On Security of TrustZone-M-Based IoT Systems

Lan Luo, Yue Zhang, Clayton White, Brandon Keating, Bryan Pearson, Xinhui Shao, Zhen Ling, *Member, IEEE*, Haofei Yu, Cliff Zou, *Senior Member, IEEE*, and Xinwen Fu, *Senior Member, IEEE*

***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 designed specifically for MCUs, is an emerging hardware security technique fortifying software security of MCU-based IoT devices. This article introduces a comprehensive security framework for IoT devices using TrustZone-M-enabled MCUs, in which device security is protected in five dimensions, i.e., hardware, boot-time software, runtime software, network, and over-the-air (OTA) update. Along developing the framework, we also present the first security analysis of potential runtime software security issues in TrustZone-M-enabled MCUs. In particular, we explore the feasibility of launching stack-based buffer overflow (BOF) attack for code injection, return-oriented programming (ROP)**

Manuscript received April 12, 2021; revised October 15, 2021 and December 11, 2021; accepted January 3, 2022. Date of publication January 19, 2022; date of current version June 7, 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0803400 and Grant 2018YFB2100300; in part by the U.S. National Science Foundation (NSF) under Award 1931871, Award 1915780, and Award 1643835; in part by the U.S. Department of Energy (DOE) under Award DE-EE0009152; in part by the National Natural Science Foundation of China under Grant 62022024, Grant 61972088, Grant 62072103, and Grant 62072098; in part by the Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060; in part by the Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201; in part by the Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant 93K-9; and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. *(Corresponding author: Zhen Ling.)*

Lan Luo, Bryan Pearson, and Cliff Zou are with the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA (e-mail: lukachan@knights.ucf.edu; bpearson@knights.ucf.edu; czou@cs.ucf.edu).

Yue Zhang was with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA 01854 USA. He is now with the College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: zyueinfosec@gmail.com).

Clayton White was with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA. He is now with Google, Chicago, IL 60607 USA (e-mail: claytonwhite@knights.ucf.edu).

Brandon Keating was with the Department of Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA 01854 USA. He is now with Globus Medical, Audubon, PA 19403 USA (e-mail: brandon_keating@student.uml.edu).

Xinhui Shao and Zhen Ling are with the School of Computer Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: xinhuishao@seu.edu.cn; zhenling@seu.edu.cn).

Haofei Yu is with the Department of Civil, Environmental and Construction Engineering, University of Central Florida, Orlando, FL 32816 USA (e-mail: haofei.yu@ucf.edu).

IoT devices, and we present potential software attacks against TrustZone-M. The SAM L11 MCU from Microchip uses the ARM Cortex-M23 processor with the TrustZone technology [12] and is employed as an example in this article to demonstrate the principles, while our methodologies can be extended to other similar products. We validate these attacks on SAM L11 and find that even the official coding demos of SAM L11 contain security vulnerabilities. We demonstrate how the code injection attack, code reuse attack (CRA), heap-based buffer overflow (BOF) attack, format string attack, and attacks against NSC functions may compromise TrustZone-M, while some of these attacks are common on other platforms, such as Linux, Windows, and MacOS.

We are the first to design and implement an image-based address space layout randomization (ASLR) scheme for IoT devices, denoted as image-based ASLR (iASLR). iASLR is unique since it relocates an image every time the device boots while the image layout is randomized only one time in the related work [13]. We design the static code patching and control flow correction schemes to tackle the addressing issues caused by image relocation.

We implement a secure and trustworthy air quality monitoring device, called STAIR, with a TrustZone-M-enabled MCU to demonstrate the proposed security framework. In particular, we demonstrate the use of nonexecutable RAM and data flash, secure NSC functions, and control flow integrity (CFI) for the overall system security of TrustZone-M-enabled IoT devices.

We evaluate the attacks using real-world examples and show even the example software projects provided by Microchip have vulnerabilities. We also present the performance of STAIR such as the cryptographic operation overhead.

A conference version published previously [14] mainly focuses on analyzing the runtime software security issues and potential attacks of TrustZone-M-based MCUs. Compared to the conference version, we discuss the overall security of TrustZone-M-based IoT devices in this article. To address the security issues, we propose a comprehensive security framework and implement an air quality monitoring device for demonstration.

