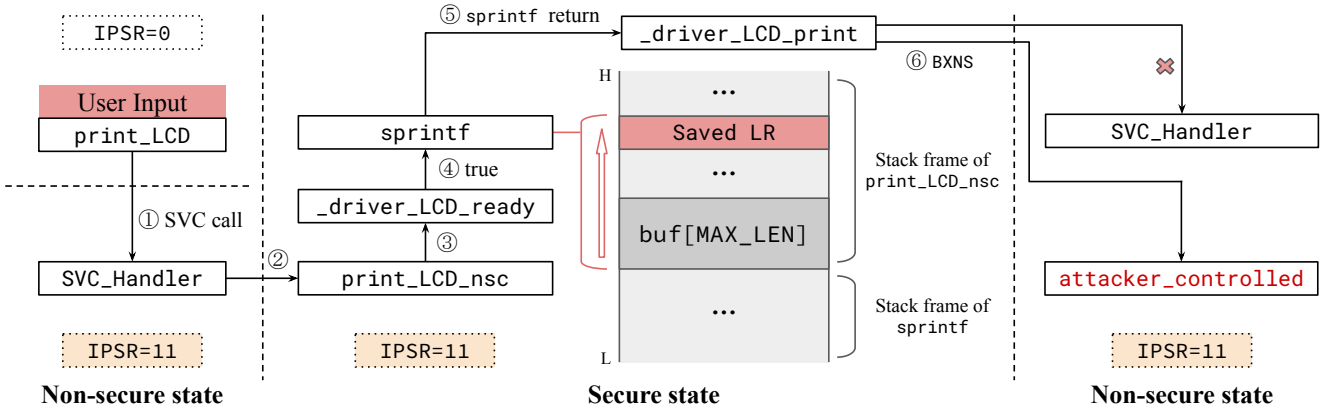


Fig. 3: Attack walk-through of the example handler-mode-originated and `bxns` case.

```c
/* Userspace function */
void attacker_controlled();

/* Library function. Callable by userspace program. */
void print_LCD(char *msg)
{
    register char* r0 __asm("r0") = msg;
    __asm volatile("svc #0"
    : "r"  (r0));
}

/* Kernel space handler */
void SVC_Handler(unsigned int *svc_args)
{
    uint32_t svc_number = (((char *)svc_args[6])[-2]);
    switch (svc_number)
    {
        case 0:
        print_LCD_nsc((char *)svc_args[0]); break;
        ...
    }
}
```

(a) Non-secure state code

```c
#define MAX_LEN 128
int32_t _driver_LCD_ready();
int32_t _driver_LCD_print(char *);

/* Non-secure callable function */
int32_t print_LCD_nsc(char *msg)
  ↪ __attribute__((cmse_nonsecure_entry));

int32_t print_LCD_nsc(char *msg)
{
    char buf[MAX_LEN] = {0};

    if (_driver_LCD_ready())
    {
        sprintf(buf, "%s %s: %s", _TIME_STAMP,
        ↪ _SYSTEM_STATUS, msg); /* Buffer overflow */

        return _driver_LCD_print(buf); /* bxns return */
    }
    else
        return -1; /* bxns return */
}
```

(b) Secure state code

Listing 1: Example code snippets for the handler-mode-originated and `bxns` case.

D. The Thread-mode-originated and blxns Case

In this section, we present a detailed analysis of a representative attack scenario originating from the thread-mode and involving the `blxns` case. The corresponding source code and attack steps can be found in Listing 2 and Figure 4. This example illustrates a secure display function analogous to that of the `bxns` case. In this scenario, the non-secure state is unable to directly control the LCD display due to the LCD peripheral registers being exclusively memory-mapped to the secure state address space. As a workaround, the non-secure state leverages the display service provided by the secure state. Prior to displaying content on the LCD, the secure state invokes a non-secure state driver-checking function to obtain the current driver status.

When a non-secure state userspace program wishes to display a message on the LCD, it invokes the userspace library function *print_LCD()*. This function first initiates an SVC call with the number 0 to escalate its privilege, i.e., `CONTROL_NS.nPRIV=0` (in ① and Listing 2a line 9). Subsequently, while operating in privileged execution mode, *print_LCD()* proceeds to call the non-secure callable function *print_chk_LCD_nsc()* (Listing 2a line 10), which transfers the user message to the secure state (②). Notably, during the state switch, since the processor is not in handler mode, the secure state continues executing in its previous mode, as determined by the `CONTROL_S.nPRIV` bit.

At step ③, the function *print_chk_LCD_nsc()* concatenates the user message with a timestamp and system status information (as shown in Listing 2b line 13). Subsequently, it calls the *check_driver* function pointer to verify the driver status (④) and display the message on the LCD. The *check_driver* pointer is initialized by the *cmse_nsfptr_create()* function (Listing 2b line 10), which is an intrinsic library function designed to create a non-secure function pointer from the secure state. As a result, the compiler is aware that it must generate a `blxns` instruction rather than the standard `blx` when calling this function pointer.

If the user-supplied message is sufficiently lengthy to overwrite the local variable *buf* (Listing 2b line 13), the *check_driver* function pointer in the stack frame of *print_chk_LCD_nsc()* may be compromised. For instance, it could be altered to point to the address of a malicious function,

```c
/* Userspace function */
void attacker_controlled();
/* Kernel space function */
void check_driver();

/* Library function. Callable by userspace program. */
void print_LCD(char *msg)
{
    __asm volatile("svc #0");
    print_chk_LCD_nsc(msg);
}

/* Kernel space handler */
void SVC_Handler(unsigned int *svc_args)
{
    uint32_t svc_number = (((char *)svc_args[6])[-2]);
    switch (svc_number)
    {
        case 0:
        __asm("msr CONTROL, %[reg] " ::[reg] "r"(0) \
        : "memory"); break;
        ...
    }
}
```

(a) Non-secure state code

```c
#define MAX_LEN 128
int32_t _driver_LCD_print(char *);

/* Non-secure callable function */
int32_t print_chk_LCD_nsc(char *msg)
  ↪ __attribute__((cmse_nonsecure_entry));

int32_t print_chk_LCD_nsc(char *msg)
{
    chk_func_ptr *check_driver;
    check_driver = cmse_nsfptr_create(fp);
    char buf[MAX_LEN] = {0};

    sprintf(buf, "%s %s: %s", _TIME_STAMP,
    ↪ _SYSTEM_STATUS, msg); /* Buffer overflow */
    if (check_driver())  /* blxns call */
    {
        return _driver_LCD_print(buf); /* bxns return */
    }
    else
    {
        return -1; /* bxns return */
    }
}
```

(b) Secure state code

Listing 2: Example code snippets for the thread-mode-originated and `blxns` case.

