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) attack, heap-based BOF attack, format string attack, and attacks against nonsecure callable (NSC) functions in the context of TrustZone-M. We validate these attacks using SAM L11, a microchip MCU with TrustZone-M and provide defense mechanisms in the runtime software dimension of the proposed framework. The security framework is implemented with a full-fledged secure and trustworthy air quality monitoring device using SAM L11 as its MCU.**

***Index Terms*—Internet of Things (IoT), software security, TrustZone.**

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).

Xinwen Fu is with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA 01854 USA (e-mail: xinwenfu@cs.uml.edu).

Digital Object Identifier 10.1109/JIOT.2022.3144405

I. Introduction

INTERNET of Things (IoT) has been deployed in a wide range of application domains, including home appliances, medical instruments, smart buildings, industrial automation, and smart environment. In this article, we focus on IoT devices that use low-cost and resource-constrained microcontrollers (MCUs) and 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 [1], among which we are particularly interested in runtime software security of MCUs. Even if software integrity and authenticity can be verified at boot time via mechanisms such as secure boot, protecting software of embedded devices at runtime remains challenging due to the heterogeneity and constrained computational resources of MCUs [2]–[4].

TrustZone-M, the TrustZone extension for ARMv8-M architecture, is an emerging solution to the runtime software security of IoT devices [5]–[7]. Specifically, it provides resource-constrained MCUs a lightweight hardware-based solution to a trusted execution environment (TEE) for security-related software, i.e., the secure world (SW), which is isolated from the rich execution environment (REE), i.e., the nonsecure world (NSW), at the hardware level. The NSW software cannot access the SW resources directly. Instead, TrustZone-M provides a nonsecure 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.

In this article, we address the issues in securing TrustZone-M-based IoT devices and make the following major contributions. We propose a security framework for TrustZone-M-enabled IoT devices. The framework is designed from five dimensions, including hardware, boot-time software, runtime software, network, and over-the-air (OTA) updates. Care has to be taken to defeat various side-channel attacks (SCAs) existing in all IoT system components [8]–[11].

We are the first to perform a comprehensive security analysis of the runtime software security in TrustZone-M-enabled

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 unprotected programming and debugging ports.	(i) User authentication with different access levels. (ii) Disabling programming and debugging ports.
Boot-time software	Boot with untrusted/modified software.	Secure/trusted boot.
Runtime software	(i) Classic memory corruption attacks in the SW and NSW. (ii) NSC-specific attacks.	(i) Data Execution Prevention (DEP), control-flow integrity (CFI) and address Space Layout Randomization (ASLR). (ii) Sanitizing communication between the SW and outside world.
Network	MITM attack, sniffing attack, replay attack, etc.	Secure communication protocols, e.g., HTTPS, MQTT over TLS.
OTA update	(i) Insecure image downloading. (ii) Downgrade attack.	(i) Secure communication protocols. (ii) Anti-rollback prevention.
Various side channels	Timing attack, cache attack, power analysis, electromagnetic attack, allocation-based side channels, etc.	Side channel elimination, randomization, delay, etc.

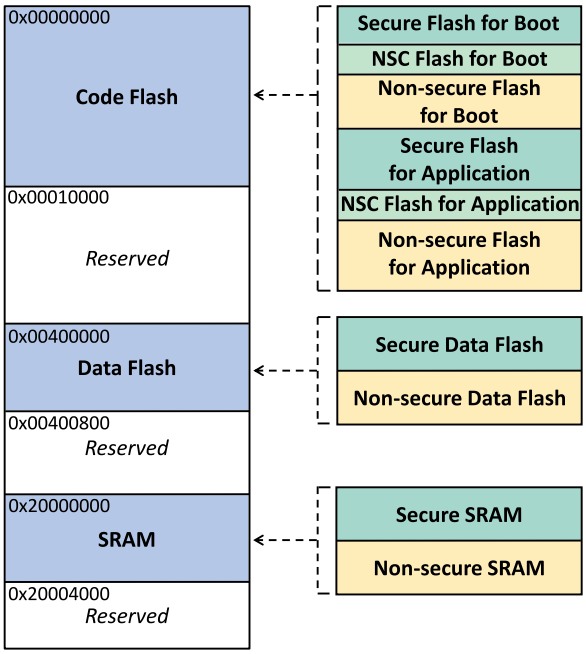


Fig. 1. Memory layout of ATSAML11E16A.

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.

Runtime software attacks aim at hijacking the program control flow by altering the control data (e.g., return address and function pointer) or changing program memory by manipulating noncontrol data [2]. Often in such an attack, an adversary corrupts the vulnerable memory by injecting a carefully crafted malicious payload, which eventually results in abnormal program behaviors.

Unlike computers or smartphones allowing authorized users to use user-defined applications, software in most IoT products is relatively fixed and can be modified only if a software update is required by the manufacturers. The relatively fixed software brings IoT devices stability, though on the flip side, the homogeneity of software means that a software exploit found in one device may also exist in hundreds and thousands of similar devices. As a result, large-scale attacks are prone to take effect.

III. SECURITY FRAMEWORK FOR TRUSTZONE-M-ENABLED IOT DEVICES

TrustZone-M is employed to provide IoT devices a TEE in order to ensure runtime security of software inside the SW. However, most of the device's functionalities unrelated to security are usually achieved in the REE, namely, the NSW. Compared to the SW, the NSW contains most of the program code and tends to communicate with the outside world much more frequently through different interfaces, and therefore, are more likely to be attacked. To maintain the security state of an IoT device during its lifetime, security principles and other defense mechanisms should work compatibly with TrustZone-M to provide full-scale protection. In this section, we propose a security framework, as presented in Table I, for TrustZone-M-based IoT devices, protecting devices according to five dimensions in terms of hardware, boot-time software, runtime software, network, and OTA update.

