On Runtime Software Security of TrustZone-M Based IoT Devices

Lan Luo*, Yue Zhang†§, Cliff Zou*, Xinhui Shao‡, Zhen Ling‡, Xinwen Fu†*
*Dept. of Computer Science, University of Central Florida, Orlando, FL, USA
Email: lukachan@knights.ucf.edu, czou@cs.ucf.edu
‡School of Computer Science and Engineering, Southeast University, Nanjing, China
Email: {xinhuishao, zhenling}@seu.edu.cn
†Dept. of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA
Email: xinwen_fu@uml.edu
§College of Information Science and Technology, Jinan University, Guangzhou, China
Email: zyueinfosec@gmail.com

GLOBECOM 2020 - 2020 IEEE Global Communications Conference | 978-1-7281-8298-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/GLOBECOM42002.2020.9322370

***Abstract*—Internet of Things (IoT) devices have been increasingly integrated into our daily life. However, such smart devices suffer a broad attack surface. Particularly, attacks targeting the device software at runtime are challenging to defend against if IoT devices use resource-constrained microcontrollers (MCUs). TrustZone-M, a TrustZone extension for MCUs, is an emerging security technique fortifying MCU based IoT devices. This paper presents the first security analysis of potential software security issues in TrustZone-M enabled MCUs. We explore the stack-based buffer overflow (BOF) attack for code injection, return-oriented programming (ROP) attack, heap-based BOF attack, format string attack, and attacks against Non-secure Callable (NSC) functions in the context of TrustZone-M. We validate these attacks using the Microchip SAM L11 MCU, which uses the ARM Cortex-M23 processor with the TrustZone-M technology. Strategies to mitigate these software attacks are also discussed.**

***Index Terms*—Internet of Things, microcontroller, TrustZone, software security**

I. Introduction

The Internet of Things (IoT) industry is booming, but has attracted cyber attacks [1], [2]. IoT involves a broad range of application domains such as home appliances, medical instruments, industry automation, and smart buildings. It is reported that more than 20 billion IoT devices have been distributed worldwide and this number will reach 41 billion by 2027 [3]. In this paper, we focus on IoT devices using low-cost and resource-constrained microcontrollers (MCUs), which can communicate with the outside world through venues such as WiFi, Bluetooth, NB-IoT and LoRa. The attack surface of such IoT devices includes data, networking, hardware, software and firmware/operating systems [4]–[7]. We are particularly interested in runtime software security of MCUs. Even if software integrity can be verified at boot time via mechanisms like secure boot, protecting software of embedded devices at runtime is challenging due to the heterogeneity and constrained computational resources of MCUs.

TrustZone-M, the TrustZone extension for ARMv8-M architecture, is an emerging solution to the runtime software security of IoT devices [8]–[10]. Specifically, it provides resource-constrained MCUs a lightweight hardware-based solution to a trusted execution environment (TEE) for security-related software, i.e., Secure World (SW), which is isolated from the rich execution environment, i.e., Non-secure World (NSW). The NSW software cannot access the SW resources directly. TrustZone-M provides a Non-secure Callable (NSC) memory region in the SW so that functions can be defined in the NSC region as the gateway from the NSW to the SW. To the best of our knowledge, TrustZone-M has not been adopted in commercial IoT products.

Corresponding author: Dr. Zhen Ling of Southeast University, China.

In this paper, we present the first security analysis of potential software security issues in TrustZone-M enabled IoT devices. We find that software vulnerabilities may exist in all regions of TrustZone-M, including the NSW, NSC and SWX (which is defined as the SW excluding the NSC). TrustZone-M is subject to stack-based code injection, code reuse attack, heap-based buffer overflow attack, format string attack, and NSC specific attacks. The first four attacks can occur in the NSW, NSC and SWX. By exploiting NSC vulnerabilities, an attacker is able to breach the security of the SW from the NSW.

A number of works have been done concerning the security issues in TrustZone. Cerdeira et al. [11] present systematization of knowledge (SoK) on the Cortex-A TrustZone security while our work focuses on the Cortex-M TrustZone. Iannillo et al. [12] propose a framework for the security analysis of TrustZone-M. However, their work does not identify concrete vulnerabilities/attacks against TrustZone-M. Jung et al. [10] design a secure platform based on the Platform Security Architecture (PSA) with a brief discussion of possible attacks. Our work demonstrates five types of realistic attacks, breaching the security of TrustZone-M.