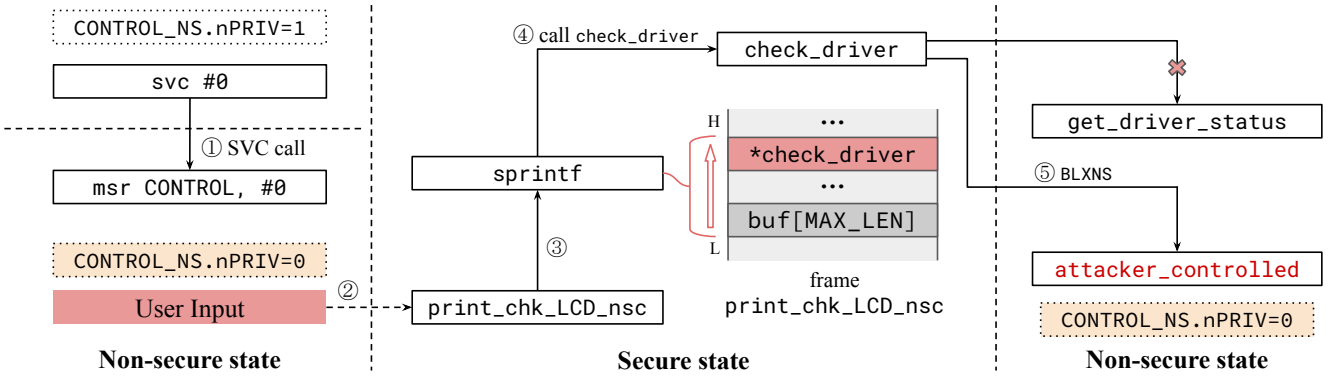


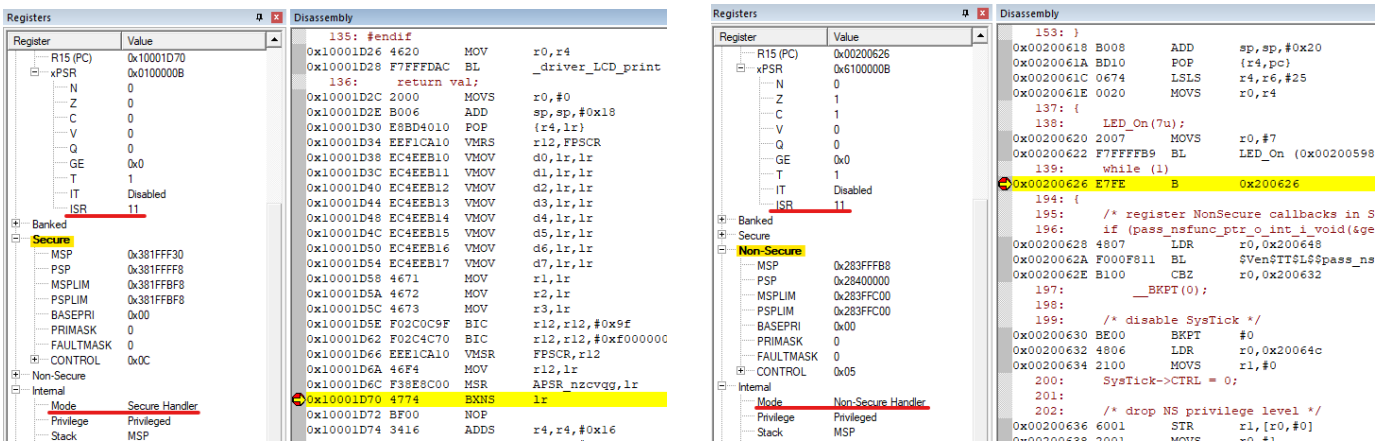
Fig. 4: Attack walk-through of the example thread-mode-originated and `blxns` case.

such as *attacker_controlled()*, in the non-secure state. When the *check_driver* function is eventually invoked using the `blxns` instruction (⑤ and Listing 2b line 14), the processor transitions to the non-secure state and resumes execution from the corrupted return address (e.g., *attacker_controlled*) in privileged thread mode, maintaining the previously banked `CONTROL_NS.nPRIV=0` setting.

E. Facilitating Bugs and Exploitability Analysis

a) Enabling Bug Types and precise corruption paths: We make explicit the vulnerability classes that enable ret2ns and the exact control data that the attacker must corrupt to influence a cross-state transfer. **Stack-based buffer overflows** are the most direct enabler for the `bxns` case, because the NSC callee's epilogue restores `LR` from its stack frame and then returns with `bxns lr`. Consequently, overflowing a local object in a secure-state NSC function to overwrite the saved `LR` causes the epilogue to load an attacker-chosen non-secure address into `LR`, after which `bxns lr` transfers control cross-state while preserving the non-secure privilege context described in Section III-C. For the `blxns` case, the branch operand is a general-purpose register loaded from a function pointer, often a local or global variable initialized via `cmse_nsfptr_create()`. Overwriting that function pointer (e.g., by a local stack overflow that spans into the pointer variable or by an out-of-bounds write into a global callback table) leads the subsequent indirect call to execute `blxns Rm` to an attacker-chosen non-secure address with the preserved privilege context. **Format-string bugs** can reach the same control data by writing attacker-chosen values into stack slots that hold the saved `LR` or a spilled function pointer, thereby triggering the `bxns` or `blxns` paths, respectively. **Integer overflows and underflows** in size or index calculations can act as enablers for both cases by defeating bounds checks and turning copies into the aforementioned stack or global overwrites. **Use-after-free or dangling-pointer bugs** are less common in heapless bare-metal builds, but remain plausible in RTOS or pool-based heaps; when present, they are relevant to ret2ns primarily through stale or reclaimed objects that contain function pointers later invoked with `blxns`, not through `LR` directly.

b) Corruption path for the handler-mode `bxns` variant: In the handler-originated scenario, the secure callee executes in handler mode and compilers emit a `bxns` return for NSC



(a) The secure state is running in the handler mode before state switching.

(b) After switching to the non-secure state, CPU will keep running in the handler mode since the IPSR register is shared between states.

Fig. 5: A successful handler-mode-originated attack. Showing the debugger with Keil IDE running on the ARM MPS2+ FPGA prototyping board with a Cortex-M33 MCU.

functions. The callee's prologue spills `LR` to the secure stack, and its epilogue reloads `LR` and issues `bxns lr`. If an attacker-controlled input overflows a local buffer (or a format-string write targets the frame), the saved `LR` in that frame is replaced with a non-secure userspace address under the attacker's control. At epilogue, the corrupted `LR` is restored and `bxns lr` transfers control to that address in the non-secure state while maintaining the privileged execution context determined by `IPSR` and `CONTROL_NS`.

c) Corruption path for the thread-mode `blxns` variant: In the thread-originated scenario, the secure code prepares a non-secure callable function pointer (e.g., via `cmse_nsfptr_create()`) and subsequently invokes it. Compilers implement this call as a register-indirect `blxns Rm`, where `Rm` is loaded from the pointer variable. If an attacker-controlled overflow overwrites that pointer variable in the stack frame (or an out-of-bounds write corrupts a global callback pointer), the subsequent indirect call loads the attacker-chosen non-secure address into `Rm`. When the call executes, `blxns Rm` transfers control to the attacker's non-secure code while preserving the privileged thread context as described in Section III-D.

d) Prevalence and exploit difficulty in real-world embedded systems: Across secure firmware for Cortex-M TrustZone, stack-based buffer overflows in NSC call paths are common due to pervasive C usage, fixed-size local buffers, and incomplete length validation [24], [25], and they directly enable the `bxns lr` path, making them high-likelihood and low-to-moderate difficulty. Function-pointer overwrites are also realistic wherever secure code holds callback pointers to non-secure services or stores `cmse_nsfptr_create()` results in stack or global variables, yielding moderate likelihood and moderate difficulty because they require the pointer to be subsequently invoked via `blxns`. Format-string bugs are less prevalent on microcontrollers because embedded C libraries may restrict directives such as `%n`, yet unsafe formatter usage still occurs; their likelihood is moderate-to-low and the difficulty depends on available write primitives into the `LR` slot or pointer slots. Integer arithmetic bugs that defeat bounds checks remain frequent and primarily act as facilitators that convert logic mistakes into the buffer- and pointer-overwrite primitives above, so their likelihood is moderate and difficulty is moderate [26]. Dangling-pointer and use-after-free paths depend on dynamic allocation; they are rare on heapless bare-metal builds but plausible in RTOS-based systems and vendor stacks that employ pool or heap allocators (e.g., fixed-block pools or FreeRTOS heap variants) [27], and when present they principally influence ret2ns by corrupting function pointers later invoked with `blxns`, yielding low-to-moderate likelihood and moderate difficulty.

F. Effectiveness Evaluation

We evaluated the effectiveness of the four variants of ret2ns attacks on two hardware systems: (1) Microchip SAM L11 evaluation platform with a Cortex-M23 microcontroller; and (2) AN505 IoT kit image for the ARM MPS2+ FPGA prototyping board with a Cortex-M33 microcontroller [28]. On the software front, we changed the example TrustZone projects that came with Microchip Studio IDE [2] and Keil IDE [3] to inject a memory corruption vulnerability in a non-secure callable function as shown in Listing 1. The experimental evaluations confirmed the attacker can escalate privilege and execute arbitrary code by exploiting the ret2ns vulnerabilities in all four cases and both hardware systems. Note that these types of vulnerabilities are inherent in the TrustZone architecture design and are vendor-agnostic.

For instance, Figure 5 depicts a successful execution of a handler-mode-originated attack on the ARM MPS2+ FPGA prototyping board. Prior to the enactment of the `bxns` instruction, the CPU operates in the secure handler mode, denoted by an `IPSR` value of 11. At this juncture, the address within the `LR` register has been manipulated to reflect the attacker-controlled function in the non-secure state. Following the state transition, the `IPSR` register maintains its value of 11,

[2]https://www.microchip.com/en-us/development-tool/microchip-studio
[3]https://www2.keil.com/mdk5/uvision/

facilitating the continued execution of the CPU in the non-secure handler mode, now under the control of the attacker's code.

Because Cortex-M TrustZone is still fairly new and there are few secure state firmware implementations, we haven't found any real-world ret2ns vulnerabilities in production systems. Nevertheless, we have responsibly disclosed ret2ns attacks to ARM.

