TOWARDS SECURE AND TRUSTWORTHY IOT SYSTEMS

by

LAN LUO
M.S. University of Central Florida, 2018
B.S. Civil Aviation University of China, 2015

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2022

Major Professor: Cliff C. Zou & Xinwen Fu

ABSTRACT

The boom of the Internet of Things (IoT) brings great convenience to the society by connecting the physical world to the cyber world, but it also attracts mischievous hackers for benefits. Therefore, understanding potential attacks aiming at IoT systems and devising new protection mechanisms are of great significance to maintain the security and privacy of the IoT ecosystem. In this dissertation, we first demonstrate potential threats against IoT networks and their severe consequences via analyzing a real-world air quality monitoring system. By exploiting the discovered flaws, we can impersonate any victim sensor device and polluting its data with fabricated data.

It is a great challenge to fight against runtime software attacks targeting IoT devices based on microcontrollers (MCUs) due to the heterogeneity and constrained computational resources of MCUs. An emerging hardware-based solution is TrustZone-M, which isolates the trusted execution environment from the vulnerable rich execution environment. Though TrustZone-M provides the platform for implementing various protection mechanisms, programming TrustZone-M may introduce a new attack surface. We explore the feasibility of launching five exploits in the context of TrustZone-M and validate these attacks using SAM L11, a Microchip MCU with TrustZone-M enabled.

We then propose a security framework for IoT devices using TrustZone-M enabled MCUs, in which device security is protected in five dimensions. The security framework is implemented and evaluated with a full-fledged secure and trustworthy air quality monitoring device using SAM L11 as its MCU. Based on TrustZone-M, a function-based ASLR (fASLR) scheme is designed for runtime software security of IoT devices. fASLR is capable of trapping and modifying control flow upon a function call and randomizing the callee function before its execution. Evaluation results show that fASLR achieves high entropy with low overheads.

The dissertation is dedicated to my beloved family.

ACKNOWLEDGMENTS

This work would not have been possible without the support and contribution of many people that I would like to appreciate.

My great gratitude to my doctoral advisor, Dr. Xinwen Fu, for his considerable and continuous assistance, support, and encouragement of my Ph.D. study. I have less research experience when I first began my Ph.D. study. It was Dr. Fu's guidance that helped me become more skilled and professional in fields of cyber security. I would also like to express my sincere thanks to my advisor, Dr. Cliff Zou. My research would not be possible without his guidance and support. And I would like to appreciate Dr. Zhen Ling, who gave me great help on my research. Besides, I would like to extend my appreciation to my dissertation committee Dr. Yan Solihin and Dr. Haofei Yu for their valuable time and comments on my research and dissertation.

I would like to thank my collaborators, including Yue Zhang, Bryan Pearson, Xinhui Shao, Chao Gao, Huaiyu Yan, Yumeng Wei, Clayton White, Brandon Keating, and Rajib Dey. I have learned various knowledge, skills, and positive attitude towards work and life from them during our collaboration.

During my Ph.D. study, UCF campus provides me a fertile learning environment and sufficient hardware support. All staffs I have met in UCF are professional and kind. They always give me proper assist once I need it. Thank UCF campus and everyone that I met in UCF.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The Internet of Things (IoT) integrates smart devices capable of accessing network into our daily routine. The booming IoT makes IoT devices and networking spread over various fields, from controlling smart home appliances to monitoring environmental activities across broad ocean. However, the boom of IoT and lack of security mechanisms also attract crafty hackers to compromise such systems for benefits. IoT devices are connected to remote server, edge devices, or other smart devices via Internet for data exchange. Insecure network protocols can allow adversaries to eavesdrop privacy data, impede normal data transmission, or even disturb the whole system behavior. Moreover, adversaries may intrude into IoT devices, compromising the software to take control of the devices as intents. Therefore, understanding potential attacks aiming IoT network and device software is of great significance to maintain the security and privacy of the whole IoT ecosystem.

Modern IoT devices are fragile to various network and software attacks mainly for the following reasons: (1) Some developers lack awareness of potential threats to IoT devices. Networking and software security are rarely considered in the designs. (2) Applications in IoT devices are prone to be buggy since they are usually written in unsafe programming languages such as C and C++. (3) IoT devices usually use resource-constraint SoC, for example microcontroller (MCU), which heavily restricts the implementation of software-based protections when considering time and space overheads. (4) IoT devices are usually bare-metal devices or use light-weight real-time operating system (e.g., free-RTOS). In both cases, there is no mature protections provided in system level. (5) The heterogeneity among SoC hardware make it complex to implement hardware-based security mechanisms. (6) Many IoT devices are deployed in open environments hence are directly exposed to adversaries. In this dissertation, we focus on network-based and software-based threats targeting resource-constraint IoT devices. As the countermeasures, we also propose a security framework and innovative protection mechanisms.