A. Hardware Security

To satisfy different demands for collecting varied ambient measurements, many IoT devices have to be deployed in open environments, hence, are physically exposed to the public. Devices that can be touched by adversaries, and have not been designed securely from the perspective of device hardware, tend to be extremely assailable to attacks. IoT devices usually provide hardware interfaces for software debugging and updates. Such programming interfaces can be categorized into debug ports and serial bootloader ports. An open debug port, such as serial wire debug (SWD) for ARM processors and joint test action group (JTAG) for many other integrated chips, will allow external access to the chip's memory contents, e.g., code, on-device data, registers, system configurations, and security keys for debugging and programming purposes. Besides debug ports, a serial bootloader, which uses a serial port, such as a universal asynchronous receiver/transmitter (UART) or serial peripheral interface (SPI) for updating software locally, is universal in MCU-based devices. Depending on the designs of the bootloader and transmission protocols, software might be read or overwritten through the serial ports. Therefore, measures should be taken to secure such programming ports from unauthorized access.

A TrustZone-M-enabled device suffers from the hardware interface attacks as well. Relying on the access privileges of the ports, attackers may compromise secure or nonsecure

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. Over-the-Air Update

OTA update is the process of distributing new software from the cloud to deployed IoT devices for updates. During this process, transmission should be encrypted so that the software would not be eavesdropped; authentication is required in order to confirm the downloaded contents are delivered from trusted sources; and integrity must be verified to avoid missing or tampered packets. As we discussed in Section III-D, employing secure network protocols is the solution.

Firmware rollback or downgrade attack [16] is another exploit existing in many OTA implementations aiming at bypassing the authentication mechanism using an old firmware with a valid signature so that adversaries can exploit bugs existing in the old firmware for further attacks. Intuitively, this issue can be addressed by comparing the version of firmware to be updated with the version of the current on-device firmware once new firmware is downloaded. The recorded current firmware version has to be stored in secure memory that can be accessed by trusted applications only. For example, once a new firmware is just delivered from the cloud and stored temporarily in spare memory, the OTA module needs to first verify its digital signature to make sure the integrity and authenticity of the firmware and the version value were not being compromised. If the signature is valid, the version value will be compared to the current version stored in the secure memory, and the firmware will be overwritten by the new firmware only if the new version value is larger than the current version.

In general, a secure OTA procedure consists of two stages, i.e.: 1) downloading new software through secure communication protocols and 2) validating the signature and version before overwriting the current software. The first stage involves secure network protocol that has been discussed in the previous section. Considering OTA update is closely relevant to the security of on-device software, a TrustZone-M-based MCU should embed OTA related software inside the SW. Moreover, both the secret key for verifying the signature and the current firmware version must be preserved in the SW or other secure memories to prevent possible leakage or attacker's modification.

F. Defense Against Side Channel Attacks

A side channel may exist in any components, including hardware, software, and networking of an IoT system. Sensitive information may be derived from timing of network packets, power consumption of cryptographic circuit, and electromagnetic radiation. To defeat side-channel attacks, we may try to eliminate side channels and reduce the leakage, for example, using special shielding to reduce electromagnetic emissions. Statistic strategies, such as randomization and delay, may be used to remove or lessen the relationship between the leaked information and the sensitive data [17].