IV. Defending Against ret2ns Attacks

The key to preventing ret2ns attacks is to make the designated non-secure userspace programs as unprivileged-execution-only with access control mechanisms. On the latest Cortex-M55 and M85 microcontrollers, this can be achieved with negligible overhead by properly setting up the MPU regions with PXN. However, there are two limitations of the PXN approach: (1) Compared to the newly released M55 and M85 cores, the older Cortex-M23, M33, and M35P microcontrollers that hold a comparable larger market share among TrustZone-enabled devices do not have the PXN feature [29], [30]; (2) only a small number of MPU regions, e.g., 8 or 16, are supported, so it is not fine-grained enough for complex RTOSes. To address these issues, we present two mechanisms, namely (i) MPU-assisted address sanitizer and (ii) software-fault isolation based approaches, which can effectively mitigate ret2ns attacks for all Cortex-M microcontrollers with TrustZone.

We only consider the following two control-flow transfers from the secure state to the non-secure state illegal: (1) any return or call from the secure handler mode to a non-secure userspace address, regardless of the non-secure state privilege level; (2) any return or call from the secure thread mode to a non-secure userspace address when the non-secure state is at the privileged level. Other secure to non-secure userspace address control-flow transfers are legal. For example, a non-secure userspace program can call an NSC function, and it is legal for the NSC function to return to a userspace address, as in this case, no privilege escalation will occur.

Our proposed mechanisms instrument destination address sanitizers at two locations: (1) the epilogues of all NSC functions, i.e., before their `bxns` instructions; and (2) before all of the `blxns` instructions in the secure state program.

A. MPU-assisted Address Sanitizer

In the MPU-assisted address sanitizing approach, we assume the non-secure state already adequately implements memory protection mechanisms by configuring userspace and kernel space memory regions with the non-secure MPU. Our sanitizer instruments secure code at the NSC epilogues (before `bxns`) and at `blxns` call sites to validate the destination of every secure to non-secure transfer. At each transfer, the sanitizer inspects the current context (e.g., handler vs. thread mode and the NS privilege bit) to determine whether a jump into NS userspace would imply privileged execution or inherited handler priority. It then queries the NS-MPU attributes of the destination address using the test-target alternate-domain interface and permits the transfer only if the target is not executable by privileged code in NS userspace or is otherwise a legal return path (e.g., `EXC_RETURN` or an NS kernel address). If the policy check fails, execution is halted to prevent cross-state privilege abuse. This approach is not intrusive to the non-secure state userspace and kernel space program since they stay intact and do not need to be re-compiled. Its effectiveness relies on a correct NS-MPU configuration and sufficient region granularity. The detailed algorithm and instruction-level walk-through are provided in the Appendix A.

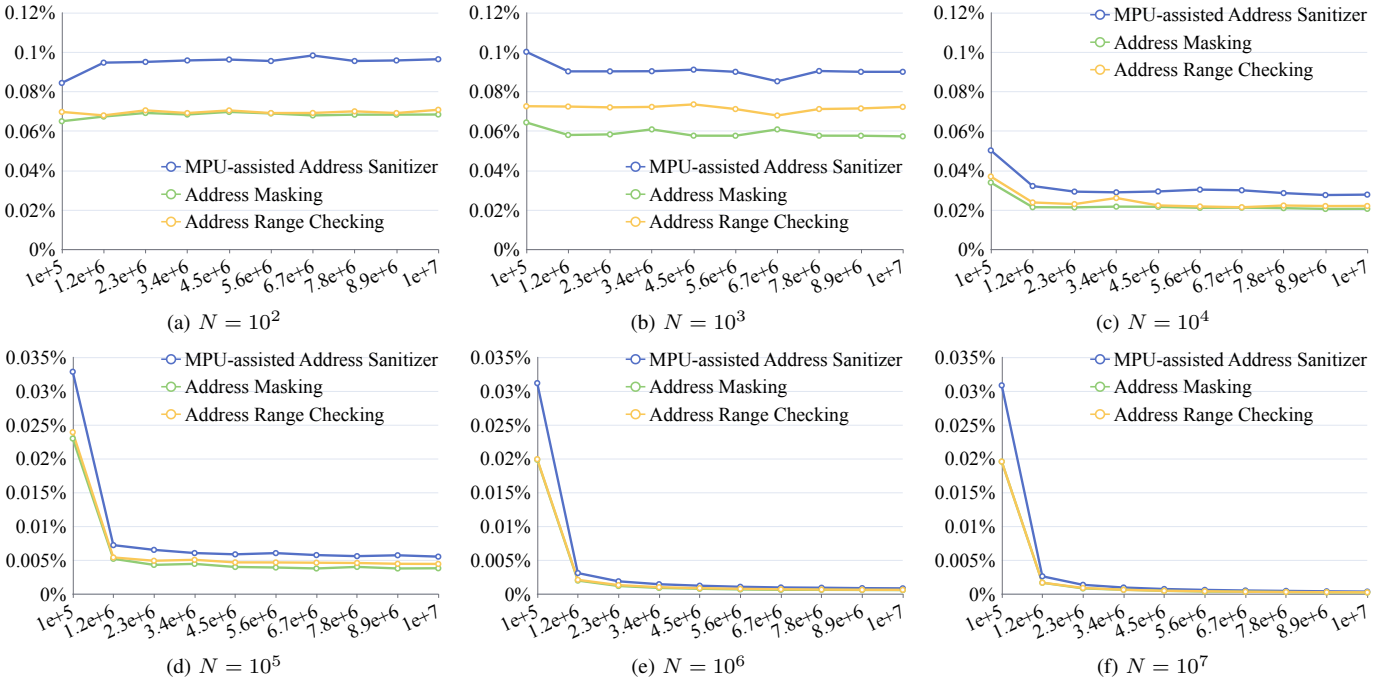
B. Software-fault Isolation

The MPU-assisted address sanitizer is generic but not optimal in terms of efficiency. To address this issue, we also present two more efficient software-fault isolation (SFI) defenses, namely address masking and address range checking. In these approaches, we assume the userspace program and kernel space program are placed in disjoint and known memory regions by the developers.

We instrument secure code at NSC epilogues (before `bxns`) and at `blxns` call sites to constrain the destination of every secure to non-secure transfer to the allowed NS kernel range whenever the NS domain is privileged or inherits handler priority. Unlike the MPU-assisted sanitizer, these checks avoid querying MPU metadata and instead enforce an address policy purely with arithmetic and comparisons, yielding lower constant overhead. SFI does not modify NS binaries and remains transparent to the NS firmware; its main limitation is the requirement for a stable region layout and careful handling of special return encodings.

1) Address Masking: Address masking forces the branch/return target into the designated NS kernel range by applying a bit-wise mask after verifying that the transfer would execute with NS privilege or inherited handler priority. For `bxns` epilogues, the sanitizer must recognize and preserve `EXC_RETURN` encodings that represent a legitimate architectural return path; these values are not masked. For `blxns` calls, `EXC_RETURN` is never valid, so the operand register is unconditionally masked when the policy requires privileged-only targets. Masking is constant-time and incurs the lowest cycle cost among our mechanisms, but it relies on a fixed partitioning of NS user and kernel code.

2) Address Range Checking: Address range checking validates that the destination falls within one of the predeclared NS kernel code intervals by comparing address prefixes (e.g., high-order bits) against the allowed range identifiers. As with masking, the sanitizer first determines whether the transfer would execute with NS privilege or inherited handler priority, and it treats `EXC_RETURN` as a permitted `bxns` return encoding that bypasses range evaluation. Compared to masking, range checking is slightly more general when multiple disjoint kernel ranges must be supported, while retaining a small, constant per-transfer overhead. Both SFI variants stop execution on policy violations to prevent cross-state privilege abuse; detailed algorithms and instruction-level walk-throughs are provided in the Appendix B and B1.



(a) $N = 10^2$ (b) $N = 10^3$ (c) $N = 10^4$

(d) $N = 10^5$ (e) $N = 10^6$ (f) $N = 10^7$

Fig. 6: Defense evaluation results with different N. The x-axis represents T from 10^2 to 10^7, and the y-axis is the overhead compared to the baseline Blinky.

C. Defense Evaluation

We evaluated the effectiveness and performance of the proposed defense mechanisms on the AN505 IoT kit image for the ARM MPS2+ FPGA prototyping board with a Cortex-M33 microcontroller. The Cortex-M33 processor was configured to execute at 20MHz.

1) Effectiveness Evaluation: We applied both the MPU-assisted address sanitizer and the software-fault isolation approaches to the vulnerable projects presented in Section III-F. The experiments confirmed that either of these defense mechanisms can defeat all four ret2ns attack variants. A detailed evaluation case is provided in Appendix C.