To explore the potential threats to IoT networking and demonstrate the corresponding results severe consequences of cyber security issues in low-cost sensor networks, we first analyze a real-world air quality monitoring system. In this work, we present a systematic analysis of cyber security and data integrity of the researched IoT system. Through a series of probing, we are able to identify multiple security vulnerabilities in the system, including unencrypted message communication, incompetent authentication mechanisms, and lack of data integrity verification. Exploiting these vulnerabilities, we have the ability of "impersonating" any victim sensor in this system and polluting its data using fabricated data. We also provide strategies to defend the discovered attacks. The research result was published in Sensors 18 as "On the Security and Data Integrity of Low-cost Sensor Networks for Air Quality Monitoring".

We then focus on the software security issues of low cost IoT devices. Attacks targeting the device software at runtime are challenging to defend against due to the heterogeneity and constrained computational resources of MCUs. TrustZone-M, the TrustZone extension for ARMv8-M architecture, TrustZone-M, a TrustZone extension for MCUs, is an emerging security technique fortifying MCU based IoT devices. We presents the first security analysis of potential runtime software security issues in TrustZone-M enabled MCUs. We explore the stack-based buffer overflow (BOF) attack for code injection, return-oriented programming (ROP) attack, heap-based BOF attack,format string attack, and attacks against Non-secure Callable (NSC) functions in the context of TrustZone-M. These attacks are validated using the Microchip SAM L11 MCU, which uses the ARM Cortex-M23 processor with TrustZone-M. We published the research results and implementations in GLOBECOM 2020-2020 IEEE Global Communications Conference as "On Runtime Software Security of TrustZone-M Based IoT Devices".

The research based on TrustZone-M then was extended to a journal paper "On security of TrustZone-m based IoT systems" published in IEEE Internet of Things Journal in 2021. In this paper, we discuss the overall security of TrustZone-M based IoT devices. To address the security issues, we

propose a comprehensive security framework and implement an air quality monitoring device for demonstration. Besides, we design and implement the first image-based address space layout randomization (ASLR) scheme for IoT devices, denoted as iASLR. iASLR for IoT devices is unique since it relocates an image every time the device boots while the image layout is randomized only one time in related work [21]. We design the static code patching and control flow correction schemes to tackle the addressing issues caused by image relocation.

We also devise fASLR, a function-based address space layout randomization (ASLR) mechanism, to defend against code reuse attacks in resource-constrained IoT devices. fASLR utilizes the ARM TrustZone-M technique and the memory protection unit (MPU). It loads a function from the flash and randomizes its entry address in a randomization region in RAM when the function is called. We design novel mechanisms on cleaning up finished functions from the RAM and memory addressing to deal with the complexity of function relocation and randomization. fASLR can run an application even if it cannot be completely loaded into RAM for execution. We test fASLR with 21 applications. fASLR achieves high randomization entropy and incurs runtime overhead of less than 10%. The research result was concluded in our research paper "fASLR: Function-Based ASLR for Resource-Constrained IoT Systems", which has been submitted to ESORICS 2022.

This dissertation covers researches of papers published or submitted by the author. In Chapter 2, we present related work in the fields of IoT networking security and device runtime software security. In Chapter 3, we present our analysis and attacks towards a real-world air quality monitoring system. A comprehensive security analysis is performed on TrustZone-M enabled IoT devices in Chapter 4. After the security analysis, we propose a security framework and an image-based randomization scheme for TrustZone-M enabled IoT devices in Chapter 5. We conclude all researches in Section 7.

CHAPTER 2: LITERATURE REVIEW

In this chapter, we discuss previous work related to our researches.

Security of Air Pollution Monitoring Systems

In regulatory monitoring activities, cyber security is an important consideration. For example, most regulatory monitoring sites are isolated from public access, and the entire process of data collection, such as data transmission and storage, are usually handled via secured protocols and networks. Security is also an important consideration for research focusing on connected IoT devices [18] in the field of computer science. Much work has been done on security enhancement of IoT devices, such as encryption algorithms, communication protocols, front-end sensor data protection, and back-end IT system protection [37, 72, 28, 11, 66, 51, 45] while there is no consensus on a generic framework securing IoT systems given the varieties and scales of IoT systems, and much research on IoT security and privacy is desired. For example, we have exploited various vulnerabilities in smart plugs and cameras [42, 41].

TrustZone-M

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. [4] proposes a remote attestation method, exploiting the separated two execution environments in TrustZone-M enabled IoT devices, to monitor the control flow intergrity (CFI) of running program. [55] implements an interrupt-aware CFI in the Cortex-M SoC. These mechanisms focus on securing only the Non-

secure applications, assuming TrustZone is set correctly and the Secure World 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 [63] identify an integer overflow vulnerability in QSEE, the Qualcomm TrustZone implementation. Kanonova et al. [34] analyze security of KNOX, the Samsung's security platform constructed based on TrustZone, and discover several design flaws and vulnerabilities. In [29], Guo et al. discuss the technical principle of ARM TrustZone and a vulnerability found in its cache architecture. Koutroumpouchos et al. [36] preform analytical exploration on vulnerabilities of TrustZone-A based TEE and give a taxonomy of the attacks targeting TrustZone-A. Cerdeira et al. [17] present 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.