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     strcpy(buf, input);}
```

Listing 1. Example of a function with BOF vulnerability.

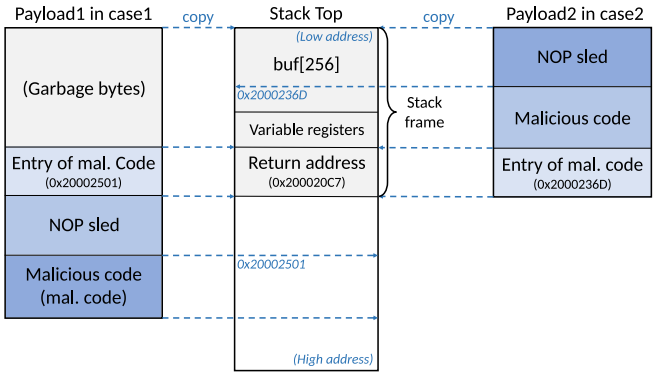


Fig. 2. Stack-based BOF attack for code injection.

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.

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 a return address, as illustrated in Fig. 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 [18]. Even if 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 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 nonexecutable memory [19], which prevents code execution from certain memory regions. However, an attacker can bypass such defense by leveraging CRA. A representative CRA is the ROP attack. Utilizing BOF to overwrite the return address,

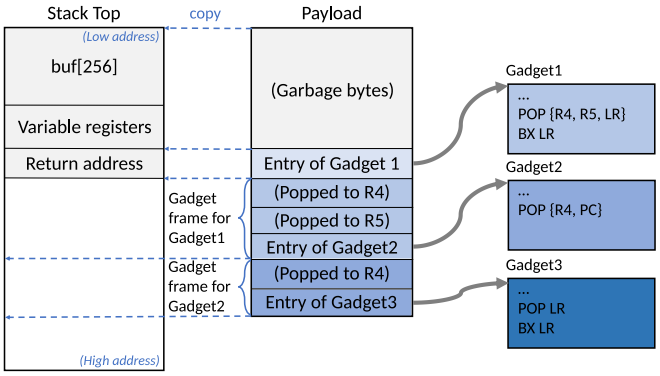


Fig. 3. ROP attack with three gadgets.

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 nonleaf subroutines, directly pops a word to the 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 send 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 includes 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 Fig. 3. The payload contains the entry of Gadget 1 and two gadget frames corresponding to Gadgets 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 [20]. The glibc allocator manages free chunks in a doubly linked list where each chunk 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

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

Listing 2. Example of a vulnerable format string function.

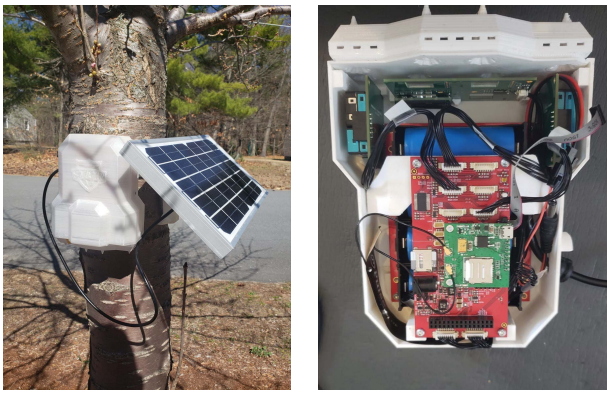
via overflowing an adjacent activated data chunk. By manipulating the pointers in the metadata, an adversary is able to corrupt arbitrary memory with arbitrary values [21].

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

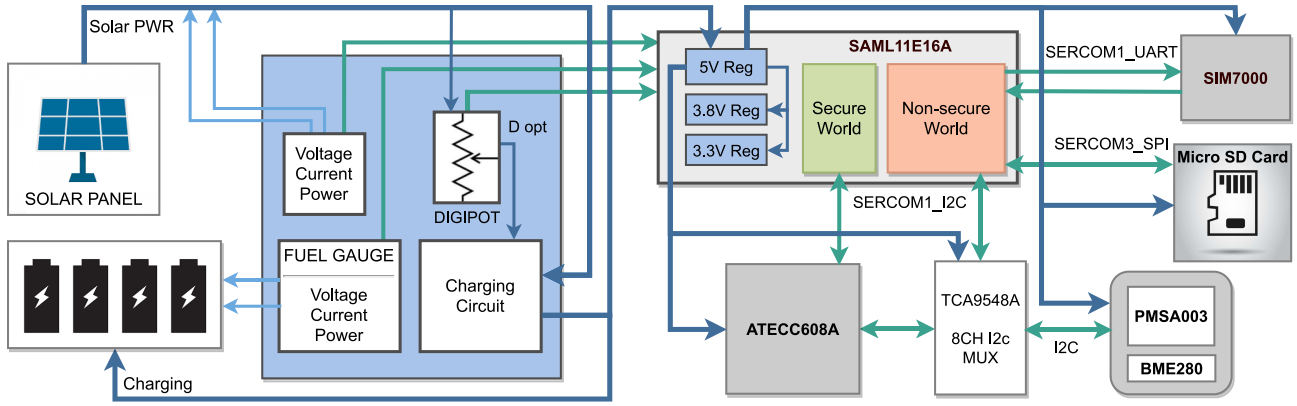


Fig. 5. Block diagram of the air quality monitoring device—STAIR.

these pointers point to the nonsecure memory. Otherwise, NSC and secure functions may assist the nonsecure 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 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.

V. Implementation: TrustZone-M-Enabled Air Quality Monitoring Device

Using SAM L11, we implement our security framework with a secure and trustworthy air quality monitoring device named STAIR. An air quality monitoring device is a small IoT device deployed for detecting ambient air quality measurements, such as mass concentrations of particulate matter (PM). The collected measurements are sent to the cloud for further analysis. Fig. 4 shows the external appearance and internal structure of the STAIR. It is worth noting that the security of the system relies on the hardware mechanisms (e.g., debug access level and boot ROM of SAML11) and adopted cryptographic algorithms [e.g., SHA256 and elliptic curve digital signature algorithm (ECDSA)]. It is reasonable to consider that compromising the hardware and well-known cryptographic algorithms is difficult. Therefore, these mechanisms are trustworthy if they are implemented without errors.

A. Device Design

Fig. 5 illustrates the hardware design of the STAIR device. The main controller board is built around a SAML11E16A MCU that provides the electronics system with a secure way to handle and transport sensor data such as the PM2.5 readings from a PLANTOWER PMSA003 laser dust sensor from the device to the remote server. The main board is embedded with some peripherals, including an I2C multiplexer (TCA9548A) to expand input ports for sensors, a mount to connect a SIM7000 cellular networking module providing a network stack for communication via secure protocols, such as HTTPS and MQTTS, an on-board cryptoauthentication chip (ATECC608A) to support security algorithms, and a built-in micro SD card for data storage. I2C, SPI, and UART protocols are used to communicate with these peripherals. Since ATECC608A is a security-critical component, it communicates with the SW through a Secure I2C port, while other components are connected to the NSW.

The whole device is powered by the battery management system (BMS), which is a large board that works in tandem with the main controller board, and handles the passive charging of the batteries using solar power from a solar panel and power regulation required to keep the device perpetually "on." The BMS contains an onboard I2C fuel gauge, a buck battery charge controller, a bidirectional current/power monitor chip, and a digital potentiometer to control the charging rate. Each of these onboard components use I2C to communicate and allow the main controller board to access data regarding the battery states.

B. Hardware Security

SAM L11 is equipped with one SWD debugging and programming interface. The SWD interface can be protected by SAM L11's debugger access level (DAL) technique. Providing three debug access levels, i.e., *DAL0*, *DAL1*, and *DAL2*, different access privileges can be granted to the external debuggers, and the transitions between different access levels are protected by security keys. DAL0, with the lowest access level, is blocked from accessing to any memory regions; DAL1 is only allowed to operate on the NS memory region; DAL2 has the highest debug access level and is able to access the whole chip without any restrictions. While transitions from higher levels to lower levels can be done with no preconditions, leveling up requires providing a specific chip erase key (CEKEY0, CEKEY1, or CEKEY2) and erasing the memory regions, which was not accessible to the original lower level but will be accessible after the level transition. For example, transitioning from DAL0 to DAL2 requires the debugger to provide CEKEY2 and execute ChipErase_ALL command for clearing both secure and nonsecure memories.

To ensure the debug port is in its most secure state, we set the SWD interface of STAIR to DAL0 and define three chip erase keys in SAM L11's NVM BOCOR row. The on-device chip erase keys are stored securely because the NVM BOCOR

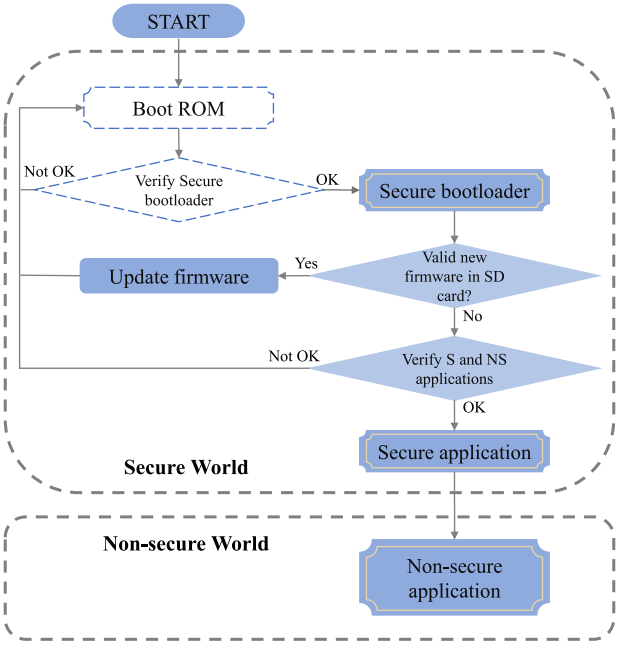


Fig. 6. Control flow of secure boot. The hollow boxes represent security checks provided by SAM L11, while other boxes are security checks designed by us.

row can only be programmed and read by secure software or external debugger with DAL2. In this way, only one who obtains CEKEY0 is able to access nonsecure memory and one who owns CEKEY0 and CEKEY1, or CEKEY2 can access both secure and nonsecure memories. The debugger cannot read memory contents through the SWD port, since chip erase commands have to be executed before DAL transfers to higher access levels.

C. Boot-Time Software Security

SAM L11 provides a secure boot mechanism based on its Boot ROM and NVM BOCOR row. Setting BOOTOPT fuse in BOCOR row to 2 or 3, Boot ROM will carry out validation checks on BOCOR row and the secure flash region for boot (named Boot Secure or BS region) using SHA256 hash with a key variant when the device boots. The key is preinstalled in BOCOR row, which can be hidden from other applications, therefore, would not be exposed to malicious people. The ignorance of checking the security states of other flash regions stimulates us to implement our own secure bootloader in the BS region for validating secure and nonsecure applications at boot, by which a chain of trust is constructed as presented in Fig. 6.