2) Costs of the Worst-case Address Sanitizing Paths: The proposed instrumentation in Listing 3, 4, and 5 only cost a small number of CPU cycles. The worst-case experiments, i.e., all instructions in the listings are executed, on the aforementioned Cortex-M device show (1) the MPU-assisted address sanitizer costs 32 CPU cycles for the `bxns` case, and 30 cycles for the `blxns` case; (2) the address masking mechanism costs 18 CPU cycles for `bxns`, and 12 CPU cycles for `blxns`; (3) the address range checking mechanism costs 18 CPU cycles for `bxns`, and 14 CPU cycles for `blxns`.

3) Performance Evaluation Setup: Because there are no benchmarks designed specifically for Cortex-M TrustZone cross-world performance evaluation, our evaluations were based on a modified Blinky application that comes with the Keil IDE. The Blinky application is a cross-world project with both non-secure and secure state programs, and it works on a system with 3 LEDs and a UART peripheral. We enabled the non-secure MPU for all the evaluation experiments.

In the modified Blinky application, the secure and non-secure programs configure the SysTick timer for its corresponding state, respectively, to generate a SysTick interrupt every S ms (around $T = S \times 20 \times 10^3$ CPU cycles). The secure state program provides three NSC functions for (1) switching on an LED; (2) switching off an LED; (3) sending messages to the UART peripheral. All three NSC functions return with `bxns` instructions. The non-secure main program is a loop that calls the three NSC functions to toggle LED-1 and send a message to the UART. There are N `nop` instructions before each NSC function call, and each `nop` instruction consumes one CPU cycle. Thus, there will be $20 \times 10^3/N$ ms delays before each NSC function call. The secure SysTick handler calls two non-secure functions to toggle LED-2 using `blxns` instructions. The non-secure SysTick handler also calls the NSC functions to toggle LED-3. The cross-state calls in the SysTick handlers represent the routine two-way communications between the states, whereas the NSC function calls in the non-secure main loop represent ad hoc service requests. By configuring T and N, we can simulate applications with different state-crossing frequencies. Higher T means less frequent routine communications between the states, and higher N means less frequent service requests from the non-secure state to the secure state.

4) Performance Evaluation Results: We chose 60 pairs of T and N to simulate scenarios with different routine cross-state communication and service request frequencies. For each pair of T and N, we recorded the cost in CPU cycles when the non-secure main loop executes 10 times. We ran each case five times and computed the cost on average.

Figure 6 shows the performance evaluation results. With higher T and N values, the cross-state transitions will be less frequent, thus the overhead introduced by the sanitizing mechanisms will be lower compared with smaller T and

TABLE I: Attack-side comparison across ret2usr-like attacks (ret2ns, ret2usr, ret2dir, and boomerang).

	ret2ns	**ret2usr** [11]	**ret2dir** [12]	**boomerang** [10]
Hardware Target	Microcontroller TEE, i.e., Cortex-M TrustZone.	Microprocessor, e.g., x86, x86-64, Cortex-A.	Microprocessor, e.g., x86, x86-64, Cortex-A.	Microprocessor TEE, i.e., Cortex-A TrustZone.
Software Target	Secure state code pointers used by `bxns` or `blxns` instructions.	Kernel-level code pointers, e.g., return addresses, jump tables, function pointers.	Kernel pointers redirected to the direct-mapped physical memory region (physmap).	Data pointers in the application-specific data structure sent to the secure state.
Threat Model	A non-secure state userspace attacker can interact with the secure state program with supervisor calls.	An adversary can overwrite, partially corrupt, or nullify code pointers in the kernel address space. They can also tamper with entire data structures that contain code pointers.	A userspace attacker exploits a kernel bug to overwrite a kernel pointer and leverages the direct map to reference attacker pages via kernel-space aliases.	The attacker controls a normal state userspace application that can interact with secure applications using system calls.
Attack Goal	The userspace attacker corrupts one code pointer in the secure state to redirect the control flow to attacker-controlled code while keeping the privileged execution mode.	Exploit the kernel vulnerability to corrupt the kernel-level code pointer with userspace addresses.	Bypass ret2usr defenses by keeping control flow in kernel space via the direct map.	The userspace attacker aims to trick the secure state code to read or write non-secure memory it does not own.
Attack Impact	Privilege escalation and arbitrary code execution in the non-secure state at a privileged level.	Redirect the kernel control flow to userspace code.	Full kernel compromise and privilege escalation through code execution or arbitrary read or write.	Bypass the security mechanisms of the untrusted OS via the secure code path and enable unauthorized access to non-secure memory.
Root Cause	The wide state transition surface on TZ-M enables an unlimited number of entries between secure and non-secure privilege levels, and the secure state does not check the destination address when switching to the non-secure state.	In Unix-like OSs that support address virtualization, the kernel address space is shared and mapped to user virtual address spaces.	Kernels permanently map all physical memory into a kernel-only virtual region (direct map), creating kernel aliases of userspace pages.	The pointer sanitization can be bypassed by the userspace application while passing the application-specific data structure. Moreover, the secure application does not validate the pointer passed in this structure.

N values. We have observed that when the value of N is lower ($N \leq 10^4$, figs. 6a to 6c), the overall overhead remains consistent across different T values. This is because the frequent service requests from the non-secure state to the secure state, which occur more often with a lower N value, dominate the evaluation process. This leaves much less time for routine communications, resulting in their impact being less significant. Even with a high cross-state transition frequency, e.g., the SysTick handler performs cross-state function calls every 5 ms ($T = 10^5$), the sanitizing overheads are still negligible.

V. Comparison with Related Work

Confused deputy vulnerabilities come into existence when a higher privileged program is manipulated by a less privileged, but harmful program into misusing its granted authority [6]. In practice, such vulnerabilities have been at the root of some high-profile security incidents, indicating their potential to cause substantial harm and underscoring the need for effective countermeasures [31], [32]. This deception typically originates from a semantic gap between a more privileged program and a less privileged domain. For instance, a hypervisor's view of a virtual machine [33] or the Cortex-A and Cortex-M TrustZone TEE's interpretation of the memory of the rich execution environment often contribute to confused deputy vulnerabilities.

A. Ret2usr, Ret2dir, Boomerang, and Nailgun Attacks

Ret2usr, ret2dir, and boomerang are disclosed confused deputy attacks on microprocessors with MMUs, e.g., x86, Cortex-A, and on modern operating systems, e.g., Linux.

The premise of the ret2usr attack [11] lies in leveraging a legitimate but compromised kernel function, which is manipulated to return not to its original caller in the kernel space, but instead to an attacker-specified location in the user space. This redirection allows malicious user-level code to be executed with the elevated privileges of kernel mode, effectively bypassing the protections inherent in separating user and kernel spaces.

Ret2dir attacks are a form of confused deputy attacks that exploit a feature of modern operating systems: direct kernel mappings of physical memory [12]. These attacks maneuver around protections against ret2usr attacks by redirecting kernel operations not to user space, but to a virtual memory region within the kernel space itself. This region directly maps all or part of the physical memory, allowing for unauthorized access or modifications. Unlike ret2usr, ret2dir attacks bypass traditional defenses, as the malicious code remains within the kernel space.

Boomerang attacks represent a type of confused deputy attacks that target systems implementing Cortex-A TrustZone [10]. These attacks manipulate the TrustZone's secure state applications to gain unauthorized access to restricted memory regions. This is achieved by making a non-secure application request the secure state application to execute an operation that shouldn't be permissible from the non-secure state. In effect, the secure application acts as a "boomerang," returning with sensitive data or causing alterations that the non-secure application should not access or perform.

Nailgun [34], [35] highlights a crucial vulnerability where, during inter-processor debugging mode, the privilege level of the debugging host is disregarded at the debugging target. This loophole allows the low privilege level at the non-secure state on both Cortex-A and Cortex-M TrustZone to access secure state resources via the debug unit.

Comparing with all the ret2usr-like attacks in Table I, the ret2ns attack introduces a specific challenge within the ARM Cortex-M TrustZone, distinct from the ret2usr, ret2dir, and boomerang attacks that target different architectures and ex-

TABLE II: Comparative evaluation of defenses against cross-state control-flow abuse and related ret2usr-like attacks.