The remainder of this article is organized as follows. We introduce the background knowledge on ARM TrustZone-M technique, TrustZone-M enabled MCUs, and runtime security issues of IoT devices in Section II. In Section III, a security framework for TrustZone-M-based IoT devices with five dimensions is presented. We then illustrate five types of practical attacks against runtime software of TrustZone-M in Section IV. An implementation of STAIR device using the security framework is described in Section V and we introduce our iASLR scheme—iASLR—in Section VI. The evaluation of the runtime software attacks and implemented STAIR device is presented in Section VII. We present related work in Section VIII and conclude this article in Section IX.

II. Background

A. TrustZone for Armv8-M

TrustZone for ARM Cortex-A processors (TrustZone-A) is a hardware-based security technology that isolates security-critical resources (e.g., secure memory and related peripherals) from rich OS and applications. An ARM system on a chip with the TrustZone extension is split into two execution environments referred to as the SW and the NSW. Software in the SW has a higher privilege and can access resources in both the SW and the NSW, while the nonsecure software is restricted to the nonsecure resources. Switching between the two worlds is implemented with the secure monitor mode of the processor.

Recently, the TrustZone technology has been extended to the ARMv8-M architecture as TrustZone-M for some ARM Cortex-M 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 NSC region to provide services from the SW to the NSW. Transitions between the two worlds through the NSC region are achieved by NSC function calls and returns.

To distinguish from general secure and nonsecure objects, in the rest of this article, we use terms *secure* and *nonsecure* to specifically describe resources in the SW and NSW.

B. SAM L11

In July 2018, Microchip announced the first TrustZone enabled MCU with the name of SAM L11. Equipped with Cortex-M23 core and TrustZone-M security extension, this chip is described as the lowest power 32-bit MCU in the industry that ensures robust hardware-based security at the same time.

Security Features: Besides TrustZone-M, SAM L11 offers multiple optional security features, including secure boot, hardware cryptoaccelerator, true random number generator, secure pin multiplexing, secure data flash, and TrustRAM.

Non Volatile Memory (NVM) Rows: NVM rows are secure memory regions containing critical system configuration fuses, which are used by the system at boot time. Security-related NVM rows include boot configuration row (BOCOR) for boot security configurations and user row (UROW) for other security related configurations. The NVM rows can only be updated by secure access and will not take effect until a reboot.

Memory Layout: Taking ATSAML11E16A, the top model of SAM L11, as an example, it has a 64-kb code flash for software images, a 16-kb SRAM for volatile data, and a 2-kb data flash for nonvolatile user data. Fig. 1 illustrates the memory mapping of SAM L11. Due to the existence of TrustZone-M, memory in SAM L11 can be divided into the SW and NSW at hardware level. While the starts and ends of code flash, SRAM, and data flash are fixed addresses, starts of NSC flash, nonsecure code flash/SRAM/data flash are modifiable and can be defined in UROW.

C. Runtime Software Security in IoT Devices

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 [15]. Such languages provide programmers a flexible platform to interact with the low-level hardware directly.

TABLE I
SECURITY FRAMEWORK FOR TRUSTZONE-M-ENABLED IOT DEVICES

Security Demensions	Vulnerabilities	Defenses
Hardware	Memory manipulation via u[illegible]	

software. Using a security key to secure the communication through the interfaces is a common way to filter out unauthorized access. In practice, different groups may be granted different privileges of accessing memory contents. For instance, an original equipment manufacturer (OEM) is able to access both the SW and NSW through the hardware interfaces, while a third party may only be allowed to access the Nonsecure applications for security concerns. Therefore, it is necessary to use at least two keys to distinguish the different access privileges. That is, users with higher privilege can access both the SW and NSW, while users with lower privilege can only access the NSW.

In addition to programming interfaces, hardware ports, such as UART, I2C, and SPI, that receive data from other peripherals might be attacked if data are maliciously manipulated by adversaries and specific vulnerabilities exist in the software. Since such attacks highly depend on bugs in the software and how the software can be exploited, we will discuss them later in Sections III-C and IV.