Research on TrustZone-M application has been emerging. Iannillo et al. [31] propose a framework for the security analysis of TrustZone-M. However, their work does not identify concrete vulnerabilities/attacks against TrustZone-M. Jung et al. [33] design a secure platform based on the Platform Security Architecture (PSA) with a brief discussion of possible attacks. O'Flynn et al. [56] discover a side-channel attack 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 code reuse attacks, we may use the address space layout randomization (ASLR) technique. Researchers have proposed to create firmware with different memory/flash layouts for each individual IoT device at the compilation time [22]. 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 paper, we design and implement an image-based ASLR (iASLR) specifically for IoT devices in order to defend against code reuse attacks. iASLR relocates an application image every time the device boots.

CHAPTER 3: NETWORK SECURITY OF IOT SYSTEM — CASE STUDY OF A LOW-COST AIR QUALITY MONITORING SYSTEM

With the rapid technology advancements, low-cost and portable air pollution monitoring systems have gained much attention in the past few years. Though low-cost air quality sensors are less accurate, precise and reliable compared with FRM and FEM instruments, their performances, however, are generally good enough for qualitative characterization of air quality status, and they only cost a fraction of the cost of FRM or FEM instruments. The emerged air quality sensors are essentially IoT devices since they are connected to the Internet, and may be remotely configured and controlled. These sensors can be quickly deployed to establish a connected air quality monitoring network to characterize spatio-temporally resolved pollutant concentration variations at local scale, and provide opportunities to considerably enhance the capabilities of existing monitoring networks. We find that data integrity and system security are rarely considered during the design and deployment of low-cost and connected air quality monitoring systems, which leaves tremendous room for unwelcome and damaging cyber intrusions.

Case Study: the PurpleAir System

We use PurpleAir, a low-cost air quality monitoring system, to demonstrate the potentially severe consequences of cyber security issues in low-cost sensor networks for air quality monitoring. Among many available low-cost and connected air quality monitoring platforms, PurpleAir is one of the most popular, and one of the largest systems that provide affordable and continuous air quality monitoring capabilities to broad communities. The PurpleAir system is able to measure ambient mass and particle number concentrations of PM2.5, a critical air pollutant regulated under the National Ambient Air Quality Standards. PM2.5 is considered to be one of the most harmful

air pollutants and contributes to millions of premature deaths annually worldwide [40, 60]. Concentrations of PM1 and PM10 are also measured by PurpleAir, and are processed the same way as PM2.5 in this system. Without loss of generality, we restrict our analysis on PM2.5 for simplification while the analysis and results are also valid for other measurements as well as PM2.5.

A-II Dual Laser Sensor. At the core of the PurpleAir system is the PurpleAir low-cost sensor. A PurpleAir sensor is able to continuously monitor multiple environmental and air quality related metrics, including temperature, humidity and PM2.5. On the PurpleAir website, the vendor provides three different types of sensors for choice. In this paper, we perform our analysis using the A-II model sensor, which uses two laser particle counters to provide two independent parallel channels (named channel A and channel B in this paper) of real time measurements and is the recommended sensor model from PurpleAir. To protect the air quality monitoring network under this investigation, we denote the sensor model as A-II.

Sensor Setup. To enable the sensor for the whole monitoring system, a user shall follow the initialization procedure as described below.

1. Installation: A sensor can be installed either inside or outside. To keep a sensor working in good condition and producing precise sensor outputs, the user should follow instructions provided by PurpleAir for finding a suitable installation location. Also, a sensor shall be able to connect to a stable WiFi connection if Internet access is desired.

2. WiFi Configuration: Once powered up, the sensor starts to act as an access point (AP) named "AirMonitor_xxx", where "xxx" is specific to each sensor. Users can connect to it via computer or smartphone and access the sensor's web user interface (UI). On the web UI, users are able to provide home WiFi configuration information, after receiving which, the sensor will stop its AP mode and connect to the WiFi for Internet access. The home WiFi configuration is saved in the sensor for future automatic connections.

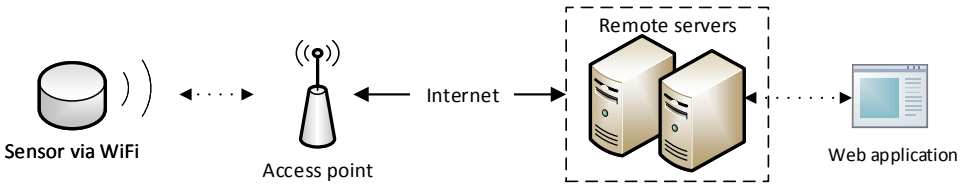


Figure 3.1: System Architecture from User's Perspective.