Metric	MICROFT		PXN (Privileged eXecute–Never)	kGuard (ret2usr) [11]	XPFO (ret2dir) [12]	Boomerang cooperative check [10]
	MPU-assisted Address Sanitizer	SFI: Address Masking / Range Checking				
Target arch. & defense preconditions	Armv8–M TZ–M; requires NS–MPU policy and `tta`.	Armv8–M TZ–M; assumes separated, known NS user/kernel code regions.	Armv8.1–M with MPU PXN (e.g., M55/M85); mark NS user regions PXN.	Linux (x86/x86–64) with MMU; compile-time instrumentation.	Linux (x86/ARM) with MMU; page-frame ownership enforcement.	Cortex–A TEEs; secure apps cooperate with REE callback for access checks.
What it blocks (defense focus)	Secure→Non-secure transfers landing in NS *userspace* while privileged (ret2ns via `bxns`/`blxns`).	Same as left (ret2ns via `bxns`/`blxns`).	Any privileged fetch from PXN-marked NS user code (generic ret2ns prevention).	Kernel→userspace returns (ret2usr).	ret2dir via kernel direct-map (*physmap*) abuse.	Secure app data-path misuse against REE memory (Boomerang).
Enforcement point / mechanism	Per-transfer check before `bxns`/`blxns` using NS–MPU attributes; deny if dest lacks unprivileged exec permissions.	Per-transfer masking or range test to force targets into allowed privileged region; preserve valid `EXC_RETURN` in `bxns` epilogues.	Hardware MPU attribute; privileged fetch in PXN region faults before execution.	Inline guards on indirect branches ensure targets remain in kernel space.	Unmap user pages from kernel direct map; remap on reclaim; exclusive ownership.	Per-access policy check via non-secure callback before dereferencing REE pointers.
Reported runtime cost	30-32 cycles per transfer (worst case); end-to-end $< 0.1\%$ (Section IV-C).	12-18 cycles per transfer (worst case); end-to-end $< 0.1\%$ (Section IV-C).	Near-zero steady-state; traps only on violations.	$\sim$10-11% on syscall and I/O microbenchmarks; negligible on CPU-bound apps; modest kernel size increase.	Minimal/negligible in typical workloads.	Workload-dependent; proportional to cross-world validations.
Deployment complexity	**Medium**: instrument NSC epilogues and `blxns` sites; no NS code changes.	**Low-Medium**: lightweight instrum.; requires fixed code layout.	**Low** if hardware present; configure NS–MPU regions.	**Medium**: rebuild kernel with instrumentation.	**Medium-High**: kernel memory-manager changes.	**Medium**: modify secure apps to validate every REE pointer access.
Usable on TZ–M today?	**Yes** (M23/M33/M35P/ M55/M85).	**Yes**.	**Partial**: only where PXN exists (M55/M85; not M23/M33/M35P).	**No**.	**No**.	**No** (not a drop-in for TZ–M; different stack/problem).
Key limitations / notes	Depends on correct NS–MPU policy; limited MPU regions available on chip.	Not fully generic (layout assumption); extra handling of `EXC_RETURN` in `bxns` epilogues.	Availability gap on deployed MCUs; limited MPU regions available on chip.	Different threat model; assumes VA split & MMU; inapplicable to secure→non-secure transfers on TZ–M.	Prevents direct-map abuse; unrelated to `bxns`/`blxns` cross-state execution on TZ–M.	Stops data-access confused-deputy; *does not* address secure→non-secure execution transfers.

Notes: PXN availability and semantics are based on Armv8.1–M and Cortex–M55/M85 documentation [18]. Features for kGuard, XPFO, and Boomerang are summarized from their respective papers [10]–[12].

ploit different vulnerabilities. *The root cause of ret2ns is the exploitation of the extensive state transition surface in Cortex-M TrustZone, due to the architecture's lack of destination address validation during state transitions.* This differs from the bypass of pointer sanitization in boomerang and the exploitation of shared kernel-user address space in Unix-like systems by ret2usr, underscoring the unique implications and characteristics of the ret2ns attack within the domain of the confused deputy attacks.

B. Existing Mitigation of Ret2usr-like Attacks

Table II summarizes and compares existing defenses against ret2usr-like attacks and our proposed defenses against ret2ns attacks.

To mitigate ret2usr attacks, kGuard [11] instruments run-time control-flow checks to verify the indirect branch target is always in kernel space and enforces lightweight address space segregation. To patch the ret2dir vulnerability, XPFO [12] uses an exclusive ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory. To thwart boomerang attacks, a cooperative approach [10] requires that all of the non-secure memory accesses from the secure state need to query a non-secure callback function to verify the access permission of the referenced memory region.

Note that recent efforts on securing cross-state communication on Cortex-M TrustZone [36], [37] can only guarantee the confidentiality and integrity of the messages sent between the two states and authenticate the communicating parties. The aforementioned confused deputy attacks, as well as the proposed ret2ns attacks do not rely on tampering cross-state messages nor impersonating any communicating parties; hence, these attacks cannot be defeated by secure communication mechanisms.

VI. LIMITATION AND DISCUSSION

a) Generalizability to Other TEEs and Embedded Platforms: The ret2ns attack relies on three properties: (i) a direct, monitor-bypass cross-state transfer from secure to non-secure code (e.g., `bxns`/`blxns`) rather than a centralized monitor path, (ii) a corruptible secure-side control data consumed at the transfer (saved `LR` for `bxns` or a function pointer/register for `blxns`), and (iii) preservation or reestablishment of non-secure privileged execution after the switch. These conditions hold generically on Armv8-M TrustZone, independent of RTOS choice or secure middleware, so ret2ns is broadly applicable across Cortex-M deployments. By contrast, on Cortex-A TrustZone, cross-world transitions are mediated by the secure monitor and a structured exception-return path, which prevents arbitrary branch targets and does not inherit secure-handler privilege into the normal world; thus ret2ns does not directly apply. Other TEEs with monitor-mediated world switches (e.g., enclave systems or hypervisor-based separation) similarly lack a direct branch-and-return primitive across domains and therefore do not exhibit the same attack surface. On non-TrustZone microcontrollers that implement

privilege overlays or SVC-based separation, an analogous vector would require a privileged component that returns into unprivileged address ranges without execute-never enforcement and with a corruptible return pointer; such designs are uncommon and typically drop privilege on return, reducing feasibility. Finally, hardware policies that forbid privileged execution in user regions (e.g., PXN on Armv8.1-M) eliminate the core precondition of ret2ns by making secure-to-non-secure transfers into non-secure userspace non-executable at privileged level.

b) Limitations of the Proposed Defenses: In evaluating our defense strategies, we incorporated inline assembly directly at the instrument point within the source code. The intention behind this was not only to validate the effectiveness of our proposed defense mechanisms but also to underscore the minimal performance overhead. Further, it's worth noting that the efficacy of these defense mechanisms could be potentially enhanced with compiler or binary level instrumentation, for instance, through the utilization of LLVM passes or binary rewriting.

c) Operational and Business Impact: Beyond privilege escalation and arbitrary code execution, the practical impact of ret2ns stems from its disruption of system functions and its effect on confidentiality guarantees. We analyze impacts along two axes: (i) control-plane determinism and availability, and (ii) data-plane confidentiality and integrity, then highlight domain-specific consequences.

Control-plane determinism and availability. ret2ns enables attacker-controlled code to run at non-secure privileged level or with inherited handler priority. This allows adversaries to preempt time-critical tasks, suppress interrupts, or tamper with scheduler and timer configurations. Such interference can break hard real-time guarantees, where missed deadlines directly translate into correctness failures with physical consequences in cyber-physical systems [38]–[40]. Examples include delayed actuation, suppression of safety alarms, or altered setpoints, all of which undermine predictable and safe control.

Data-plane confidentiality and integrity. ret2ns can also compromise the integrity of cryptographic services that embedded stacks delegate to secure partitions [41]. Although secure memory remains isolated, attackers can manipulate cryptographic APIs into misusing keys (e.g., signing or decrypting attacker-controlled data), or they can exploit privileged non-secure direct memory access (DMA) and drivers to capture plaintext before encryption or divert ciphertext [42], [43]. Such attacks erode system trust assumptions about confidentiality, authenticity, and secure bootstrapping.