B. Boot-Time Software Security

Software should be validated before being loaded and executed at device's boot time so that any alteration of the software can be detected. Usually secure boot works as the root of trust for IoT devices, making sure that the software is from the OEM and starts the execution in the normal state. The work flow of secure boot begins with a trusted piece of code (which is usually write protected, e.g., Boot ROM and efuse) as the root of trust, which will validate other programs to be executed. Devices enabled by TrustZone-M require such trusted code to verify the integrity and authenticity of all nonvolatile memory in both the SW and NSW.

C. Runtime Software Security

Runtime software security is a critical issue for IoT devices, for which C or C++ is a preferable programming language. Coarsely programmed C or C++ software may contain memory corruption errors and is naturally fragile to software attacks, such as program crash, data leakage, control flow hijack, and firmware altering. Though TrustZone-M is designed to protect runtime execution inside the SW, software attacks may occur in the NSW, or even in the SW if the secure applications are not programmed in a correct way. In Section IV, we demonstrate in detail how such attacks could occur in TrustZone-M-enabled devices, and how they could compromise system security.

D. Network Security

In the context of IoT, devices are connected to the cloud or other devices via the Internet. Data transmitted through the network must be carefully protected in case of cyber attacks, such as man-in-the-middle attack, eavesdropping attack, replay attack, etc. To overcome the network security issues, secure communication protocols, such as hypertext transfer protocol secure (HTTPS) and message queuing telemetry transport over TLS (MQTTS), should be used so that servers and clients are authenticated before the connection is established, the integrity of messages is checked upon being received, and network traffic is encrypted during transmission.

E. O9.5(u).3(sed)- 3(dle)-AirSecurityU313p6(o).1 Tf 1 -1.515 TD 0 Tc [(R

IV. Runtime Security of 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 a TrustZone-M-enabled MCU. In such a device, the application image consists of an app in the SW (Secure app), app in the NSC region (NSC app), and app in the NSW (NS app). We will focus on runtime software security in this section. That is, it is assumed that the adversary tries to compromise a target device through runtime software attacks.

We also assume that the attacker cannot alter application code on the flash, which can be set as a nonwritable through memory protection unit (MPU). However, the adversary can obtain the application binary code, e.g., through purchasing a device and disassembling it to understand the code and find the programming errors and software vulnerabilities.

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 is designed for providing a TEE that the NSW software cannot access directly, the TEE can only function 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 when software development in these regions is available. Even if the SW does not accept inputs from the Internet or peripherals and only the NSW communicates with the outside world, an attacker may compromise the NSW and feed malicious inputs into vulnerable NSC functions, which can access secure resources. Therefore, a vulnerable NSC function may lead to the entire SW to be compromised.

The ultimate goal of the adversary discussed here is to hijack the control flow or manipulate data so as to control the IoT device. To achieve such a goal, the adversary may want to send malicious payloads to the device and exploit programming errors in its software.

B. Runtime Software Attacks

Table II 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 BOF, and format string attacks, if requisite software flaws are 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 as follows.

TABLE II
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.

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

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 no-operation (NOP) 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 as follows.

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 addresses from 0x20000000 to 0x20004000, within which the higher halfword of any addresses is 0x2000, containing a null byte all the time. Taking Payload1 in Fig. 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 Fig. 2, where the entry of malicious code is placed at the bottom. Because of 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 first. 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 its higher halfword is kept 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 such as

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

Listing 2. Example of a vulnerable format string function.

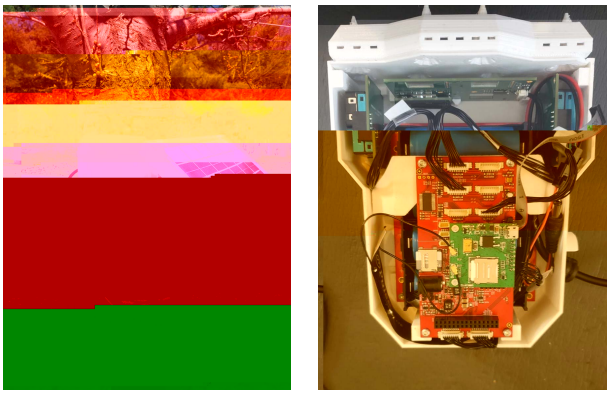
via overflowing an adjacent activated data chunk. By manipulating the pointers in the metadata, an adversary is able to corrupt arbitrary memory with arbitrary values [21].