The boot sequence starts from the execution of Boot ROM. The Boot ROM checks the authenticity and integrity of BOCOR row and the secure bootloader, which will be executed next if all security checks are passed. The secure bootloader is in charge of validating both the secure and nonsecure applications before their executions. We implement this validation by using ECDSA signature verification. Specifically, the digests of the applications are calculated by SAM L11's on-device cryptoaccelerator using the SHA256 hash algorithm, and the digests and the signatures, which are appended at the end of applications, are sent to ATECC608 for signature verification, of which the public key is protected in ATECC608's secure memory.

D. Runtime Software Security

We secure the on-device runtime software for the bare-metal device from two aspects: 1) protecting the applications from classic memory corruption attacks and 2) checking any inputs from nonsecure applications to NSC functions.

A common way of compromising the runtime software is to inject malicious code to the device's memory and then divert the control flow to the malicious code. For SAM L11, an attacker may inject code to the code flash, data flash, stack, or heap. In our system, The stack and heap are parts of the SRAM; application code runs from the flash. The SRAM and data flash are only used for nonexecutable data. Therefore, we set SAM L11's SRAM and data flash to be nonexecutable in order to avoid code injection attacks. This is achieved by setting two BOCOR row fuses, namely, data flash is execute never (DXN) and RAM is execute never (RXN), to 1. In addition, the secure bootloader will set the secure and nonsecure code flash to be nonwritable, using the MPU, before starting to run the application code. The flash will be recovered to be writable when the device reboots so that only the bootloader is able to write to the code flash if an OTA update is required. By this means, attackers cannot alter any code at runtime.

As we discussed in Section IV, exploits in the SW might be triggered by parameters from the NSW through NSC functions. Therefore, inputs of NSC functions should be carefully checked. To this end, we validate any pointer parameters at the beginning of the NSC functions, ensuring they point to Nonsecure addresses. Data parameters are inspected as well with specific rules, by which crafted malicious payload would be filtered out. However, there is no general ways to sanitize any data inputs. Validation rules should be created depending on specific functions. We present our NSC function and parameter checks, as an example, in Section VII-B.

Besides, we introduce two possible security techniques against CRAs such as ROP. One of them, namely, the ASLR for MCU, is devised and implemented in our system.

1) Control Flow Integrity: CFI [25] 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. Nyman *et al.* [4] provided an implementation of CFI for TrustZone-M to protect the NSW. In it, a control flow graph (CFG) of the nonsecure program is constructed by static or dynamic analysis of its code and is saved in a nonwritable 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 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 indispensable secure storage and TEE 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. We have also implemented CFI, but the code instrumentation needed by CFI adds too much code to implement a full version of the air quality monitoring device. We will not include the evaluation of CFI in this article.

2) Address Space Layout Randomization: ASLR [26] is another security technique used to mitigate CRAs. With ASLR, the memory layout of a process's address space is randomized instead of being fixed, so that the attacker cannot predict the location of memory of interest, such as the stack, libraries, heap, and code modules. To defeat potential CRAs, we design and implement an ASLR scheme for Cortex-M processors and will introduce it in Section VI.

E. Network Security

The cellular module SIM7000 provides the network stack to the STAIR. Through the cellular network, our air quality device connects to the AWS IoT platform for the sake of cloud services. To enable two-way authentication, message encryption and integrity checks, MQTT over TLS is adopted as the communication protocol.

F. OTA Update Security

To ensure the authenticity, integrity, and privacy of the OTA procedure, HTTPS serves as the communication protocol for securely downloading newly released software. The downloaded software is temporarily stored in the SD card and is authenticated through the ECDSA signature verification provided by ATECC608A.

To enable anti-rollback prevention, we preserve the current firmware version in SAM L11's secure data flash so that only secure applications are able to read or write it. Once a new firmware is downloaded to the SD card, the secure application will compare the new firmware version to the current one and will continue to upgrade the firmware only if the new version is larger than the current version. The stored firmware version will also be updated to the newest version. Otherwise, the downloaded firmware will be deleted from the SD card.

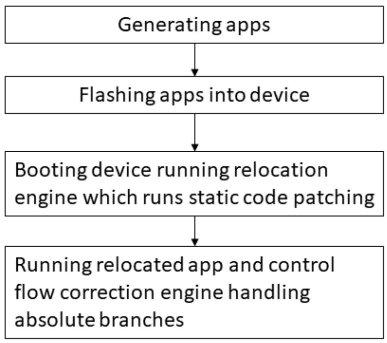


Fig. 7. Workflow of iASLR.

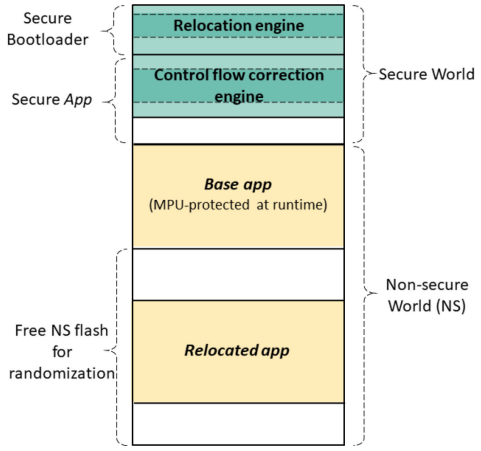


Fig. 8. Flash layout of an iASLR-enabled system.

VI. iASLR–Image-Based ASLR

In this section, we first introduce the workflow of iASLR and the challenges, and then address the challenges. We discuss the limitations of iASLR at the end of this section.

A. Workflow and Challenges

Fig. 7 gives the workflow of iASLR and Fig. 8 illustrates the memory (flash) layout of the iASLR powered system. Recall we run the system directly from the flash. Our iASLR always keeps one copy of NSW app image at the start of the nonsecure flash (denoted as base app) and relocates the base app within the free nonsecure flash every time the device boots. In a TrustZone-M-enabled system, although we can relocate the secure app in the SW, we assume the secure app is secure and will focus on the design of relocating the nonsecure (NS) app located in the nonsecure world.

Generating Apps: First, we generate the secure app and the NS app. We run Python scripts on the two apps to collect the size of free nonsecure flash (i.e., the nonsecure flash excluding the base app), the addresses of all NSC calls, and the destination address of each absolute branch instruction in the base app as the metadata. The metadata are stored in the SW and will be used for image randomization and code patching.

One challenge with relocating the base app is the absolute addresses in the binary code when the app is generated with the default compiler settings. These absolute addresses only work with a fixed base address (0x0 in our case). An intuitive

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.
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.

through the UARTs to trigger the attacks. The sizes of the payloads and experiment results are given in Table III.

Experimental Results: In the BOF-based code injection attack, we configure the stack to be executable, which is commonly configurable in 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 prestored 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 IV-B2. We take a basic nonsecure application image, which only initializes necessary peripherals, as an example and search all the subroutine epilogues in it. The Capstone disassembly engine [28] is used to dissemble and search in the binary code. The size of the example image is 4.14 kB 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.

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 preinjected malicious code. The malicious code is later executed when that function is called.

As we stated in Section IV-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