Domain-specific consequences. According to the recent market analysis, embedded systems with dedicated hardware security features like ARM TrustZone-M start to emerge in various sectors, including industrial control, medical devices, automotive ECUs, and consumer IoT [44]–[46]. In **industrial control systems**, ret2ns can disable protective interlocks, delay emergency trips, or inject false actuation commands. Such behavior mirrors past incidents where adversaries manipulated safety instrumented systems, creating risks of physical damage and process hazards [47], [48]. In **medical devices**, attacker-controlled privileged execution can alter monitoring parameters, suppress alarms, or delay therapy delivery, which regulators explicitly recognize as cybersecurity-linked safety risks [49]. In **automotive ECUs**, attackers may modify braking or steering logic timing, bypass in-vehicle network isolation, or misconfigure sensors, raising both safety and liability concerns [50]. In **consumer IoT and smart home systems**, abuse of privileged code can expose sensitive user data, disable security sensors, or subvert trusted peripherals, leading to privacy loss and loss of consumer trust [51]. Across domains, these technical failures propagate into downtime, safety incidents, compliance violations, and costly recalls, underscoring the broader business relevance of mitigating cross-state privilege abuse.

d) Implications for ARM and Embedded Security Standards: MICROFT highlights a systematic gap in cross-state control-flow validation on Armv8-M, suggesting that future architectural and profile guidance should require privileged execute-never (PXN-equivalent) or an enforceable policy that forbids privileged fetches from non-secure user regions after secure-to-non-secure transfers. At the software standard layer, CMSIS [52] and TrustedFirmware-M [53] guidelines could mandate NSC epilogue and `blxns` pre-call target checks, provide reference instrumentation, and add conformance tests that verify no secure component can return or call into non-secure userspace while the non-secure domain is privileged or in an inherited handler context. PSA Certified threat models and evaluation criteria [54], [55] can incorporate ret2ns as an explicit abuse case, requiring evidence of MPU policy that separates non-secure user and kernel code, default-on PXN where available, and hardening rules for NSC functions. For domain standards (e.g., industrial, medical, automotive), certification baselines can include cross-state privilege-preservation analysis and test vectors that exercise `bxns`/`blxns` boundaries under fault injection and memory-corruption scenarios, ensuring that timing, safety, and cryptographic services do not become confused deputies across the state boundary. Collectively, these changes would convert MICROFT's findings into actionable architectural requirements, vendor guidance, and compliance checks that reduce the feasibility of ret2ns-style abuse on embedded platforms.

VII. CONCLUSION

ARM Cortex-M is the most popular 32-bit microcontroller architecture in the market with unique performance optimization from its microprocessor counterparts. However, the security implication of its optimization has not been thoroughly studied. In MICROFT, we took a close look at the fast state switch mechanism of Cortex-M TrustZone, and we presented ret2ns, an exploitation technique that takes advantage of the fast state switch mechanism to perform arbitrary code execution with escalated privilege. The vulnerability arises from the inherent semantic gap and the wide transition surface between the security states. We demonstrated the detailed methodology of ret2ns attacks with two detailed cases and confirmed the

feasibility of all four variants of attacks on two hardware platforms. To defeat ret2ns attacks, we designed and evaluated two address sanitizing mechanisms with negligible runtime overhead.

ACKNOWLEDGEMENTS

This material is based upon work supported in part by National Science Foundation (NSF) grants (2512972, 2453496, 2523436, and 2508320) and a National Centers of Academic Excellence in Cybersecurity (part of the National Security Agency) grant (H98230-22-1-0307).

REFERENCES

[1] "The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter." https://newsroom.arm.com/news/the-arm-ecosystem-ships-a-record-6-7-billion-arm-based-chips-in-a-single-quarter.

[2] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, 2019.

[3] Arm, "Armv8-M Architecture Reference Manual." https://developer.arm.com/documentation/ddi0553/latest?_ga=2.1957362.2138159006.1623856318-792272022.1611588763.

[4] J. Yiu, "ARMv8-M architecture technical overview," *ARM white paper*, 2015.

[5] Arm, "TrustZone technology for the Armv8-M architecture Version 2.1." https://developer.arm.com/documentation/100690/latest/.

[6] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, 1988.

[7] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Network and Distributed System Security (NDSS)*, 2012.

[8] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[9] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *USENIX Security Symposium*, 2020.

[10] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *Network and Distributed System Security (NDSS)*, 2017.

[11] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-User Attacks," in *USENIX Security Symposium*, 2012.

[12] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security Symposium*, 2014.

[13] Arm, "Cortex-M23 Technical Reference Manual." https://developer.arm.com/documentation/ddi0550/.

[14] Arm, "Cortex-M33 Technical Reference Manual." https://developer.arm.com/documentation/100230/.

[15] Arm, "Cortex-M35P." https://developer.arm.com/ip-products/processors/cortex-m/cortex-m35p.

[16] Arm, "Cortex-M55 Technical Reference Manual." https://developer.arm.com/documentation/101051/.

[17] Arm, "Cortex-M85 Technical Reference Manual." https://developer.arm.com/documentation/101924/.

[18] "Armv8.1-M Architecture Reference Manual." http://kib.kiev.ua/x86docs/ARM/ARMARMv8m/DDI0553B_k_armv8m.pdf.

[19] J. Yiu, *Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors*. Newnes, 2020.

[20] "FreeRTOS." https://www.freertos.org/.

[21] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *IEEE Symposium on Security and Privacy*, 2017.

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

[23] Arm, "Trusted Firmware-M." https://developer.arm.com/Tools%20and%20Software/Trusted%20Firmware-M.

[24] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu, "On Runtime Software Security of TrustZone-M Based IoT Devices," in *IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2020.

[25] L. Luo, Y. Zhang, C. White, B. Keating, B. Pearson, X. Shao, Z. Ling, H. Yu, C. Zou, and X. Fu, "On security of trustzone-m-based iot systems," *IEEE Internet of Things Journal*, 2022.

[26] X. Tan, Z. Ma, S. Pinto, L. Guan, N. Zhang, J. Xu, Z. Lin, H. Hu, and Z. Zhao, "SoK: Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems," in *18th USENIX WOOT Conference on Offensive Technologies*, 2024.

[27] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, "HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 1082–1099, IEEE, 2022.

[28] Arm, "AN505: Cortex-M33 with IoT kit FPGA for MPS2+ Version 2.0." https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/download-fpga-images.

[29] S. M. Khan, D. Peterson, and A. Mann, "The Semiconductor Supply Chain," *Center for Security and Emerging Technology*, 2021.

[30] R. Sharma, "ARM Cortex-M55 Helium DSP MCU Market Research Report 2033." https://growthmarketreports.com/report/arm-cortex-m55-helium-dsp-mcu-market.

[31] V. Rajani, D. Garg, and T. Rezk, "On access control, capabilities, their equivalence, and confused deputy attacks," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, IEEE, 2016.

[32] S. Brookes and S. Taylor, "Containing a confused deputy on x86: a survey of privilege escalation mitigation techniques," *International Journal of Advanced Computer Science and Applications*, 2016.

[33] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "SoK: Introspections on Trust and the Semantic Gap," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[34] Z. Ning and F. Zhang, "Understanding the Security of Arm Debugging Features," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[35] E. Perla and M. Oldani, *A guide to kernel exploitation: attacking the core*. Elsevier, 2010.

[36] A. K. Iannillo, S. Rivera, D. Suciu, R. Sion, and R. State, "An REE-independent Approach to Identify Callers of TEEs in TrustZone-enabled Cortex-M Devices," in *ACM Cyber-Physical System Security Workshop (CPSS)*, 2022.

[37] A. Khurshid, S. D. Yalew, M. Aslam, and S. Raza, "ShieLD: Shielding Cross-zone Communication within Limited-resourced IoT Devices running Vulnerable Software Stack," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2022.

[38] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein, "Control-system stability under consecutive deadline misses constraints," in *32nd euromicro conference on real-time systems (ECRTS 2020)*, pp. 21–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[39] A. Cervin, "Analyzing the effects of missed deadlines in control systems," in *ARTES Real-Time Graduate student conference*, 2001.

[40] K. Stouffer, K. Stouffer, M. Pease, C. Tang, T. Zimmerman, V. Pillitteri, S. Lightman, A. Hahn, S. Saravia, A. Sherule, *et al.*, "Guide to Operational Technology (OT) Security," *NIST Special Publication 800-82, Revision 3*, 2023.

[41] A. de Angelis, "TF-M Crypto Service design." https://trustedfirmware-m.readthedocs.io/en/latest/design_docs/services/tfm_crypto_design.html.

[42] J. D. B. Lavernelle, P.-F. Bonnefoi, B. Gonzalvo, and D. Sauveron, "Dma: A persistent threat to embedded systems isolation," in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 101–108, IEEE, 2024.

[43] A. Trikalinou and D. Lake, "Taking DMA attacks to the next level," *BlackHat USA*, pp. 22–27, 2017.

[44] Renesas, "Renesas Teams Up with Applus+ Laboratories to Achieve PSA Certified Level 1 with CRA Extension for Three New MCU Groups." https://www.renesas.com/en/about/newsroom/renesas-teams-applus-laboratories-achieve-psa-certified-level-1-cra-extension-three-new-mcu-groups.