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* and *%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 Fig. 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.

5) Attacks Against NSC Functions: The nonsecure software in the NSW may desire to use the secure services in the SW. For the sake of such requirements, 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: 1) they can be called from the NSW and 2) they have the privilege of accessing Secure resources since the NSC is a region within the SW. With such abilities, nonsecure 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 nonsecure callers.

As the gateway to the SW, the implementation of the NSC software should be particularly cautious. According to



(a) (b)

Fig. 4. Secure and trustworthy air quality monitoring device (STAIR). (a) STAIR in the field. (b) Internals of STAIR.

```
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;}
```

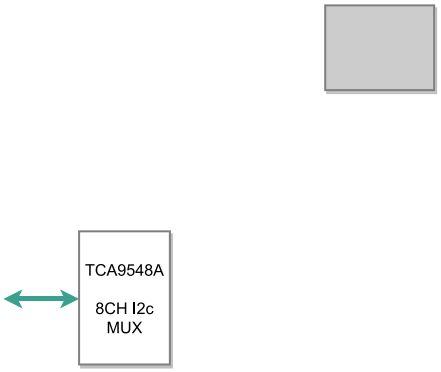
Listing 3. Example of a vulnerable NSC function.

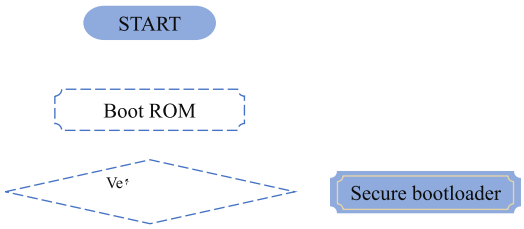
the guidance from ARM [22], hardware, toolchain, and software developers share a common responsibility to implement the NSC software securely. Though some requirements are offered in the guidelines, since the hardware and toolchain vary from vendors to vendors, there is no off-the-shelf solution to implementing trusted NSC software.

Securing the NSC functions is related to the research on interface security, such as [23] and [24], which analyze the potential vulnerabilities existing in TEE when interfacing the untrusted program execution to the trusted enclave. According to [23], the interface vulnerabilities are concentrated on invalid sanitization of the low-level application binary interface (ABI) and the high-level application programming interface (API). As for ABI, the adversary may control the low-level machine state such as register values transferred to the TEE. TrustZone is considered to be relatively resistant given its hardware design. A developer may pay more attention to developing the secure API, which takes potentially compromised parameters from the NSW.

We identify two potential 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 be present in the NSC functions as well and can be exploited by crafting Nonsecure 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.

The second pitfall comes from the untrusted pointer inputs. When nonsecure software passes pointer arguments to the SW through NSC functions, NSC functions should ensure that


TCA9548A
8CH I2c
MUX


START
Boot ROM
Ver
Secure bootloader

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 TEE 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 II that all software attacks may occur in both the SW (including NSC and SWX) and NSW. Recall that the CFI mechanism in [4] requires a TEE 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 provide the

solution is to compile the code with relative addressing flags. We use the GCC compiler with two specific compilation flags, including *-fpic* and *-mno-pic-data-is-text-relative*, so that all data accesses and most branches become PC-relative and can function after the image is relocated.

However, we still face two challenges after compilation with relative addressing flags: 1) NSC calls that call NSC functions in the SW and 2) function pointers that still use absolute addresses. We address the issue of NSC calls with static code patching at boot time and address the issue of function pointers with control flow correction at runtime.

Flashing Apps: We then flash the NS app to the start of the nonsecure flash, denoted as base app or base NS app, and secure app, including iASLR runtime program and the metadata, into the device.