```
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.

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 nonsecure 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.

B. Security of the Implemented Air Quality Monitoring Device

We evaluate the security mechanisms that we have used for the STAIR device.

Debug Interface: The STAIR device is equipped with one SWD debug interface. We connect a laptop, on which Atmel Studio, the Microchip's integrated development environment (IDE) for programming SAM MCU, has been installed, to the debug interface through an SWD debugger. Using Atmel Studio, we are able to identify the SAM L11 MCU of the target STAIR device. However, memory contents, such as firmware image and system configurations, i.e., UROW and BOCOR, are invisible to the users. The IDE shows that the debug access level is set to DAL0; hence, the 32-bits CEKEY1/CEKEY2 is required to raise the level to DAL1/DAL2 for debugging and programming the NSW/the whole chip. The sensor will be protected from attacks through the debug port only if CEKEY1 and CEKEY2 are not divulged.

Secure Boot: The secure boot mechanism involves two stages of verification. At the first stage, the Boot ROM verifies the secure bootloader using the SHA256 hash function with an authentication key. To evaluate its effectiveness, we

```
bool __attribute__((cmse_nonsecure_entry))
    nsc_sha256(uint8_t* input, uint32_t input_length
    , uint32_t* output){
    if((input < NS_FLASH_START) || (input >
    NS_FLASH_END && input < NS_RAM_START)
    || (output < NS_FLASH_START) || (output >
    NS_FLASH_END && input < NS_RAM_START)){
        return false;
    }
    if((input < NS_FLASH_END && input_length >
    NS_FLASH_END - input) || input_length >
    NS_RAM_END - input){
        return false;
    }
    sec_sha256(input, input_length, output);
    return true;}
```

Listing 5. NSC function for providing secure SHA256 algorithm to the NSW.

manipulate one byte of the secure bootloader from 0x00 to 0xFF and reboot the device. As a result, the device halts at the boot stage. If a debugger is connected, a fault exception will be triggered.

Defense Against Classic Memory Corruption Attacks: To defend against the code injection attack, we set both data flash and RAM to be nonexecutable. We verify this by launching two BOF-based code injection attacks in the NSW, which eventually divert the control flows to code snippets stored in either nonsecure data flash or nonsecure RAM. In both cases, fault exceptions are generated when control flows are diverted. Moreover, secure flash and nonsecure flash are nonwritable during application execution in order to protect application code from being altered. An ROP attack is conducted trying to divert the control flow to a memory writing function and provide the address of malicious code injected on the RAM as the parameter of the writing function. As could be expected, the MCU ignores the writing operation—it does not write the new code to the target address and just continues to execute the next instruction.

Defense Against NSC-Specific Attacks: The STAIR sensor application contains one NSC function as shown in Listing 5. Two parameters of this function are pointers. We check them in order to ensure both are within the NSW. Another parameter is an integer, which defines the length of the input array. We check the lower boundary of this value because the input array must be within the NSW; in other words, the end of the input array should not exceed the end of the NSW.

OTA Update: We evaluate the anti-rollback prevention by downloading an image with a valid signature but lower version to the SD card. After we reboot the device, this image is detected by the secure bootloader. Then, the bootloader deletes the image from the SD card because of the invalid image version and continues to boot with the original image.