This paper makes the following major contributions:

1) We are the first to perform a comprehensive security analysis of the runtime software security in TrustZone-M enabled IoT devices, and present potential software attacks against TrustZone-M. The SAM L11 MCU from Microchip uses the ARM Cortex-M23 processor with the TrustZone technology [13] and is used as an example in this paper to demonstrate the principles, while our

978-1-7281-8298-8/20/$31.00 ©2020 IEEE

methodologies can be extended to other similar products. We validate these attacks on SAM L11 and find that even the official code examples of SAM L11 contain security vulnerabilities.

2) We are the first to demonstrate how code injection attack, code reuse attack, heap-based buffer overflow attack, format string attack, and attacks against NSC functions compromise TrustZone-M, although some of these attacks are common on other platforms such as Linux, Windows and macOS.
3) To defeat these software attacks, we discuss the use of control flow integrity (CFI) and point out its limitations. We present guidelines, particularly with respect to fortifying the NSC functions, for the overall system security of TrustZone-M enabled IoT devices.

The rest of this paper is organized as follows. We introduce the background knowledge on ARM TrustZone-M and runtime security in IoT devices in Section II. We next present the five types of practical attacks against runtime software of TrustZone-M in Section III. The evaluation of the attacks is presented in Section IV. We discuss defense mechanisms in Section V and the paper is concluded in Section VI.

II. Background

In this section, we introduce the TrustZone-M technology and runtime software security issues in IoT applications.

A. TrustZone-M

TrustZone for ARM Cortex-A processors (TrustZone-A) is a security technology that isolates security-critical resources (e.g., secure memory and related peripherals) from the rich OS and applications. An ARM system on a chip with the TrustZone extension is split into two execution environments referred to as the **Secure World** (SW) and the **Non-secure World** (NSW). Software in the SW has a higher privilege and can access resources in both the SW and the NSW, while the Non-secure software is restricted to the Non-secure resources. The NSW may communicate with the SW using the monitor mode of TrustZone-A.

Recently, the TrustZone technology has been extended to the ARMv8-M architecture as TrustZone-M for some Cortex-M series processors, which are specifically optimized for resource-constrained MCUs. TrustZone-M has the SW and NSW, but differs from TrustZone-A in terms of implementation. One prominent difference is that TrustZone-M introduces a special memory region in the SW named Non-secure Callable (NSC) region to provide services from the SW to Non-secure software. Transition between the two worlds through the NSC region is achieved by NSC function calls and returns.

B. Runtime Software Security in IoT Devices

IoT devices are usually capable of connecting to remote servers or controllers and transferring messages to them via communication venues such as WiFi, Bluetooth, and low-power wide-area network. MCU based IoT devices are often programmed with languages such as C and C++ because they are compact, highly efficient and have the ability of direct memory control [14]. Such languages provide programmers a flexible platform to interact with the low-level hardware. On the flip side, they are notoriously error-prone and daunted by security issues. Attackers may perform runtime software attacks against vulnerable IoT devices with such features.

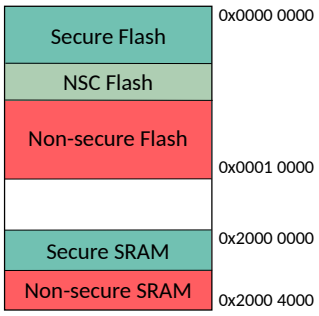

Fig. 1. Memory layout of SAM L11. The memory is divided into the SW and NSW at the hardware level. Code in the SW (Secure Flash, NSC Flash, and Secure SRAM) can access the whole chip, while code in the NSW (Non-secure Flash and Non-secure SRAM) can only directly access resources inside the NSW.

Runtime software attacks can hijack the program control flow by altering the control data (e.g., return address and function pointer) or change program memory by manipulating non-control data [4]. Often in such an attack, an adversary corrupts the vulnerable memory by inputting a carefully crafted malicious payload, which eventually results in abnormal program behaviors.

III. Attacks against Runtime Software in TrustZone-M

In this section, we first introduce the threat model on how a TrustZone-M enabled IoT device may be attacked. We then present five runtime software attacks against TrustZone-M enabled IoT devices. We use the SAM L11 MCU as the example while the principle is the same for all TrustZone-M enabled devices.