3. Sensor Registration: To conveniently check a sensor's collected data and utilize the graphical presentation of historical sensor data provided by the PurpleAir Map web application, a user is required to conduct a prior online sensor registration at the PurpleAir website, submitting important sensor information including sensor name, sensor geographical location, sensor MAC address, and a specific user's email. The provided email must match the one used during sensor purchase, otherwise the registration would be denied. The requirement of providing the associated user email is a prevention measure for avoiding unauthenticated sensor registration and utilization by malicious attackers even if they somehow possess the sensor MAC address. After completing the registration process, sensor data will be visualized on the PurpleAir Map at the registered geographical location.

User-Perspective Architecture. After successfully setting up the sensor, by observing the PurpleAir system's operating mechanism, we can obtain a preliminary view of the system architecture from a user's perspective. We illustrate this architecture in Figure 3.1.

From a user's perspective, PurpleAir can be recognized as an IoT system that consists of user-end air quality monitoring sensors and remote servers. The sensor connects to the user provided WiFi router, via which it then accesses the Internet. Utilizing the network connection, the sensor automatically uploads air quality measurement data to the remote servers.

For viewing uploaded sensor data, PurpleAir provides users with the following options.

1. **From PurpleAir Map:** In this option, users are able to use a web application called "Map", which is an integrated map system overlaid with all public sensors located at their reported geographical locations. Users can use the sensor geographical location or sensor name to pinpoint the desired sensor unit. In the "Map" application, a user is able to view both real-time and historical statistical sensor data in numerical and graphical presentations.

2. **From downloaded csv files**: In the second option, there exists a webpage for cvs format historical sensor data downloading. For each sensor unit, data items appear at an interval of approximately 80 s and are divided into four files, with two for each sensor channel.

3. **From sensor's web UI**: The last option for viewing sensor data is accomplished via a sensor's web UI. When a user connects to the sensor directly at bootstrapping (before connecting it to the Internet), a user can also access a web interface that shows real time sensor measurements. Nonetheless, this approach has its deficiencies: First, the sensor is disconnected from the Internet and the PurpleAir servers, interrupting data uploading and resulting in a permanent gap of the server's historical sensor data collection. Second, this option does not have any historical or graphical data illustration accessibility.

We believe that most users shall adopt the PurpleAir Map application as it is the most convenient and intuitive sensor data inspection choice. This observation is further consolidated by the popularity of publicly visible sensors on the PurpleAir Map application, as presented in Figure 3.2.

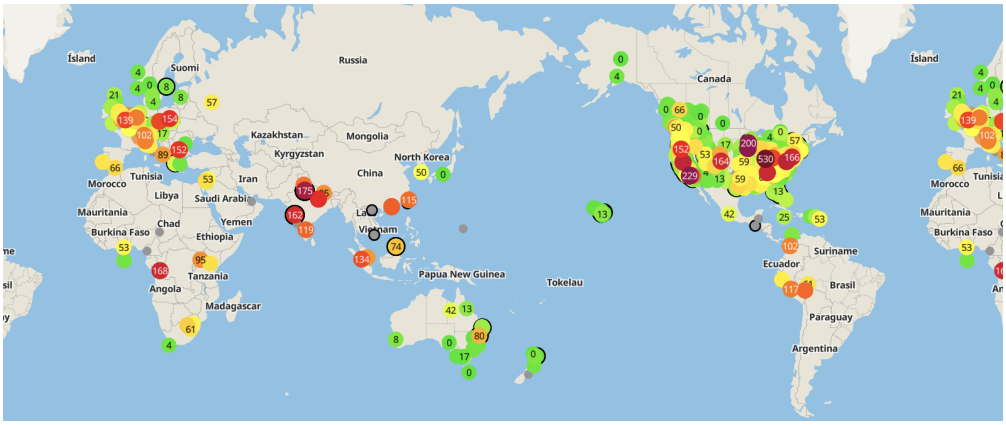


Figure 3.2: Screenshot from the PurpleAir Map in reference for application popularity.

Methodology of Analyzing Network Security of the PurpleAir System

To explore the architecture of the PurpleAir system, we construct an experimental environment capable of capturing network traffic between PurpleAir sensors and their servers, as shown in Figure 3.3. Firstly, we establish a wireless local area network (WLAN) using a wireless router with Internet access. A sensor is configured to connect to this router wirelessly. Secondly, a laptop installed with an ARP spoofing tool, the "ARPspoof" from the "dsniff" tool package [69], is connected to the same WLAN. ARP spoofing is an attack methodology where Address Resolution Protocol (ARP) messages with false association of a target IP address to the attacker's MAC address are fabricated and broadcasted, with the purpose of redirecting network traffic of interest from the original destination to the attacker, such that communication interception and alteration could be performed. The ARPspoof tool is configured to reroute all communications between the PurpleAir sensor and its web servers to the laptop for network traffic analysis, interception, and possible modification.