[45] P. Valerio, "IoT Security is at its highest demand; PSA Certified reports find." https://www.eetimes.com/iot-security-is-at-its-highest-demand-psa-certified-reports-find.

[46] S. Sinha, "Cellular IoT module market Q2 2023: 66% of IoT modules shipped without dedicated hardware security." https://iot-analytics.com/cellular-iot-module-security.

[47] A. Di Pinto, Y. Dragoni, and A. Carcano, "Triton: The first ics cyber attack on safety instrument systems," *Proc. Black Hat USA*, vol. 2018, pp. 1–26, 2018.

[48] M. Geiger, J. Bauer, M. Masuch, and J. Franke, "An analysis of black energy 3, crashoverride, and trisis, three malware approaches targeting operational technology systems," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 1537–1543, IEEE, 2020.

[49] Food, D. Administration, *et al.*, "Cybersecurity in medical devices: quality system considerations and content of premarket submissions," *Guidance for Industry and Food and Drug Administration Staff*, 2023.

[50] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *black hat USA*, vol. 2014, no. 2014, p. 94, 2014.

[51] Z. Wang, D. Liu, Y. Sun, X. Pang, P. Sun, F. Lin, J. C. Lui, and K. Ren, "A survey on iot-enabled home automation systems: Attacks and defenses," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2292–2328, 2022.

[52] Arm, "Common Microcontroller Software Interface Standard (CMSIS)." https://www.arm.com/technologies/cmsis.

[53] Linaro, "Trusted Firmware-M (TF-M)." https://www.trustedfirmware.org/projects/tf-m/.

[54] GlobalPlatform and Arm, "Platform Security Architecture (PSA) Certified." https://www.psacertified.org/.

[55] F. Chen, D. Luo, J. Li, V. C. Leung, S. Li, and J. Fan, "Arm psa-certified iot chip security: a case study," *Tsinghua Science and Technology*, 2022.

APPENDIX

A. MPU-assisted Address Sanitizer

Our instrumentation examines the access permissions of the non-secure destination address by querying the MPU settings.

Listing 3 shows the example instrumentation before a `bxns` or `blxns` instruction. We assume the destination address of a `blxns` is already loaded in `R0` or the return address for a `bxns` will be loaded in `R0`, as shown in line 1. The instrumentation first checks the values of the `IPSR` register and the `nPRIV` bit of non-secure `CONTROL_NS` register (line 2 and 4-5). If `IPSR` is zero (line 3) or `CONTROL_NS.nPRIV` equals one (line 6), the control transfer is legal. Otherwise, it uses the test target alternate domain instruction (`tta`) to check the MPU attribute of the destination address and save the result

```
1  ldr   R0, [SP, #12]  ; only need for bxns
2  mrs   R3, IPSR       ; read IPSR value
3  cbnz  R3, #6         ; check if IPSR is zero
4  mrs   R3, CONTROL_NS ; read CONTROL_NS
5  lsls  R3, R3, #31    ; get CONTROL_NS.nPRIV bit
6  bne   #28            ; check if nPRIV bit zero
7  tta   R0, R0         ; test target on target addr
8  lsls  R3, R0, #15    ; get MRVALID bit
9  bpl   #20            ; check if region is valid
10 uxtb  R0, R0         ; get MPU region number
11 movw  R3, #0xED98    ; load MPU_NS->RNR addr
12 movt  R3, #0xE002    ; 0xE002ED98
13 str   R0, [R3, #0]   ; set MPU_NS->RNR to region
14 ldr   R0, [R3, #4]   ; get MPU_NS->RBAR
15 lsls  R0, R0, #30    ; get UP read permission bit
16 beq   #2             ; check UP read permission
17 cpsid i              ; disable IRQ
18 b     .              ; error handling
```

Listing 3: MPU-assisted Address Sanitizer

```
1  mrs    R2, IPSR       ; read IPSR
2  cbnz   R2, #6         ; check if IPSR is zero
3  mrs    R2, control_ns ; read CONTROL_NS
4  lsls   R2, R2, #31    ; get CONTROL_NS.nPRIV bit
5  bne    #12            ; check nPRIV bit
6  ldr    R1, [SP, #4]   ; get dest addr (not for blxns)
7  cmn    R1, #0x100     ; Is EXC_RETURN? (not for blxns)
8  itt    cc             ; not EXC_RETURN (not for blxns)
9  movtcc R1, #0x21      ; address masking
10 strcc  R1, [SP, #4]   ; store result (not for blxns)
```

Listing 4: Address Masking. Instructions that are marked with *(not for blxns)* are not used in masking the `blxns` destination address.

into `R0` (line 7). From the result, the procedure can acquire (1) whether the destination address is within a software-defined MPU region rather than the background region using the `MRVALID` bit (line 8); and (2) the corresponding MPU region number using the `MREGION` field (line 10). If the MPU region number is invalid, it means the destination address is either in the default kernel space background region or the `EXC_RETURN` value from the NSC's return address, both of which are legal (line 9). Otherwise, the procedure loads the `MPU_NS.RNR` register address `0xE002ED98` into `R3` (line 11-12), and assign the region number to the `MPU_NS.RNR` register (line 13) to retrieve the MPU attributes on this region. From the MPU attributes (line 14), the second bit in the `MPU_RBAR` register represents the unprivileged read permission for this region (line 15). The next step is to check whether this bit is set, which reflects the unprivileged execution permission on the destination address (line 16). If it is set, a ret2ns attack is detected, and the execution will be stopped (line 17); otherwise, the destination is legal, and the execution continues.

B. Address Masking

In this approach, the instrumentation simply performs a bit-wise masking on the destination address after checking the non-secure privilege level to force the destination address to fall into the allowed address range.

A valid destination address of a `bxns` instruction could be `EXC_RETURN` (`0xFFFFFF**`), so the address masking mechanism should take care of this case and does not mask an `EXC_RETURN` value for `bxns`. This could happen in the following scenario, which we found in the code generated by the ARMClang compiler. In the non-secure handler mode, `LR`

```
1  mrs    R2, IPSR          ; read IPSR
2  cbnz   R2, #6            ; check if IPSR is zero
3  mrs    R2, control_ns    ; read CONTROL_NS
4  lsls   R2, R2, #31       ; get CONTROL_NS.nPRIV bit
5  bne    #18               ; check nPRIV bit
6  ldr    R1, [SP, #4]      ; get dest addr (not for blxns)
7  cmn    R1, #0x100        ; Is EXC_RETURN? (not for blxns)
8  bcs    #10               ; not EXC_RETURN (not for blxns)
9  lsr    R2, R1, #16       ; take top 2 bytes of dest addr
10 cmp    R2, #0x21         ; address range checking
11 beq    #2                ; dest within range
12 cpsid  i                 ; not within range, disable IRQ
13 b      .                 ; error handling
```

Listing 5: Address Range Checking. Instructions that are marked with *(not for blxns)* are not used in masking the `blxns` destination address.

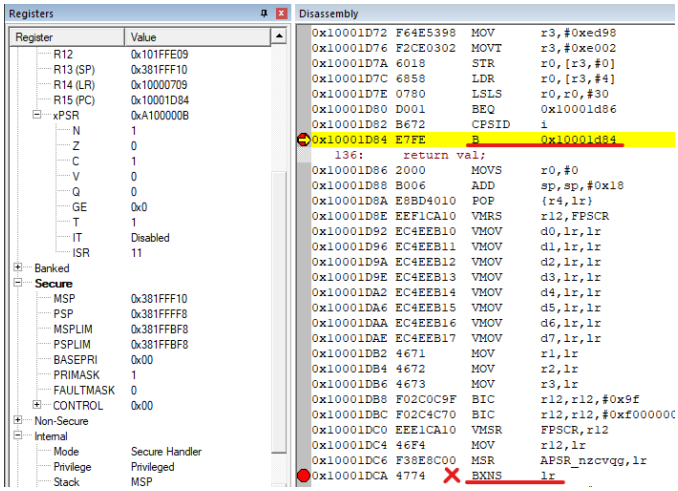
has the value of `EXC_RETURN`. The non-secure handler uses the `bx` instruction to branch to an NSC function. As a result, `LR` and the return address of the NSC are still the value of `EXC_RETURN` after the state switch. Note that `EXC_RETURN` should never be a valid destination address for `blxns`.