C. Performance Evaluation

We evaluate time efficiency of different security mechanisms, i.e., secure boot and OTA validation, and transmitting air quality measurements to AWS IoT. We repeat each test for 30 times and present the evaluation results in Figs. 9–11. The

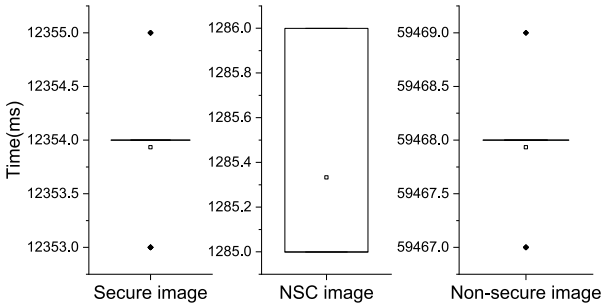


Fig. 9. Overheads of updating secure image (5.59 kB), NSC image (32B), and nonsecure image (29.8 kB) through secure OTA.

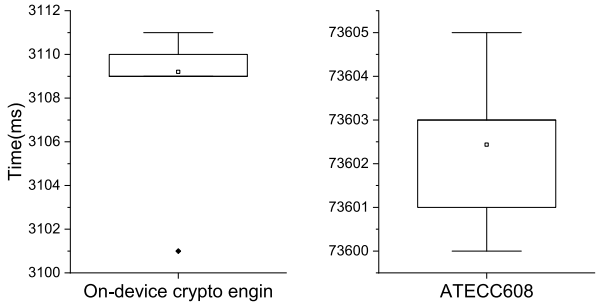


Fig. 10. Overheads of secure boot using on-device crypto engine or ATECC608 for hashing.

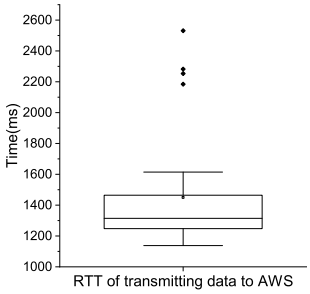


Fig. 11. RTT of transmitting data to AWS IoT through MQTTS protocol.

overheads for secure OTA and secure boot are both acceptable for a small air quality sensor, which does not boot and update frequently and does not need much interaction.

The secure OTA is able to update the three images (i.e., the SW, the NSC region, and the NSW) separately. Accordingly, we evaluate the time efficiency of validating each of the images as shown in Fig. 9. It is observed that the time taken by validation is approximately proportional to the size of an image. The very large portion of the overhead comes from the transmission and response time when hashing the image via ATECC608.

For secure boot, image validation involves calculating the signatures of images and verifying the signatures. Since both SAM L11's crypto accelerator and ATECC608 support SHA256 hash algorithm, we evaluate the performance of secure boot with each of them. As a result, the on-device hardware cryptoaccelerator shows a much better performance with average time efficiency of 3.109 s than the ATECC608, which needs 66.913 s in average.

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

MCU/Development Board	Flash Size	Entropy Bound (bits)
Microchip SAML11 [12]	64KB	15
ARM musca b1 [30]	4MB	21
Nordic nRF5340 [31]	1MB	19
NXP LPC55S69 [32]	640KB	18.32
STM32L562E-DK [33]	512KB	18
NuMicro M2351 [34]	512KB	18

While sending data to the AWS IoT platform, we use the MQTTS protocol, which requires mutual authentication and traffic encryption. It takes 1300 ms on average to send a message and get the response from the server.

The reported performance of our device is appropriate for our purpose of environmental monitoring, which does not require the real-time interaction with users or much computing at the device end. Our STAIR device uses the ARM Cortex-M23 32-bit processor core operating at up to 72-MHz frequency. The successor of Cortex-M23, the TrustZone-enabled Arm Cortex-M33 32-bit RISC core, operates at a frequency of up to 110 MHz. It can be observed that those chips are designed for low-power systems that are not compute intensive. The system developers shall select appropriate chips based on their application requirements. For example, Cortex-A-based application processors with the TrustZone technology are high performance processors for applications such as smartphones and can be used for IoT applications that require real-time user interaction and performance-intensive computing.

D. Evaluation of iASLR for IoT

We analyze iASLR from the perspectives of security and performance. Time and memory overheads are evaluated with three applications. One is the lightweight version of our STAIR application (without remote control and OTA). The other two are demos of SAM L11 provided by Microchip [29].

1) Security Analysis: The efficacy of the ASLR scheme against CRAs depends on the randomness of the relocated image address. The randomness is usually quantified by entropy and formulated as in (1), where S is the entropy of the iASLR-enabled system, f is the size of free flash space that can be used for relocation, f_s is the size of the entire flash that can be used to run code, and i is the application image size (byte). Recall that the base address of the application image must be even

$$S = \log_2 \frac{f-i}{2} < \log_2 \frac{f}{2} < \log_2 \frac{f_s}{2} = \log_2 f_s - 1. \quad (1)$$

Therefore, the entropy has a bound of $\log_2 (f_s/2)$.

Table IV lists TrustZone-M powered MCUs from known companies, their embedded flash size that can be used to run code, and the bound of the system entropy. It can be observed that MCUs with larger flash will have better entropy. We recommend that an alert shall be sent to the system administrator if an attack triggers errors and causes system rebooting.

2) Time and Memory Overheads: Our iASLR for IoT devices introduces both boot-time and runtime overheads. At boot time, the image relocation and NSC call patching involve large amount of *memory read* and *write* operations. As presented in Table V, when iASLR is enabled, the larger the NS app is, the longer time the boot procedure takes. At runtime, the CF correction engine traps and patches every absolute branch, thus introducing runtime delay. All three applications in Table V show small runtime overheads no more than 36%, and two of them have overheads of 2.8% and 3.5%. Note that we have removed all big time delay functions (delay for more than 0.5 ms) from these three applications during experiments since the delays do not create overhead but will dilute the runtime overheads introduced by iASLR. In practice, there may be large time delays in IoT device code.

Memory overhead of an iASLR-enabled system exists in both the SW and NSW. First, all the code and metadata of iASLR implementation are in the SW. Second, compiling with required flags changes the size of the nonsecure application. For most of the cases in Table V, the overhead is no greater than 10% while there is one with overhead at 17%.

Based on the evaluation and analysis above, we believe iASLR is suitable for long-running IoT devices.