A. Threat Model

We consider a victim IoT device using the TrustZone-M enabled MCU. It is assumed that security-related coding mistakes exist in the software of the victim device which is able to receive inputs from the Internet or peripherals. Though the SW of TrustZone-M provides a TEE that the NSW software cannot directly access, the TEE can only work normally under the assumption that Secure software is well crafted with no security-related coding mistakes. However, coding mistakes may exist in TrustZone-M's NSW, the NSC region, and the SWX region. Memory layout of a TrustZone-M based MCU, SAM L11, is shown in Figure 1. An adversary can exploit the coding mistakes and send a malicious input (i.e. payload) to deploy software attacks. Even if the SW does not accept inputs from the Internet or peripherals and only the NSW communicates with the outside world, an attacker

TABLE I
SOFTWARE ATTACKS IN TRUSTZONE-M

Software Attacks	NSW	NSC	SWX
Code injection	✓	✓	✓
ROP	✓	✓	✓
Heap-based BOF	✓	✓	✓
Format string attack	✓	✓	✓
NSC-specific exploit	N/A [a]	✓	N/A [a]

[a] The *NSC-specific exploit* targets the NSC memory and is not applicable (N/A) for the NSW and the SWX.

may compromise the NSW and feed malicious inputs into vulnerable NSC functions, which can access Secure resources. Therefore, if a NSC function is vulnerable, the entire SW may be compromised.

B. Runtime Software Attacks

Table I lists software attacks we have identified against the NSW, NSC and SWX of TrustZone-M. It can be observed that traditional software attacks found in other platforms such as computers and smart phones can be conducted in all regions of TrustZone-M, including code injection, return-oriented programming (ROP), heap-based buffer overflow (BOF), and format string attacks, if requisite software flaws present. We also discover potential exploits specifically targeting the NSC. Here, all attacks against the NSC refer to those deployed from the NSW. We present the details and challenges of these attacks in the context of TrustZone-M below.

*1) **Stack-based Buffer Overflow Attack for Code Injection**:* The stack-based BOF is a canonical memory corruption attack that occurs on the stack when a larger input is written to a local buffer without checking the buffer's boundary. Listing 1 presents an example, in which *buf[256]* will overflow if the input array is longer than 256 bytes. As a result, the extra data will overwrite the adjoining stack contents including the return address, at which the control flow will continue after the subroutine return. Adversaries may perform stack-based BOF attack for malicious code injection. The control flow can be redirected to the malicious code sent along with the payload by overflowing the local buffer and overwriting the original return address with the entry address of the malicious code.

```
1 void BOF_func(char *input){
2     char buf[256];
3     strcpy(buf, input);}
```

Listing 1. Example of a function with BOF vulnerability

To specifically implement a stack-based BOF attack against the ARMv8-M architecture, we first investigate its stack structure. A stack frame for a function in ARMv8-M consists of local variables, variable registers (R4–R7), and return address, as illustrated in Figure 2. By exploiting functions with BOF vulnerabilities, an adversary is able to copy a crafted payload to the buffer, overwrite the return address, and inject malicious code onto the stack. While constructing the malicious payload, the adversary needs to know the entry of the malicious code on the stack. A common solution is to utilize the *JMP SP* instruction presenting in the device's firmware [15]. Even if

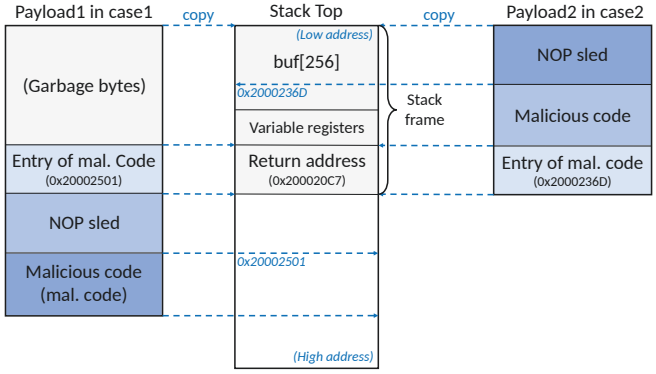

Fig. 2. Stack-based buffer overflow attack for code injection

there is no such instruction in the firmware, an adversary may enumerate possible entry addresses of malicious code to find the correct one. A wrong address in the payload leads to program crash and restart (if automatic restart is enabled), and the malicious code would not be executed until the correct entry address is hit. This entry scanning process can be more efficient by inserting a sequence of *NOP* (no-operation) instructions, called a NOP sled, before the injected malicious code in the payload, since any hit of a NOP instruction will lead to the execution of malicious code eventually.