Booting Device: When the system boots, as part of the secure bootloader, a *relocation engine* copies the base app image to a randomized address within the free nonsecure flash. The relocation engine sets the base app image to be nonexecutable through MPU so as to prevent the base app from being exploited since the base app image has a fixed base address and the attacker can know its layout. Therefore, there are two nonsecure app images in our system: 1) the base app image that is always located at the start of the nonsecure flash and 2) the relocated app image somewhere in the rest of the nonsecure flash. The relocation engine also performs *static code patching* to patch NSC calls in the relocated app as detailed in Section VI-B.

Running Relocated App and Control Flow Correction Engine: Now, system booting is finished and the boot code jumps to run the app in the relocated app image. At runtime, a *control flow correction engine* is used to handle absolute branches involving function pointers as detailed in Section VI-C.

B. Static Code Patching

Static code patching is applied to all NSC calls in the relocated image. An NSC call is a PC-relative branch, addressing the NSC function with the offset from the current PC value to the NSC function entry. Since the NSC functions are in the NSC region and are not relocated, the offset in each NSC call has to be patched. Recall we store the offsets of all the NSC calls in the metadata. The relocation engine can locate those NSC calls in the relocated image. For each NSC call, the offset of the relocated image relative to the base app image is added to its offset in the NSC call.

C. Control Flow Correction

At runtime, we use a control flow correction engine to patch all absolute branches introduced by function pointers. The correction engine is implemented in the Armv8-M *HardFault handler*. When an absolute branch in the relocated app image executes, the destination address of the absolute branch is an absolute address and points to the base app image. Since the base app image has been labeled as nonexecutable, any attempt of executing instructions within the base app image leads to a HardFault exception, which is handled by the *HardFault handler*. In this way, our correction engine is able to trap the control flow. The *HardFault handler* knows the address of the instruction that incurs the HardFault exception since the instruction's address is pushed onto the stack as the return address of the *HardFault handler*.

The correction engine then verifies whether the absolute address of interest is actually a destination address of an absolute branch by searching it in the metadata. Recall we store all the destination addresses of absolute branches in the metadata. The verification makes sure that the correction engine only handles the exceptions caused by absolute branches since there are other types of HardFault exceptions. After successful verification, the correction engine adds the offset of the relocated image to the absolute address of interest and derives the address of the target instruction in the relocated image. The *HardFault handler* changes its return address on the stack to the address of the target instruction so that the handler can return to run the target instruction.

D. Limitations

The ASLR scheme has its limitations. First, if an adversary knows the destination addresses of the absolute branches in the base app image, they may deploy a ROP attack, and divert the control flow to those addresses. Since the control flow correction mechanism cannot differentiate raised exceptions by such operations, the corresponding code in the relocated image then executes. However, in this case, the adversary has to use a whole function as a gadget to assemble a ROP chain. Such large gadgets are considered to be of very low quality to achieve certain operations [27]. An adversary can hardly launch a successful ROP attack. Second, our iASLR scheme has the storage overhead since there are two app images in the system: 1) the base app image and 2) relocated app image. Third, the boot time of an iASLR powered system will increase because of the relocating operations.

VII. Evaluation

We evaluate the five software attacks presented in Section IV. We are able to successfully perform these attacks against a TrustZone-M-enabled MCU, SAM L11. We also evaluate the effectiveness and performance of security mechanisms implemented in the STAIR air quality monitoring device.

A. Software Attacks

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 nonsecure 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 nonsecure and secure applications of SAM L11 and malicious payloads will be sent

TABLE III
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.

```
bool __attrib
```

TABLE IV
ENTROPY BOUND OF ARM CORTEX-M23/M33-ENABLED BOARDS

MCU/

TABLE V
TIME AND MEMORY PERFORMANCE OF APPLICATIONS WITH AND WITHOUT iASLR ENABLED

Application	Time performance (ms)				Memory performance (byte)			
	Boot time		Runtime		SW		NSW	
	Baseline (BL)	with iASLR	BL	with iASLR	BL	with iASLR	BL	with iASLR
STAIR-LWT	12	125	62636	64408	11552	12720	15736	16980
TZ-GetSta								

