Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense

Zheyuan Ma[†‡], Xi Tan[†‡], Lukasz Ziarek[†], Ning Zhang[*], Hongxin Hu[†], Ziming Zhao[†‡]
[†]University at Buffalo [‡]CactiLab [*]Washington University in St. Louis
{zheyuanm, xitan, lziarek, hongxinh, zimingzh}@buffalo.edu, zhang.ning@wustl.edu

***Abstract*—ARM Cortex-M is one of the most popular microcontroller architectures designed for embedded and Internet of Things (IoT) applications. To facilitate efficient execution, it has some unique hardware optimization. In particular, 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 demonstrate how this 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 experimentally confirmed the feasibility of four variants of ret2ns attacks on two Cortex-M hardware systems. To defend against ret2ns attacks, we design two address sanitizing mechanisms that have negligible performance overhead.**

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 protection unit (MMU) and targets 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.

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, 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 [3], [4], where the secure state program can be tricked or exploited by a non-secure state program into misusing its authority or ability [5], [6]. Boomerang [7] 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.

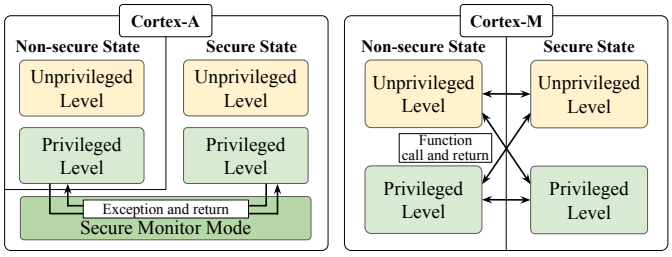

Figure 1: State switches on Cortex-A TrustZone must go through the privileged secure monitor mode, whereas state switches on Cortex-M can occur between different secure and non-secure privilege levels.

In this paper, we report a new class of confused deputy attacks, namely return-to-non-secure attacks (ret2ns), that exploit the fast state switch mechanism of Cortex-M TrustZone. More dangerous than boomerang, ret2ns can lead to arbitrary code execution with escalated privilege in the non-secure state. Ret2ns is a new type of return-to-user (ret2usr) attacks [8], [9] that redirect compromised secure state pointers to code residing in non-secure state userspace. The wide state transition surface on Cortex-M also makes the exploitation of ret2ns vulnerabilities easier than exploiting ret2usr on x86 or boomerang on Cortex-A. Ret2ns attacks affect all Cortex-M processors with TrustZone. 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 [8], ret2ns can be thwarted by preventing control-flow transfers from non-secure state to secure state userspace by using privileged execute-never (PXN) feature. However, PXN is not available on most Cortex-M processors. Only the planned Cortex-M55 and M85 will support this feature [10]. 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 present a detailed methodology for four variants of ret2ns attacks and experimentally evaluate the effectiveness of them on Cortex-M23 and M33 systems;
- We present two address sanitizing mechanisms to mitigate ret2ns attacks with negligible runtime overhead.
- We open-source our project[1], which includes vulnerable code examples, proof-of-concept exploits, and defense instrumentation.

II. Background and Related Work

A. Cortex-M and TrustZone

All Cortex-M processors with the TrustZone extension have thread and handler execution modes [11]. They also have privileged (kernel space) and unprivileged (userspace) levels. The current mode and privilege level are determined by the combination of the interrupt program status register (`IPSR`) and the `CONTROL` register. `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 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 will automatically perform unstacking, which pops the eight registers off the stack.

TrustZone adds another orthogonal partitioning of states. 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. 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.

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 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.

B. Ret2usr, Ret2dir, and Boomerang 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. Ret2usr attacks [8] exploit bugs in the kernel space and redirect the data or control flow to the data or code in user space. Ret2dir [9] introduces a kernel exploitation technique that utilizes the virtual memory region inside the kernel space that directly maps part or all physical memory, which bypasses ret2usr protections. Boomerang [7] attacks work on Cortex-A TrustZone, in which a user-level non-secure application can leverage a secure state application to access a portion of memory that shall not be accessible to it.

To mitigate ret2usr attacks, kGuard [8] 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 [9] uses an exclusive ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory. To thwart boomerang attacks, a cooperative approach [7] 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 [12], [13] 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.

III. The ret2ns Attacks

In this section, we first discuss the threat model and ret2ns attack overview, which are followed by the detailed methodology and a walking example of the attacks.

A. System and Threat Model

Our work focuses on Cortex-M microcontrollers with TrustZone extension. 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

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

system (RTOS), e.g., FreeRTOS[2], 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 [14] and ACES [15] identify operations requiring privileged execution in bare-metal systems and modify the systems to only execute those operations in the privileged thread mode. We do not assume vulnerabilities in the kernel module of the non-secure state, but we assume a buggy secure state firmware or library, e.g., ARM TF-M [16], running in the secure state. Note that ret2ns vulnerabilities are likely to exist in any TEE implementations that allow direct control-flow transfers from secure state to non-secure state userspace programs but keep 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's userspace program 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 only needs to corrupt a 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 program. After all, the attacker's goal is not to execute arbitrary code in the secure state.