A challenge of implementing BOF with respect to ARMv8-M comes from the null bytes (0x00) in the payload, which also function as the C string terminator. If the exploitable function treats the payload as a string (e.g., *strcpy()* and *strcat()*) and some null bytes exist in the crafted payload, the function will cease to copy the payload right after hitting a null byte and the attack will fail. We discuss two scenarios of null bytes below.

First, null bytes can exist in the malicious code and NOP sled since null bytes are naturally contained in many ARM instructions. To eliminate these null bytes, one can replace the problematic instructions by alternative instructions with the same functionalities but without null bytes. For an instance, a NOP instruction (0xBF00) can be replaced by the instruction *MOV R2, R2* (0x121C).

The second scenario refers to the null bytes in the entry address of the malicious code. In SAM L11, the malicious code has to be injected onto the stack, which is on the SRAM with a fixed range of address from 0x20000000 to 0x20004000, within which the higher halfword of any addresses is 0x2000, containing a null byte all the time. Taking Payload1 in Figure 2 as an instance, since the NOP sled and malicious code are positioned after the entry address, the copy process of Payload1 will terminate when the null byte in the entry address is hit. Copying either the NOP sled or the malicious code to the stack would fail in this case. A potential solution is to construct the payload like Payload 2 in Figure 2, where the entry of malicious code is placed at the bottom. Because the little-endian ordering in ARMv8-M, the 0x2000 is located at the last two bytes of Payload 2 and shall be the only two bytes missing when copied to the stack. The original return address already contains 0x2000 in its upper halfword if the

caller function is executed from the SRAM, in which case the BOF will still be applicable.

Payload2 shows an example that the malicious code is copied to address 0x2000236D. In this case, the NOP sled and malicious code are copied firstly. The copy operation will not stop until it reaches the null byte in the entry address if both NOP sled and malicious code do not contain any null bytes. For the return address on the stack, the lower halfword will be overwritten by the last two bytes (0x236D) of the entry address in the payload and keep its higher halfword unchanged. So the updated return address would be 0x2000236D, which is the entry of the malicious code.

*2) **Return-oriented Programming Attack**:* BOF based code injection can be mitigated by security mechanisms like non-executable memory [16], which prevents code execution from certain memory region. However, an attacker can bypass such defense by leveraging code reuse attack (CRA). A representative CRA is ROP attack. Utilizing BOF to overwrite the return address, ROP redirects the control flow to a target code sequence (called a gadget) found in the existing software code. It is also possible to chain several gadgets for more complex program control. Each gadget in the chain is a code segment responsible for certain operations (e.g., arithmetic operations and load/store data) and must end with the epilogue of a subroutine for the sake of chaining the gadgets. In ARMv8-M, the instruction sequence {*POP LR, BX LR*}, which is the epilogue of leaf subroutines, pops a word to link register (*LR*), and then branches to the address specified by *LR*. Instruction *POP PC*, which is the epilogue of non-leaf subroutines, directly pops a word to program counter (*PC*).

Now we explain how to chain the gadgets utilizing the subroutine epilogue in each of them. An adversary needs to craft a "gadget stack" and sends it along with the payload. Each gadget in the chain except the last one, has a corresponding gadget frame placed on the gadget stack. A gadget frame consists of several words of data that will be popped to the operand registers of the last *POP* instruction in that gadget. Data provided by the gadget frame include the address of the next gadget, which helps to jump to the next gadget after being popped. An example of a chain of three gadgets in ARMv8-M is presented in Figure 3. The payload contains the entry of Gadget 1 and two gadget frames corresponding to Gadget 1 and 2. To ensure the entry of Gadget 2 will be popped to *LR*, the gadget frame for Gadget 1 contains two more words before the entry word since the second to last instruction in Gadget 1 pops the third word from the stack frame to *LR*. Two words before the entry of Gadget 2 are provided such that they will be popped to *R4* and *R5* instead. Similarly, the gadget frame for Gadget 2 provides two words of data which will be popped to *R4* and *PC* so that execution of Gadget 3 can be routed to start.