Through this environment setup, we can successfully modify the WLAN topology without any change of physical network connections for enabling the functionality of the network interception

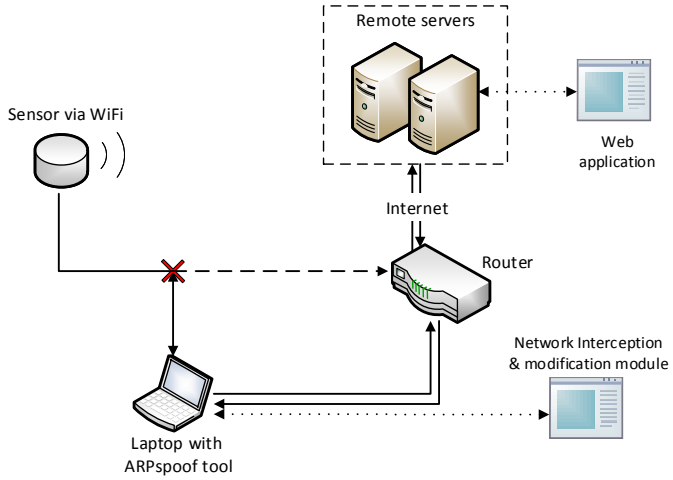


Figure 3.3: Experiment Setup.

and modification module. This enables the possibility of capturing sensor network traffic, inspecting the sensor message structure, and carrying out MITM attack against the sensor. Messages originated from and destined to the sensor can now be freely monitored and modified at the laptop if necessary. Additionally, to assist with the initial message inspection, we install Wireshark [79], a network protocol analyzer, on the laptop to capture and analyze network traffic of the sensor for further exploration.

By analyzing the captured traffic packets, we find that the PurpleAir sensors use HTTP as the communication protocol. This drives us to choose mitmproxy [24] for further network traffic analysis. Mitmproxy is a HTTP/HTTPS proxy that can perform the MITM attack, where attackers hijack an ongoing network communication and perform network traffic analysis or information alteration. With its Python scripting API, a user is able to execute customized scripts and manipulate network traffic automatically. Here, we run mitmproxy in its transparent mode, which does not require particular configuration of the sensor of interest. We are able to observe corresponding HTTP requests and responses, and changes reflected on the map for modified messages.

Analyzing Communication Protocol

After analyzing the captured traffic between the A-II sensor and its servers, we also discover a few characteristics about sensor communication: **Firstly**, the communications between a sensor and servers are not encrypted. Messages sent by a sensor use the HTTP GET request method. **Secondly**, a sensor communicates with its servers automatically in a periodical pattern. In each period, a A-II sensor constructs six HTTP non-persistent connections with three for each independent sensor channel (corresponding to each independent particle counter inside the sensor) as shown in Figure 3.4. Messages constructed for each channel are identical in structure. **Thirdly**, for three messages from a channel in each communication period: the first message contains measurement data of PM2.5 and two API keys, and is sent to www.PurpleAir with the sensor's MAC address as identification; the other two messages contain measurement data and API keys, and are sent to api.thingspeak.com. ThingSpeak further allocates two channels for each independent particle counter in the sensor (thus four ThingSpeak channels in total for a sensor unit with dual laser particle counters). To upload data to the ThingSpeak server, each ThingSpeak channel will be bonded with a unique API key, which is used for identity verification. Therefore, only with a correct key contained in the message can a sensor upload data through the corresponding ThingSpeak channel.

Message Composition

To better understand the message composition, the structure of these messages is now stated as follows. Without loss of generality, we use three messages for channel A as an example, and provide message notations for later reference:

- Message **M-1A**: A sensor constructs a HTTP connection to www.PurpleAir and sends a HTTP GET request which contains the sensor MAC address, two keys for sensor channel

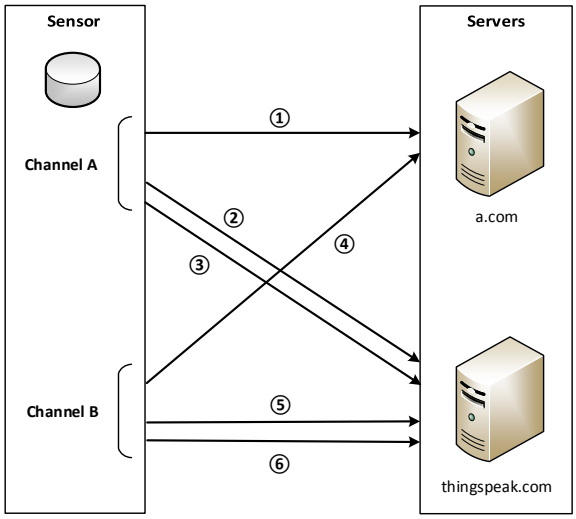


Figure 3.4: Communication Protocol between an PurpleAir Sensor and Servers.