1) *Security:* When an iASLR-enabled IoT device is attacked, the system may crash and hang. Many embedded devices have watchdog timers [35], which can automatically reboot the devices when program crash is detected, while IoT devices may be rebooted manually too. When a device reboots, iASLR relocates the image and protects the device. The attacker may continue to perform the attack and guess the location of the relocated image repeatedly. That is, why we recommend an IoT device shall have relatively large flash to achieve large entropy and an alert shall be sent to the system administrator if there is a crash.
2) *Performance:* We have performed evaluation of the iASLR-enabled IoT devices and the performance is promising while we will investigate how to further reduce the overhead of iASLR and improve the performance in future work.

VIII. RELATED WORK

We now introduce related work on TrustZone-M. Traditional defense mechanisms against software attacks can be resource intensive in terms of storage and computation power. Lightweight techniques are often needed for those resource-constrained IoT devices. Abera *et al.* [36] proposed a remote attestation method, exploiting the separated two execution environments in TrustZone-M-enabled IoT devices, to monitor the CFI of running program. Nyman *et al.* [4] implemented an interrupt-aware CFI in the Cortex-M SoC. These mechanisms focus on securing only the nonsecure applications, assuming TrustZone is set correctly and the SW is secure.

Security issues may also exist in TrustZone itself. Research has been performed regarding the security issues in ARM TrustZone and its implementation. Rosenberg [37] identified an integer overflow vulnerability in QSEE, the Qualcomm TrustZone implementation. Kanonov and Wool [38] analyzed security of KNOX, the Samsung's security platform constructed based on TrustZone, and discovered several design

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-GetStart	< 0.1	26.3	31.1	32.2	8984	10548	9724	10428
TZ-SecureDriver	< 0.1	35.7	1321.7	1788.3	11456	12600	6980	7424

flaws and vulnerabilities. Guo *et al.* [39] discussed the technical principle of ARM TrustZone and a vulnerability found in its cache architecture. Koutroumpouchos *et al.* [40] preformed analytical exploration on vulnerabilities of TrustZone-A-based TEE and gave a taxonomy of the attacks targeting TrustZone-A. Cerdeira *et al.* [41] presented Systematization of Knowledge (SoK) on the vulnerabilities of Cortex-A TrustZone-assisted TEE. However, our work focuses on the Cortex-M TrustZone. Because of the implementation differences of TrustZone-A and TrustZone-M, the security analysis on TrustZone-A cannot be applied uniformly to TrustZone-M.

The research on TrustZone-M application has been emerging. Iannillo and State [42] proposed a framework for the security analysis of TrustZone-M. However, their work does not identify concrete vulnerabilities/attacks against TrustZone-M. Jung *et al.* [7] designed a secure platform based on the platform security architecture (PSA) with a brief discussion of possible attacks. O'Flynn and Dewar [8] discovered an SCA in SAM L11, using power analysis to breach cryptographic algorithms. Instead, our work demonstrates five types of realistic attacks, breaching the software security of TrustZone-M.

To defeat CRAs, we may use the ASLR technique. The researchers have proposed to create firmware with different memory/flash layouts for each individual IoT device at the compilation time [13]. However, the memory layout does not change after compilation. In such a scheme, the attacker will succeed after trying a number of times since the memory layout is the same even after the device reboots. In this article, we design and implement an iASLR specifically for IoT devices in order to defend against CRAs. iASLR relocates an application image every time the device boots.

IX. CONCLUSION

In this article, we presented a security framework for IoT devices that use TrustZone-M-enabled MCUs from perspectives of hardware, boot-time software, runtime software, network, and OTA update. We performed systematic runtime software security analysis of TrustZone-M-enabled IoT devices. We presented potential pitfalls of TrustZone-M programming and five potential software attacks against TrustZone-M, including the stack-based BOF attack for code injection, ROP, heap-based BOF attacks, format string attacks, and attacks against NSC functions. We validated these attacks on a TrustZone-M-enabled MCU, SAM L11. To defend against these attacks, guidelines for the overall system security of TrustZone-M-enabled IoT devices are presented. We implemented the proposed security framework for an air quality monitoring device and evaluated its performance. One lesson we learned from implementing the air quality monitoring device is the MCU shall be carefully chosen with enough amount of flash to host the application code.

ACKNOWLEDGMENT

Any opinions, findings, conclusions, and recommendations in this article are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] H. HaddadPajouh, A. Dehghantanha, R. M. Parizi, M. Aledhari, and H. Karimipour, "A survey on Internet of Things security: Requirements, challenges, and solutions," *Internet Things*, vol. 14, Jun. 2021, Art. no. 100129.

[2] A. Mohanty, I. Obaidat, F. Yilmaz, and M. Sridhar, "Control-hijacking vulnerabilities in IoT firmware: A brief survey," in *Proc. 1st Int. Workshop Security Privacy Internet Things (IoTSec)*, 2018, pp. 1–4.

[3] K. V. English, I. Obaidat, and M. Sridhar, "Exploiting memory corruption vulnerabilities in connman for IoT devices," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, Portland, OR, USA, 2019, pp. 247–255.

[4] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses (RAID)*, 2017, pp. 259–284.

[5] "Trustzone for Cortex-M." Arm. [Online]. Available: https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m (accessed May 1, 2021).

[6] 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 Trans. Comput. Sci. Eng.*, Dec. 2019, doi: 10.12783/dtcse/cisnrc2019/33312.

[7] J. Jung, J. Cho, and B. Lee, "A secure platform for IoT devices based on arm platform security architecture," in *Proc. 14th Int. Conf. Ubiquitous Inf. Manag. Commun. (IMCOM)*, Taichung, Taiwan, 2020, pp. 1–4.

[8] C. O'Flynn and A. Dewar, "On-device power analysis across hardware security domains," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 4, pp. 126–153, 2019.

[9] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay, "How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip," in *Proc. IFIP Int. Conf. Inf. Security Theory Pract.*, 2017, pp. 93–109.

[10] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 181–194.

[11] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Proc. IFIP Int. Workshop Inf. Security Theory Pract.*, 2011, pp. 224–233.

[12] "Saml11 Xplained Pro Evaluation Kit," Microchip. [Online]. Available: https://www.microchip.com/DevelopmentTools/ProductDetails/PartNO/DM320205 (accessed May 1, 2021).