*3) **Heap-based BOF Attack**:* Heap-based BOF refers to a form of BOF exploitation in the heap area. As a SAM L11 project is linked with the GNU libc, the heap in SAM L11 is managed by the glibc allocator [17]. The glibc allocator manages free chunks in a doubly-linked list where each chunk

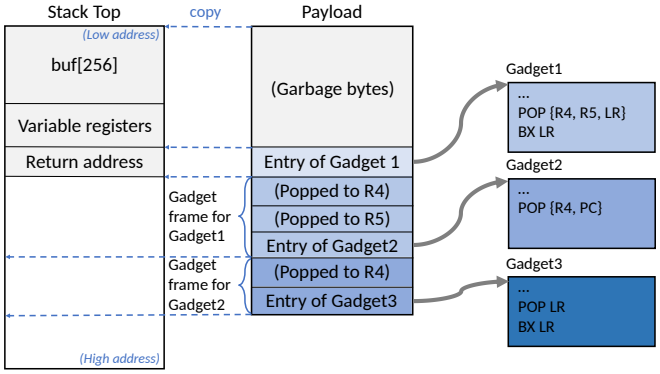

Fig. 3. Return-oriented programming attack

contains the metadata of a forward pointer and a backward pointer pointing to the free chunks before and after it. A simple exploitation of heap-based BOF is to overwrite the function pointers stored on the heap to hijack the program control flow. An adversary may also overwrite the metadata of a free chunk via overflowing an adjacent activated data chunk. By manipulating the pointers in the metadata, an adversary is able to corrupt arbitrary memory with arbitrary values [18].

*4) **Format String Attack**:* A format function such as *printf()* usually requires several arguments. The first argument is a format string which may contain some format specifiers (e.g., *%s*, *%x*). When the format function is executed, those format specifiers will be replaced by the subsequent arguments with the specified formats. Therefore, the number of specifiers in the format string is supposed to match the number of additional arguments. The format string exploits occur when a format function receives a format string input that contains more format specifiers than additional arguments supplied. By sending a well-crafted format string with specific format specifiers to a vulnerable format function, an adversary may eventually cause program crash, memory leakage, and memory alteration at a specific memory location of the stack, or even in an arbitrary readable/writable memory location specified by an address.

In SAM L11, an adversary is able to exploit format string vulnerabilities for memory crash and reading/writing some values at a specific stack location by sending a malicious string input containing more format specifiers than expected. For example, by sending the string *"%x %x %x"* to the vulnerable function illustrated in Listing 2, in which no arguments are provided to the three specifiers in the input format string, three bytes of data following the return address on the stack will be printed in hexadecimal. However, reading or writing at an arbitrary memory location specified by an address is unachievable in SAM L11 due to the particular memory addressing as shown in Figure 1. Such attacks require the target address to present in the input format string, e.g., *"\x34\x12\x00\x20 %x %x %x %x %s"*. Adversaries who aim at the memory of SAM L11 will find that any address of the memory would contain at least one null byte. During the compilation, the process of parsing the input format string will

terminate when the null byte in the target address is reached. The rest of the input format string cannot be parsed correctly, hence, the attack would fail.

```
void fmt_str(char *input){
    printf(input);
    ...
```

Listing 2. Example of a vulnerable format string function

*5) **Attacks against NSC Functions**:* The Non-secure software in the NSW may desire to use the Secure services in the SW. For the sake of such requirement, TrustZone-M provides the NSC memory region within the SW. Developers are able to define NSC functions in the NSC as the gateway to the SW. NSC functions are characterized with two features: (i) They can be called from the NSW; (ii) They have the privilege of accessing Secure resources since the NSC is a region within the SW. With such abilities, Non-secure software can call specific Secure services by first calling the corresponding NSC functions. The NSC functions then help to call the target Secure functions and pass the required arguments assigned by the Non-secure callers.

As the gateway to the SW, the implementation of the NSC software should be particularly cautious. According to the guidance from ARM [19], hardware, toolchain, and software developers share a common responsibility to implement the NSC software securely. Though some requirements are offered in the guidelines, since the hardware and toolchain vary from vendors to vendors, there is no off-the-shelf solution to implementing trusted NSC software.

We identify two pitfalls that software developers may meet while programming the NSC functions. The first pitfall is caused by the data arguments sent from the NSW. The toolchain of SAM L11 only helps to generate the Secure gateway veneer for NSC functions but leaves the function programming to the developers. Security-related coding mistakes may present in the NSC functions as well and can be exploited by crafting Non-secure data inputs. Software exploits in the NSC region would lead to a compromised SW. This is because the NSC region belongs to the SW and a compromised NSC program under the control of an adversary can access any resources inside the SW.

```
int NSC_func(int *a, int b, int *c){
    int *addr = a; int num = b; int *sum = c;
    for (int i = 0; i < num; i++){
        *sum += addr[i];}
    return *sum;}
```

Listing 3. Example of a vulnerable NSC function

The second pitfall comes from the untrusted pointer inputs. When Non-secure software passes pointer arguments to the SW through NSC functions, NSC functions should ensure that these pointers point to the Non-secure memory. Otherwise, NSC and Secure functions may assist the Non-secure software to read or write the Secure memory. The vulnerable NSC function illustrated in Listing 3 can leak and corrupt the Secure memory contents at arbitrary Secure addresses if the first and the third arguments are Secure addresses and the second argument is set to 1. The violation of the principle that "Secure resources are not allowed to be accessed by the NSW" severely harms the fundamental security of the TrustZone-M implementation.