B. Attack Overview

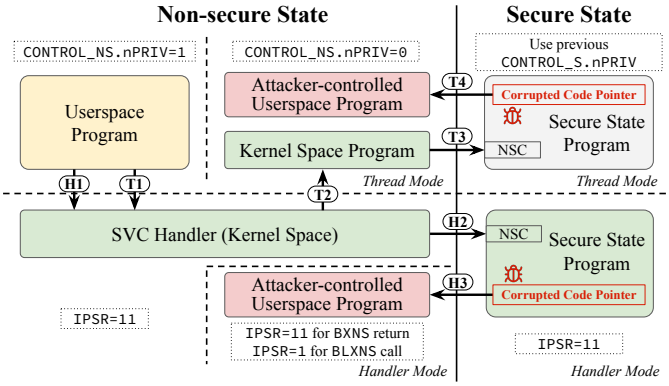

Figure 2: Overview of ret2ns attacks. A secure state code pointer used by `bxns` or `blxns` is corrupted and redirected to an attacker-controlled userspace program in the non-secure state. After the state switch, the attacker-controlled userspace program executes at the privileged level.

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 under the attacker's control 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 state 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. Another attack path exists when a secure state program makes an indirect call (`blxns`) with a corrupted code pointer (**H3**). In this case, `IPSR` has the value of 1.

In the thread-mode-originated attacks, the attacker-controlled 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 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. Walking Example: the Handler-mode-originated and `bxns` Case

Due to the page limit, we only 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.,

[2]https://www.freertos.org

```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

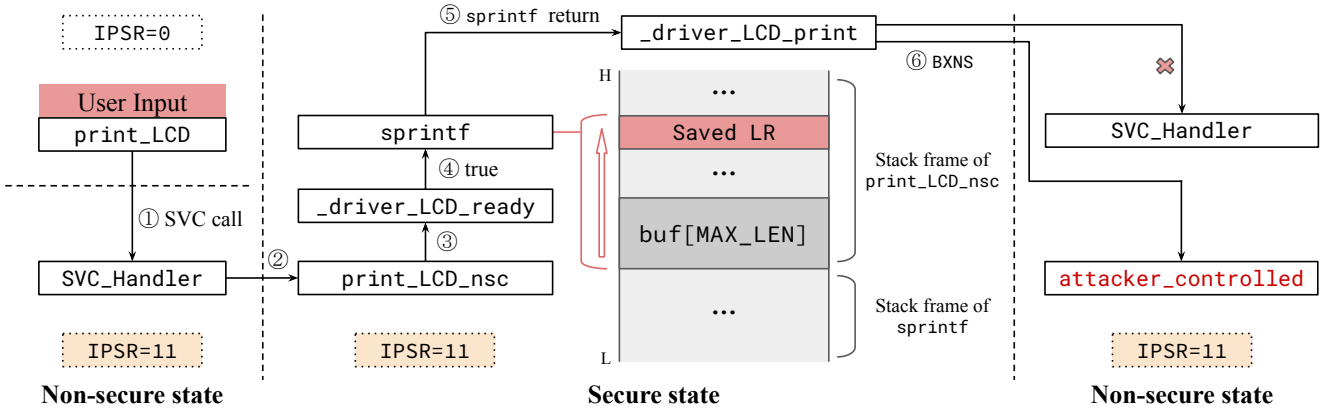

Figure 3: Attack walk-through of the example handler-mode-originated and `bxns` case.

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.

D. 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 [17]. On the software front, we changed the example TrustZone projects that came with Microchip Studio IDE [3] and Keil IDE [4] 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. Because Cortex-M 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.

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

IV. Defending Against ret2ns Attacks

The key to preventing ret2ns attacks is to disallow the execution of non-secure userspace programs at the privileged level. On the planned 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) Cortex-M23, M33, and M35P microcontrollers that hold a large market share do not have the PXN feature; (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) address masking, 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

```
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 2: MPU-assisted Address Sanitizer

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

Listing 2 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. We first check 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, we use the test target alternate domain instruction (`tta`) to check the MPU attribute of the destination address and save the result into `R0` (line 7). From the result, we 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, we load 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). We 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.

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

Listing 3: Address Masking. Instructions that are marked with *(not for blxns)* are not used in masking the `blxns` destination address.

B. Address Masking

The MPU-assisted address sanitizer is generic but not optimal in terms of efficiency. To address this issue, we also present a more efficient address masking mechanism. In this approach, we assume the userspace program and kernel space program are placed in different and known memory regions. 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 making 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` 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 3 presents an example address masking instrumentation for both `bxns` and `blxns` cases. Same as in the MPU-assisted address sanitizer, we first determine the non-secure privilege level by examining `IPSR` and `CONTROL_NS` (line 2-6). If the non-secure state is at the unprivileged level, the control flow is legitimate. Otherwise, we further verify 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, we perform a bit-wise masking operation to ensure the resulting address falls into the designated kernel space region (line 9). In the `bxns` case, after sanitizing the return address, we store it back to the stack for the original epilogue to use (line 10).

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 address masking approach to the vulnerable projects presented in Section III-D. The experiments confirmed that both defense mechanisms can defeat all four ret2ns attack variants.

TABLE I: Performance Evaluation Results in CPU cycles. Overhead shown in ().

T, N	Blinky	MPU-assisted Addr Sanitizer	Address Masking
10^7, 10^7	1,200,503,441	1,200,508,359 (0.0004%)	1,200,506,190 (0.0002%)
10^5, 10^5	12,503,869	12,508,385 (0.0361%)	12,506,793 (0.0234%)
10^7, 10^5	12,473,897	12,474,465 (0.0046%)	12,474,243 (0.0028%)
10^5, 10^7	1,203,433,289	1,203,892,073 (0.0381%)	1,203,674,733 (0.0201%)

2) Costs of the Worst-case Address Sanitizing Paths: The proposed instrumentation in Listing 2 and Listing 3 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) 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. 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 four 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.

Table I 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 N values. 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. 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 this paper, 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. We demonstrated the detailed methodology of ret2ns attacks with a walking example and confirmed the feasibility 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 (2237238 and 2037798) 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://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-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] N. Hardy, "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, 1988.

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.