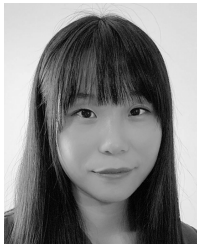
[13] A. A. Clements *et al.*, "Protecting bare-metal embedded systems with privilege overlays," in *Proc. IEEE Symp. Security Privacy (SP)*, San Jose, CA, USA, 2017, pp. 289–303.

[14] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu, "On runtime software security of TrustZone-M based IoT devices," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Taipei, Taiwan, 2020, pp. 1–7.

[15] D. Song *et al.*, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Security Privacy (S&P)*, San Francisco, CA, USA, May 2019, pp. 1275–1295.

[16] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade attack on TrustZone," 2017, *arXiv:1707.05082*.

[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).
[27] A. Follner, A. Bartel, and E. Bodden, "Analyzing the gadgets," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2016, pp. 155–172.
[28] "The Ultimate Disassembly Framework—The Ultimate Disassembler." [Online]. Available: http://www.capstone-engine.org/ (accessed May 1, 2021).
[29] "Atmel Start." Microchip. [Online]. Available: https://start.atmel.com/ (accessed May 1, 2021).
[30] "Musca-B1 Test Chip Board." [Online]. Available: https://developer.arm.com/tools-and-software/development-boards/iot-test-chips-and-boards/musca-b-test-chip-board (accessed Sep. 30, 2021).
[31] "nRF5340." [Online]. Available: https://www.nordicsemi.com/Products/nRF5340 (accessed Sep. 30, 2021).
[32] "LPC55s69-EVK: LPCXpresso55s69 Development Board." [Online]. Available: https://www.nxp.com/design/development-boards/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK (accessed Sep. 30, 2021).
[33] "STM32l562E-DK Discovery Kit." [Online]. Available: https://www.st.com/en/evaluation-tools/stm32l562e-dk.html (accessed Sep. 30, 2021).
[34] "Numicro M2351 Series—A TrustZone Empowered Microcontroller Series Focusing on IoT Security." [Online]. Available: https://www.nuvoton.com/products/microcontrollers/arm-cortex-m23-mcus/m2351-series/?__locale=en (accessed Sep. 30, 2021).
[35] J. Ganssle. "A Designer's Guide to Watchdog Timers." Digikey. 2012. [Online]. Available: https://www.digikey.com/en/articles/a-designers-guide-to-watchdog-timers
[36] T. Abera *et al.*, "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 743–754.
[37] D. Rosenberg, "QSEE TrustZone kernel integer over flow vulnerability," in *Proc. Black Hat Conf.*, 2014, p. 26.
[38] U. Kanonov and A. Wool, "Secure containers in android: The Samsung KNOX case study," in *Proc. 6th Workshop Security Privacy Smartphones Mobile Devices*, 2016, pp. 3–12.
[39] P. Guo, Y. Yan, C. Zhu, and J. Wang, "Research on arm TrustZone and understanding the security vulnerability in its cache architecture," in *Proc. Int. Conf. Security Privacy Anonymity Comput. Commun. Storage*, 2020, pp. 200–213.
[40] N. Koutroumpouchos, C. Ntantogian, and C. Xenakis, "Building trust for smart connected devices: The challenges and pitfalls of TrustZone," *Sensors*, vol. 21, p. 520, Jan. 2021.
[41] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted tee systems," in *Proc. IEEE Symp. Security Privacy (S&P)*, San Francisco, CA, USA, 2020, pp. 18–20.
[42] A. K. Iannillo and R. State, "A proposal for security assessment of TrustZone-M based software," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Berlin, Germany, 2019, pp. 126–127.



Lan Luo received the B.S. degree in electrical engineering from the Civil Aviation University of China, Tianjin, China, in 2015, and the M.S. degree in computer engineering from the University of Central Florida, Orlando, FL, USA, in 2018, where she is currently pursuing the Ph.D. degree in computer science.

Her research interests mainly cover security and privacy of Internet of Things, security of embedded system, network and software security, and trustworthy computing.

Yue Zhang received the Ph.D. degree from the College of Information Science and Technology and College of Cyber Security, Jinan University, Guangzhou, China, in 2020, under the supervision of J. Weng.

He also studied and worked with the University of Central Florida, Orlando, FL, USA, and the University of Massachusetts Lowell, Lowell, MA, USA, under the supervision of X. Fu. He has published papers in international conferences and journals, such as USENIX Security, IEEE INFOCOM, IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, and RAID. His research focuses on system security, especially IoT security.

Clayton White received the B.S. degree in electrical engineering and the B.S. degree in computer engineering from the University of Central Florida, Orlando, FL, USA, in 2021.

He is currently employed as a Hardware Systems Integrator with Google, Mountain View, CA, USA. His current research interests include Internet of Things, embedded data acquisition, and various VLSI design projects.

Brandon Keating received the B.Sc. and M.Sc. degrees in electrical engineering from the University of Massachusetts Lowell, Lowell, MA, USA, in 2020 and 2021, respectively.

He currently works as an Electrical Design Engineer with Globus Medical, Audubon, PA, USA, concentrating on medical robotics which specialize in spinal and cranial surgery operations. His technical interests cover a multitude of robotics-focused design platforms, including 3-D CAD design, electrical design, PCB layout, and firmware.

Bryan Pearson received the B.S. degree in computer science from Stetson University, DeLand, FL, USA, in 2018. He is currently pursuing the Ph.D. degree in computer science with the University of Central Florida, Orlando, FL, USA.

His papers have been published at conferences, such as INFOCOM, ICC, ICPADS, and IFIP. His research interests include software and network security of Internet of Things devices and communication protocols, binary analysis, and fuzz testing.

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.

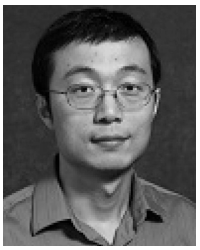
Zhen Ling (Member, IEEE) received the B.S. degree in computer science from Nanjing Institute of Technology, Nanjing, China, in 2005, and the Ph.D. degree in computer science from Southeast University, Nanjing, in 2014.

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.