IV. Evaluation

In this section, we evaluate the five software attacks described in Section III. We are able to successfully perform these attacks against a TrustZone-M enabled MCU, SAM L11.

A. Experiment Setup

We use a laptop as the attacker to continually send inputs to a SAM L11 Xplained Pro Evaluation Kit as the victim device. The laptop is connected with SAM L11 through a USB-to-UART adapter while an attacker may also inject malicious strings into an Internet connection of a SAM L11 based IoT device. In SAM L11, two UARTs are configured accordingly as a Non-secure peripheral and a Secure peripheral to receive inputs sent from the laptop to the NSW and SW respectively. For the first four attacks, we construct specific vulnerable functions in both Non-secure and Secure applications of SAM L11 and malicious payloads will be sent through the UARTs to trigger the attacks. The sizes of the payloads and experiment results are given in Table II.

B. Experiment Results

In the BOF-based code injection attack, we configure the stack to be executable, which is commonly configurable in TrustZone-M enabled MCUs. We assemble the payload with a constant string, malicious code, the entry of the malicious code (obtained via random brute-force scanning), and a NOP sled with 50 NOP instructions. The malicious code is designed to call a print function and supply the address of a constant string as the argument of the print function. Our attack succeeds and the constant string is printed in the adversary's terminal.

As a proof-of-concept implementation of ROP, we craft a chain of gadgets with three exploitable gadgets by splitting the assembly code of a program, which prints the memory content at a given address, into three code segments. A subroutine epilogue (i.e., *POP PC*) is appended at the end of each code segment. These gadgets are pre-stored at different locations of the flash in advance. We craft a gadget stack to chain these gadgets and send it along with the payload to SAM L11. As a result, the intended constant string is successfully printed on the adversary's terminal. A way to evaluate the feasibility of ROP against a certain program is to count up the occurrences of potential gadgets in the program. In fact, this process is equivalent to counting up the number of "*POP PC*" and "*BX LR*" instructions according to the definition of potential gadgets introduced in Section III. We take a basic Non-secure application image, which only initializes necessary peripherals, as an example and search all the subroutine epilogues in it. The Capstone disassembly engine [20] is used to dissemble and search in the binary code. The size of the example image is 4.14KB with 1908 instructions in total. As a result, 49 "*POP PC*" and 16 "*BX LR*" are found in the image binary, representing 3.41% of the whole image.

TABLE II
SIZES OF PAYLOADS AND EXPERIMENT RESULTS IN DIFFERENT ATTACK SCENARIOS.

Attack Scenarios	Sizes of Payloads (byte)	Experiment Results
Code injection	256	90.62s is spent on scanning the entry of the malicious code, which is then successfully executed.
ROP	96	Crafted string is printed; 65 potential gadgets are found in a 4.14KB image.
Heap-based BOF	24	The malicious code is successfully executed.
Format string exploits	24	Five sequential bytes are read from the Non-secure and Secure stacks.
NSC-specific attacks	24 & 4	3 of 5 demo projects contain vulnerable NSC functions; Five sequential bytes are read from the Secure stack; Secure memory content is printed.

To launch the heap-based BOF attack, we first construct two adjacent data blocks on the heap of SAM L11 and a vulnerable *memcpy()* function, which copies the input payload to a buffer in the first data block without checking its boundary. Our payload successfully triggers the BOF attack and overwrites a function pointer in the next data block with the entry of a pre-injected malicious code. The malicious code is later executed when that function is called.

As we stated in Section III-B4, an adversary can exploit the vulnerable format string function in SAM L11 to read out the stack contents. The payload used is "%08x %08x %08x %08x %08x" and we eventually read five sequential bytes from the stack via UARTs.