Listing 4 presents an example address masking instrumentation for both `bxns` and `blxns` cases. Same as in the MPU-assisted address sanitizer, the initial process involves determining the non-secure privilege level by examining `IPSR` and `CONTROL_NS` (line 1-5). If the non-secure state is at the unprivileged level, the control flow is legitimate. Otherwise, the algorithm verifies if the return address is `EXC_RETURN` for the `bxns` case as aforementioned (line 7). If the return address is not `EXC_RETURN` or it is for the `blxns` case, a bit-wise masking operation is instituted to ensure the resulting address falls into the designated kernel space region (line 9). In the `bxns` case, after sanitizing the return address, it is repositioned back to the stack for the original epilogue to use (line 10).

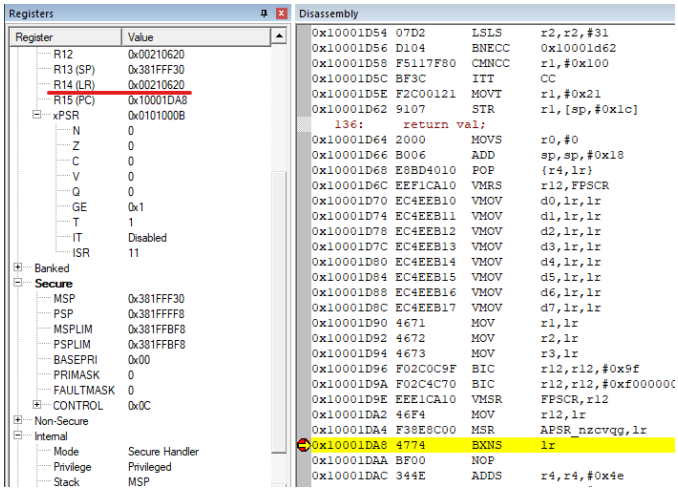
1) Address Range Checking: Address range checking is a fast and more fine-grained software-fault isolation mechanism. The instrumentation performs range checking on the destination address since we already know the address range of the program code range in each privilege level, which has been placed separately in compile time. This allows the program to catch any violations that occur during the range checking process.

Listing 5 demonstrates the instrumented code used for address range checking, following the consistent methodology with earlier outlined procedures. Initially, the code assesses the non-secure state's privilege level by checking the `IPSR` and `CONTROL_NS` registers (lines 1-5). After this preliminary check, it examines if the destination address of the `bxns` instruction corresponds to the `EXC_RETURN` value (lines 6-7). If the address matches `EXC_RETURN`, the verification process stops, and normal execution resumes.

If the destination address does not match the `EXC_RETURN` value, the code proceeds to the address range check. It isolates the upper two bytes of the destination address (line 9) and compares them against the predefined range threshold (line 10). If this comparison fails, it indicates a potential ret2ns attack, triggering a protective trap to interrupt the execution (lines 11-13). Conversely, if the comparison succeeds, it confirms the execution's integrity and allows normal operations to continue.



(a) Utilizing a MPU-assisted address sanitizer, the execution will be halted if the destination address lacks unprivileged execution permissions, thereby preventing the control flow from entering the cross-state transition routine.



(b) Employing an address masking strategy ensures that the bit-wise masking operation invariably confines the address used for cross-state transitions to the specified region within the kernel space.

Fig. 7: Defense effectiveness evaluation of the handler-mode-originated attack. Showing the debugger with Keil IDE running on the ARM MPS2+ FPGA prototyping board with a Cortex-M33 MCU.

C. Defense Effectiveness Evaluation Example

Figure 7 demonstrates the effectiveness of two defense mechanisms on the handler-mode-originated attack on the ARM MPS2+ FPGA prototyping board.

Zheyuan Ma is currently pursuing his Ph.D. degree in Computer Science and Engineering at the University at Buffalo, NY, U.S. His research focuses on system security, with a special emphasis on embedded systems. He received his M.S. in Engineering Science (Robotics) from the University at Buffalo, NY, U.S., and his B.E. in Software Engineering from Fuzhou University, Fujian, China. He has also participated in various team projects, such as the CyberTruck Challenge, MITRE eCTF, and Baidu AutoDriving CTF, showcasing his skills in embedded security, computer vision, and robotics.

Xi Tan is an Assistant Professor in the Department of Computer Science at the University of Colorado Colorado Springs (UCCS) and the director of the secure and reliable system research lab (SUNRISE). She received her PhD in Computer Science from the University at Buffalo. Her research focuses on embedded system security, dynamic analysis techniques, and human factors in cybersecurity. She has published in leading venues, including ACM CCS, RTAS, and DAC.

Lukasz Ziarek is an Associate Professor in the Department of Computer Science and Engineering at the University at Buffalo, The State University of New York, Buffalo, NY, USA. He earned his B.S. degree in computer science from the University of Chicago, Chicago, IL, USA, and his Ph.D. degree from Purdue University, West Lafayette, IN, USA. His research interests encompass programming languages, software engineering, and real-time systems, with a specific focus on language, compiler, and runtime design for achieving reliability. Throughout his career, he has received several prestigious awards, including the NSF CAREER Award (2018), University at Buffalo Exceptional Scholar: Young Investigator Award (2018), University at Buffalo Award for Teaching Innovation (2017), SEAS Early Career Teacher of the Year (2016), among others. His research has been supported through eight NSF grants. He currently servers as the co-director of undergraduate studies in the department of computer science and engineering and is an Equity, Diversity, Justice, and Inclusion fellow for the School of Engineering and Applied Sciences.

Ning Zhang received the Ph.D. degree from Virginia Tech in 2016. He is currently an Assistant Professor leading the the Laboratory of Computer Security and Privacy (CSPL), in the Department of Computer Science and Engineering, at Washington University in St. Louis. His research interest is cyber-physical security.

Shambhu Upadhyaya is a Professor of Computer Science and Engineering at the State University of New York at Buffalo where he also directs the Center of Excellence in Information Systems Assurance Research and Education (CEISARE), designated by the National Security Agency. Prior to July 1998, he was a faculty member at the Electrical and Computer Engineering department. His research interests are in broad areas of information assurance, computer security, behavioral biometrics authentication, and fault tolerant computing. He has authored or coauthored more than 300 articles in refereed journals and conferences in these areas. His research has been supported by the National Science Foundation, U.S. Air Force Research Laboratory, the U.S. Air Force Office of Scientific Research, DARPA, and National Security Agency. He is a Fellow of the IEEE.

Hongxin Hu is an Associate Professor at the Department of Computer Science and Engineering (CSE) at University at Buffalo. His research spans security, networking, and machine learning, reflecting the interdisciplinary nature of his work. His research has been funded by the U.S. National Science Foundation (NSF), the U.S. Department of Transportation, the National Security Agency, VMware, Amazon, Google, Dell, and others. He has received the prestigious NSF CAREER Award, as well as the Amazon Research Award. His research outcomes have been published in renowned conferences, including CCS, USENIX Security, NDSS, SIGCOMM, NSDI, NeurIPS, ICML, EMNLP, CHI, and CSCW. He has also received best paper awards from ACM ASIACCS 2022, ACSAC 2020, IEEE ICC 2020, ACM SIGCSE 2018, and ACM CODASPY 2014, in addition to winning the First Place Award in ACM SIGCOMM 2018 SRC. His research has been featured by the IEEE Special Technical Community on Social Networking and has received wide press coverage, including mentions in ACM TechNews, InformationWeek, Slashdot, and NetworkWorld.

Ziming Zhao is an Associate Professor at the Khoury College of Computer Sciences and the director of the CyberspACe securiTy and forensIcs lab (CactiLab) at Northeastern University, Boston, USA. His current research interests include systems and software security, network and web security, and human-centric security. His research has been supported by the U.S. National Science Foundation (NSF), the U.S. Department of Defense, the U.S. Air Force Office of Scientific Research, and the U.S. National Centers of Academic Excellence in Cybersecurity (part of the National Security Agency). He is a recipient of an NSF CAREER award and an NSF CRII award. His research outcomes have appeared in IEEE S&P, USENIX Security, ACM CCS, NDSS, ACM MobiSys, ACM/IEEE DAC, IEEE RTAS, ACM TISSEC/TOPS, IEEE TDSC, IEEE TIFS, etc. He is a recipient of the Test-of-Time paper award at ACM SACMAT 2024 and best/distinguished paper awards from several prestigious conferences, including USENIX Security 2019, ACM AsiaCCS 2022, ACM CODASPY 2014, and ITU Kaleidoscope 2016. He earned his Bachelor's and Master's degrees from the Beijing University of Posts and Telecommunications in 2006 and 2009, respectively, followed by a Ph.D. in Computer Science from Arizona State University, Tempe, AZ, in 2014.