A (named K-1A and K-2A) used in later communication with api.thingspeak.com, and all measurement data from sensor channel A. The MAC address is used for one-way device identification and the server will respond with a message corresponding to the validity of the MAC and the ThingSpeak keys.

- Message **M-2A**: A sensor connects to api.thingspeak.com and sends a request with K-1A and a part of measurement data from channel A for data analysis and visualization purposes.
- Message **M-3A**: A sensor connects to api.thingspeak.com again, but sends a HTTP request with K-2A and the remaining measurement data from channel A not involved in M-2A.

The same pattern will be utilized again for constructing three non-persistent HTTP connections (M-1B to M-3B) for channel B with corresponding ThingSpeak keys K-1B and K-2B. Each communication period of six messages lasts about 80 s, with three channel A messages coming first, followed by three channel B messages.

Table 3.1: Data format description for M-1A and M-1B.

HTTP Request Header	Description	HTTP Request Header	Description
mac	MAC address	pm2_5_atm	PM2.5 w/ correction
lat	latitude of the sensor location	pm10_0_atm	PM10.0 w/ correction
lon	longitude of the sensor location	pm1_0_cf_1	PM1.0 w/o correction
key1	key K-1A (K-1B)	pm2_5_cf_1	PM2.5 w/o correction
key2	key K-2A (K-2B)	pm10_0_cf_1	PM10.0 w/o correction
uptime	uptime in sec	p_0_3_um	particles$\leq$0.3 μm count/dl
rssi	signal strength	p_0_5_um	particles$\leq$0.5 μm count/dl
current_temp_f	temperature	p_1_0_um	particles$\leq$1.0 μm count/dl
current_humidity	humidity	p_2_5_um	particles$\leq$2.5 μm count/dl
current_dewpoint_f	dewpoint temperature	p_5_0_um	particles$\leq$5.0 μm count/dl
pressure	pressure	p_10_0_um	particles$\leq$10.0 μm count/dl
pm1_0_atm	PM1.0 w/ correction		

Data Format and AQI

Comprehensive understanding of data formats embedded in communication messages between a sensor and servers is essential for both system architecture analysis and sensor data pollution. Recall that the HTTP GET request is used by sensors. In such a request, sensor data is presented in the header section with a "name: value" pair format. These headers are stored in the URL of the HTTP request, hence can be easily identified.

As aforementioned, three respective messages for channels A and B are entirely identical in structure. Therefore, when dissecting internal data formats of the messages for each communication period, it suffices to look at the three unique data formats derived from three messages of each channel. We depict these three unique data formats in Table 3.1 for messages M-1A and M-1B, Table 3.2 for messages M-2A and M-2B, and finally Table 3.3 for messages M-3A and M-3B.

From the internal data format as presented in Tables 3.1–3.3, we are capable of obtaining a well-rounded observation of data communication between sensors and servers. We also discover that sensor data presentation on the PurpleAir Map not only uses raw sensor data, but also analytic

Table 3.2: Data format description for M-2A and M-2B.

HTTP Request Header	Description
key	ThingSpeak key K-1A (K-1B) for identification
field1 - field8	a part of sensor measurements

Table 3.3: Data format description for M-3A and M-3B.

HTTP Request Header	Description
key	ThingSpeak key K-2A (K-2B) for identification
field1 - field10	the other part of sensor measurements

results calculated from raw sensor data. In fact, Air Quality Index (AQI), a major quantitative index for determining air quality, is not transferred via direct communication between a sensor and servers. The PurpleAir claims to use the "Federal Environmental Protection Agency (EPA) Air Quality Index (AQI) scale". Accordingly, we present the AQI calculation formula in Equation (3.1) [76]:

$$AQI = \frac{I_{high} - I_{low}}{C_{high} - C_{low}}(C - C_{low}) + I_{low}, \tag{3.1}$$

where C is the pollutant concentration, C_{low} is the concentration breakpoint that is $\leq C$, C_{high} is the concentration breakpoint that is $\geq C$, I_{low} is the index breakpoint corresponding to C_{low}, and I_{high} is the index breakpoint corresponding to C_{high}. Furthermore, as expressed in Equation (3.1), AQI is computed based on a selected reference pollutant. We discovered that PurpleAir calculates two separate AQI values using PM2.5 and PM10 by the same formula (with corresponding parameters). The data pollution mechanisms for PM2.5 and PM10 are identical. Therefore, we restrict our later description and experiments to PM2.5.

Table 3.4: Response format from the www.PurpleAir server.