To verify the feasibility of NSC-specific attacks, we look into the example software projects provided by the vendor of SAM L11, five of which contain NSC software implementations. We statically analyze the source code of these five NSC implementations and find three to be vulnerable. These three implementations share two vulnerable NSC functions as in Listing 4, where two of them contain the first function and the other contains the second function. The first vulnerable function is subject to the format string attack when it is called by the Non-secure software and the argument is a crafted format string input that can be controlled by an adversary. In our experiment, we send "%08x %08x %08x %08x %08x" as the payload and five sequential bytes from the Secure stack are eventually printed in the adversary's terminal. The second function has an information leakage problem. We call this function in the NSW with an argument which is a Secure address, as a result, the Secure memory content at the target location is then printed.

```
1 void __attribute__((cmse_nonsecure_entry))
      nsc_secure_console_puts (char *string){
2   non_secure_puts(string);}
3
4 void __attribute__((cmse_nonsecure_entry)) nsc_puts(
      uint8_t * string){
5   printf("%s", string);}
```

Listing 4. Vulnerable NSC functions in SAM L11 demo code

V. DISCUSSION: DEFENSE TO RUNTIME SOFTWARE ATTACKS AGAINST TRUSTZONE-M

With the increasing concerns of IoT security, more and more manufacturers start to make their MCU products support various security features such as secure boot and secure storage [21]. Runtime software attacks targeting IoT devices can be mitigated in a way with few costs by properly enabling some of the security features provided by MCUs. Taking SAM L11 as an example, the code injection attack can be neutralized via setting the RXN (RAM eXecute never) and DXN (Data eXecute never) fuse bits, via which code execution from data memory would not be allowed. In addition, SAM L11 supports secure debugging port with key authentication and allows to disable the write operation on code memory. These two security mechanisms ensure the code integrity at runtime so that adversaries cannot easily hijack the control flows by rewriting the firmware. However, other aforementioned runtime software attacks (i.e., ROP, heap-based BOF, format string exploits, and NSC-specific attacks) are difficult to resist with the equipped security features.

The control flow integrity (CFI) is a technique for preventing runtime control-oriented attacks such as ROP. By monitoring the control flow of a program at runtime, it can detect unexpected control flow changes. [7] provides an implementation of CFI for TrustZone-M to protect the NSW. In it, a control flow graph (CFG) of the Non-secure program is constructed by static or dynamic analysis of its code and is saved in a non-writable region of the NSW. Code instrumentation is performed so that the program jumps to a *branch monitor* before any control flow changes in the original code. The branch monitor refers to the CFG and monitors control flow changes at runtime. Before a function call, the correct return address is pushed on a *shadow stack* in the SW. Since a function might be called by different callers and return to different places at runtime, the CFG cannot tell the exact return address at runtime. The shadow stack in the SW is used to record the correct return address for a certain function call. Here the stored return addresses must be fully protected from being altered. The SW, which can be seen as a trust anchor for the NSW, provides the required secure storage, namely shadow stack, for the correct return addresses and a trusted execution environment for any operations on the shadow stack.

The CFI for protecting the control flow of the NSW is not sufficient for the overall system security. It can be observed from Table I that all software attacks may occur in both the SW (including NSC and SWX) and NSW. Recall that the CFI mechanism requires a trusted execution environment and a secure storage. In the case of TrustZone-M, the SW is supposed to play the role of such a trust anchor. If the SW itself is insecure and vulnerable to potential software attacks at runtime, it cannot provides the indispensable secure storage and trusted execution environment required by CFI for the NSW. Thus the effectiveness of the CFI enforcement for the

NSW would be harmed. Another issue of CFI is that it may not defeat the heap-based BOF or format string attacks if control flows are unchanged but sensitive data are modified.

For the overall security of TrustZone-M based IoT devices, the following strategies may be adopted. First, the SW shall be very cautious about authenticity and security of the Internet connection in order to avoid remote exploits of SW software vulnerabilities. Second, the arguments sent from the NSW to an NSC function may be an address or application data. If it is an address, the NSC function must verify that it is not a Secure address before passing the address argument to the SW, since an NSW program shall not access the SW resources directly. If it is application data, input validation and sanitization shall be carefully performed. Third, security mechanisms including CFI and onboard executable space protection of TrustZone-M shall be applied to the NSW for control flow integrity. Finally, secure coding, code review and penetration testing are critical to the overall system security and the best practice shall be adopted [22]. Runtime software attacks may also be mitigated by programming the MCUs with security oriented languages such as Java [23], [24] and Rust [25].