[17] "Side-Channel Attack." [Online]. Available: https://en.wikipedia.org/wiki/Side-channel_attack (accessed Dec. 9, 2021).
[18] "Return Oriented Programming (ARM32)." Azeria Labs. [Online]. Available: https://azeria-labs.com/return-oriented-programming-arm32/ (accessed May 1, 2021).
[19] "NX Bits—Microsoft Wiki—Fandom." Microsoft. [Online]. Available: https://microsoft.fandom.com/wiki/NX bit (accessed May 1, 2021).
[20] "Arm Heap Exploitation." Azeria Labs. [Online]. Available: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/ (accessed May 1, 2021).
[21] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *Proc. 22nd Int. Symp. Rel. Distrib. Syst.*, Oct. 2003, pp. 260–269.
[22] "ARMv8-m Secure Software Guidelines 2.0." Arm. [Online]. Available: https://developer.arm.com/docs/100720/0200/secure-software-guidelines (accessed May 1, 2021).
[23] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 1741–1758.
[24] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN attacks: On insecurity of enclave untrusted interfaces in SGX," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 971–985.
[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Security*, vol. 13, no. 1, pp. 1–40, Nov. 2009.
[26] "Address Space Layout Randomization." Wikiwand. [Online]. Available: https://www.wikiwand.com/en/Address_space_layout_randomization (accessed May 1, 2021).

Xinhui Shao received the B.S. degree in communication engineering from Shanghai University, Shanghai, China, in 2019. He is currently pursuing the master's degree in cyber science and engineering with Southeast University, Nanjing, China.

His current research interests include Internet of Things, privacy, and security.

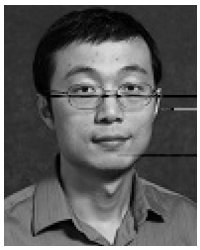
Zhen Ling (Member, IEEE) received the B.S. degree in computer science from Nanjing Institute of Technology, Nanjing, China, in 2005, and the Ph.D. degree in computer science from Southeast University, Nanjing, in 2014.

He is a Professor with the School of Computer Science and Engineering, Southeast University. His research interests include network security, privacy, and Internet of Things.

Prof. Ling won the ACM China Doctoral Dissertation Award and the China Computer Federation Doctoral Dissertation Award, in 2014 and 2015, respectively.

Haofei Yu received the B.S. degree in environmental engineering from Hangzhou Dianzi University, Hangzhou, China, in 2005, the M.S. degree in environmental engineering from the University of Shanghai for Science and Technology, Shanghai, China, in 2008, and the Ph.D. degree in environmental health from the University of South Florida, Tampa, FL, USA, in 2013.

He was a Postdoctoral Fellow with Georgia Institute of Technology, Atlanta, GA, USA, and Pacific Northwest National Laboratory, Richland, WA, USA. He is currently an Assistant Professor of Environmental Engineering with the University of Central Florida, Orlando, FL, USA. His research interests mainly focus on air quality modeling, emission estimation, exposure assessment, and low-cost air quality sensors.



Cliff Zou (Senior Member, IEEE) received the B.S. and M.S. degrees from the University of Science and Technology of China, Hefei, China, in 1999 and 1996, respectively, and the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Massachusetts at Amherst, Amherst, MA, USA, in 2005.

He is an Associate Professor with the Department of Computer Science, the Program Coordinator of both Digital Forensics Master program and Cyber Security and Privacy Master Program, University of Central Florida, Orlando, FL, USA. He has published more than 100 peer-reviewed research papers, and has obtained more than 7300 Google Scholar Citations. His research interests focus on cybersecurity and computer networking.

Xinwen Fu (Senior Member, IEEE) received the B.S. degree in electrical engineering from Xi'an Jiaotong University, Xi'an, China, in 1995, the M.S. degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 1998, and the Ph.D. degree in computer engineering from Texas A&M University, College Station, TX, USA, in 2005.

He is a Professor with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA. He was a Tenured Associate Professor with the Department of Computer Science, University of Central Florida, Orlando, FL, USA. He has published at prestigious conferences, including the four top computer security conferences (Oakland, CCS, USENIX Security, and NDSS), and journals, such as ACM/IEEE TRANSACTIONS ON NETWORKING (ToN) and IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING (TDSC). He spoke at various technical security conferences, including Black Hat. His current research interests are in computer and network security and privacy.