Scenarios	Response
MAC address and keys correct	geographic coordinates
MAC address correct but keys wrong	geographic coordinates and correct keys
MAC address wrong	NOT FOUND

Server Response Format

The www.PurpleAir server would send back HTTP responses when they receive HTTP requests originated from the PurpleAir sensor. Exemplar message exchanges are presented in Table 3.4. If a message is sent to the server with both correct sensor MAC address and correct keys, the server will send back the geographic coordinates of this registered sensor. If the MAC address is correct but the keys are invalid, the response will return both the geographic coordinates and the correct keys for future communication between the sensor and the api.thingspeak.com server. Finally, If the MAC address is invalid, the server responds with "NOT FOUND", indicating that this MAC address does not belong to any registered PurpleAir sensor. These responses enable us to verify possible sensor MAC addresses and also obtain their corresponding ThingSpeak keys.

Network-based Attacks in Three Different Scenarios

We consider three data pollution methodology against the PurpleAir system in different scenarios. We first introduce how to pollute data of a victim sensor if we physically possess the sensor, or if the victim sensor's MAC address is known. We then discuss how to enumerate the MAC addresses of every PurpleAir sensor so that we can pollute any sensor of the PurpleAir system.

Scenario A: Possessing a Sensor

As a rudimentary case, the first scenario (denoted as Scenario A) for data pollution is that we physically possess the PurpleAir sensor. Recall in Section 3, we identify the following features which contribute to data pollution: Firstly, the network traffic between PurpleAir sensors and servers are in plaintext; secondly, only the sensor MAC address is used for identification. Therefore, it suffices to use the same environment setup as in Figure 3.3, which is used in system architecture discovery as well, for launching the pollution.

According to the experiment setup, the data pollution can be sufficiently accomplished by the MITM attack. Through the environment setup, we are capable of intercepting messages between a sensor and its servers and modifying them freely as needed. To accomplish this in practice, the HTTP proxy tools such as mitmproxy (on Linux) and Fildder (on Windows) can be used. Such tools are employed for listening to bidirectional sensor-server communications, and plug-in scripts are used to manipulate messages automatically in these communications. If the message contains raw sensor data of interest, we modify the message with selected data values before forwarding it to the original server destination. Since sensor data is stored in the HTTP request header when uploaded in the clear data format as mentioned in Section 3, locating specific sensor data by corresponding header name and rewriting it to any value we desire is a sound approach with no apparent obstacles.

Scenario B: Knowing a MAC

In this scenario (denoted as Scenario B), physical access to an PurpleAir sensor is no longer permitted. A plausible approach is to fabricate all messages from scratch to imitate the victim sensor if we know the MAC address of the victim sensor.

There are indeed multiple ways to obtain a sensor's MAC address without physical possession. We now present one representative approach with only publicly available information within the scope of the PurpleAir system. In this approach, we utilize the observable geographical locations of registered sensors on the PurpleAir Map. Thus the issue becomes, given the geographical location of a sensor, without direct physical access to the sensor and the possibility of direct network manipulation, can its MAC address still be attained? To carry out this task, we leverage wardriving, which refers to scanning and sniffing for WiFi network information in a moving vehicle by using a laptop or other computer devices.

In demonstration, we ask a volunteer to set up an PurpleAir sensor within his/her household. We know no information for the sensor of interest, including its MAC address, but only the geographical location of the volunteer's household. By wardriving around the volunteer's household using the popular sniffing tool kismet [35], we successfully intercept WiFi network communication information in the surrounding area. Although traffic from multiple active WiFi networks are merged together through this process, the desired sensor MAC address can still be spotted with ease. This is because, by convention, the first 6 hex digits (prefixes) of the 12 digits MAC address represents a specific manufacture. The PurpleAir sensors contain a specific WiFi microchip ESP8266 in which the prefix of the MAC address belongs to one of several pre-assigned prefixes owned by the manufacturer Espressif Systems. MAC address prefixes allocated to a vendor such as Espressif can be looked up at various websites [78], by which 24 MAC address prefixes are found given to Espressif Systems (Table 3.5). By using prefix patterns, we match and find the actual sensor MAC via wardriving without encountering any ambiguous situations in sensor MAC recognition during wardriving. Nonetheless, even if ambiguity appears, we can leverage message responses from the PurpleAir server to easily distinguish the actual sensor MAC from other candidates.

Table 3.5: Discovered MAC address prefixes assigned to Espressif Systems.