VI. Conclusion

This paper gives the first systematic runtime software security analysis for TrustZone-M enabled IoT devices. We present possible pitfalls of TrustZone-M programming and present five potential software attacks against TrustZone-M, including the stack-based BOF attack for code injection, return-oriented programming, heap-based BOF attacks, format string attacks and attacks against NSC functions. We validate these attacks on a TrustZone-M enabled MCU, SAM L11. To defend these attacks, we present guidelines for an overall system security of TrustZone-M enabled IoT devices.

Acknowledgements

This research was supported in part by National Key R&D Program of China 2018YFB2100300, 2018YFB0803400, and 2017YFB1003000, US National Science Foundation (NSF) Awards 1931871 and 1915780, US Department of Energy (DOE) Award DE-EE0009152, National Natural Science Foundation of China (Grant Nos. U1736203, 61877029, 61972088, 61532013), Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] SonicWall, "2020 sonicwall cyber threat report: Threat actors pivot toward more targeted attacks, evasive exploits," Feb. 2020. [Online]. Available: https://www.sonicwall.com/news/2020-sonicwall-cyber-threat-report/

[2] "Owasp internet of things project," 2018. [Online]. Available: https://owasp.org/www-project-internet-of-things/

[3] K. Gyarmathy, "Comprehensive guide to iot statistics you need to know in 2020," Mar. 2020. [Online]. Available: https://www.vxchnge.com/blog/iot-statistics

[4] A. Mohanty, I. Obaidat, F. Yilmaz, and M. Sridhar, "Control-hijacking vulnerabilities in iot firmware: A brief survey," in *The 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 2018.

[5] H. HaddadPajouh, A. Dehghantanha, R. M. Parizi, M. Aledhari, and H. Karimipour, "A survey on internet of things security: Requirements, challenges, and solutions," *Internet of Things*, p. 100129, Nov. 2019.

[6] K. V. English, I. Obaidat, and M. Sridhar, "Exploiting memory corruption vulnerabilities in connman for iot devices," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 247–255.

[7] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2017, pp. 259–284.

[8] "Trustzone for cortex-m," Arm. [Online]. Available: https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m

[9] L. Liu, J. Ma, C. Zhang, T. Chong, H. Zhang, and Y. Dong, "Security software system design and implementation for microcontrollers based on trustzone," *DEStech Transactions on Computer Science and Engineering*, no. cisnrc, 2019.

[10] J. Jung, J. Cho, and B. Lee, "A secure platform for iot devices based on arm platform security architecture," in *2020 14th International Conference on Ubiquitous Information Management and Communication (IMCOM)*. IEEE, 2020, pp. 1–4.

[11] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, 2020, pp. 18–20.

[12] A. K. Iannillo and R. State, "A proposal for security assessment of trustzone-m based software," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 126–127.

[13] "Saml11 xplained pro evaluation kit," Microchip. [Online]. Available: https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/DM320205

[14] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2019, pp. 1275–1295.

[15] "Return oriented programming (arm32)," Azeria Labs. [Online]. Available: https://azeria-labs.com/return-oriented-programming-arm32/

[16] "Nx bits - microsoft wiki - fandom," Microsoft. [Online]. Available: https://microsoft.fandom.com/wiki/NXbit

[17] "Arm heap exploitation," Azeria Labs. [Online]. Available: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

[18] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *22nd International Symposium on Reliable Distributed Systems*, Oct 2003, pp. 260–269.

[19] "Armv8-m secure software guidelines," Arm. [Online]. Available: https://developer.arm.com/docs/100720/0200/secure-software-guidelines

[20] "The ultimate disassembly framework – capstone – the ultimate disassembler." [Online]. Available: http://www.capstone-engine.org/

[21] B. Pearson, L. Luo, Y. Zhang, R. Dey, Z. Ling, M. Bassiouni, and X. Fu, "On misconception of hardware and cost in iot security and privacy," in *ICC 2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[22] Berkeley Information Security Office, "Secure coding practice guidelines," 2020. [Online]. Available: https://security.berkeley.edu/secure-coding-practice-guidelines

[23] S. A. Ito, L. Carro, and R. P. Jacobi, "Making java work for microcontroller applications," *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 100–110, 2001.

[24] "Stm32 ide," STMicroelectronics. [Online]. Available: https://www.st.com/en/development-tools/stm32-ides.html

[25] "Rust: Embedded devices," 2020. [Online]. Available: https://www.rust-lang.org/what/embedded