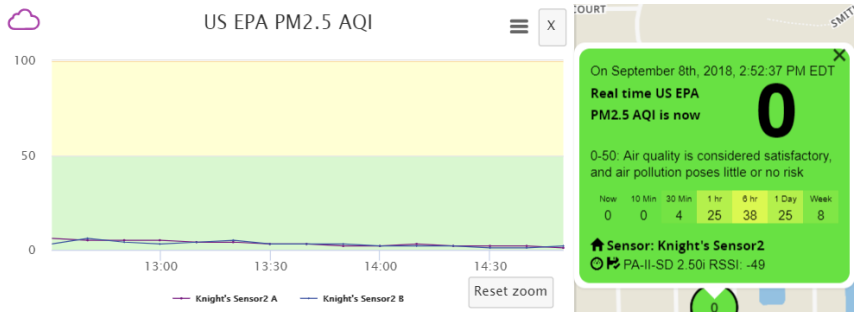
MAC Prefixes	**MAC Prefixes**
18:FE:34	24:0A:C4
24:B2:DE	2C:3A:E8
30:AE:A4	3C:71:BF
54:5A:A6	5C:CF:7F
60:01:94	68:C6:3A
84:0D:8E	84:F3:EB
90:97:D5	A0:20:A6
A4:7B:9D	AC:D0:74
B4:E6:2D	BC:DD:C2
C4:4F:33	CC:50:E3
D8:A0:1D	DC:4F:22
EC:FA:BC	80:7D:3A

Once knowing a specific registered sensor's MAC address, we pollute its data sent to the PurpleAir system by creating a fake sensor (a computer program), fabricate messages according to the discovered data formats, and send the fabricated data to corresponding servers. The fabricated messages will contain the victim sensor's MAC address, and will be accepted by the servers as authentic data from the specific victim sensor. We call this attack as a spoofing attack since the fake sensor pretends to be the victim sensor and sends fake data to PurpleAir web servers.

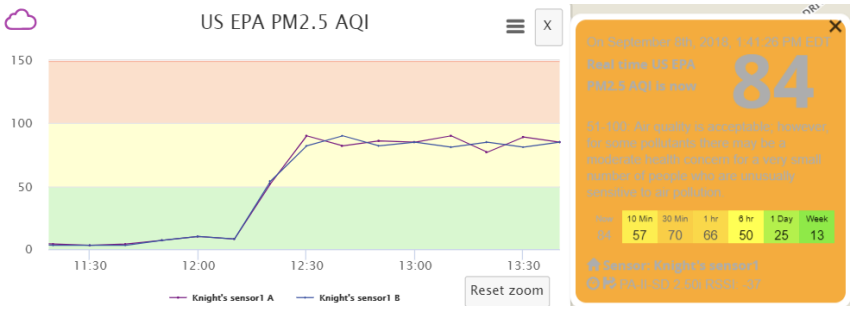
In the spoofing attack, PurpleAir servers receive two sets of data for one sensor: authentic data from the victim sensor, and fabricated data from the fake sensor. The servers merge the two sets of

data and use the merged data to indicate air quality. Inevitably, the authentic data from the victim sensor are "polluted" by the fake data.

In addition, we can vary the data transmission frequency of counterfeit sensor data to better suppress authentic data, to conceal malicious activities from the owners of the victim sensor and the servers, and to achieve a desired level of air pollution. We find that each of the two channels within a real sensor transmits data at an interval of approximately 80 s. If our fake sensor also sends counterfeit data every 80 s, the PurpleAir servers would receive both authentic and fabricated sensor data at the same frequency, and the two data sets will be averaged. The resulting effect of data pollution, taking the AQI calculated based on measured PM2.5 as an example, is shown in Figure 3.5. One can observe two phenomena: **Firstly**, the polluted data presents many ups and downs. This fluctuation is the result of averaging the received real and fake measurements within each 10-min period. When the fake data sending frequency is relatively low, it is likely that the number of received fake data samples varies slightly between different periods. Meanwhile, the fake data is usually much larger (or smaller) than the real data. Hence, receiving even one more fake message may lead to apparent fluctuation in the averaged result. For the real measurements, slight fluctuation exists as well due to natural variations of pollutant concentrations in the ambient environment. Therefore, the polluted data fluctuation is actually the combined result of the variations in real data and the instability caused by received fake data. **Secondly**, the AQI is not the intended value, which should be 151, as suggested by the fabricated message, due to average of both authentic and fabricated data. Such phenomena, especially the data fluctuation, could raise the possibility of detection by the sensor owner or the PurpleAir system. It is of great interest to better understand the phenomena and whether optimization procedures can be performed to reduce data fluctuation, to better suppress authentic sensor data, and to achieve a desired level of AQI.



(**a**)



(**b**)

Figure 3.5: AQI fluctuation for Scenario B. (**a**) Before data pollution; (**b**) After data pollution.

After significant efforts in analyzing the phenomena, we arrive at the following explanation: the graphical representation of the AQI value on the PurpleAir Map is calculated using averaged PM2.5 measurement data in the past 10 min. Numerical presentation of the AQI value follows the same computation methodology but with a varied interval.

Therefore, an attacker may want to adjust the data update frequency of the fake sensor so that the AQI can be manipulated to the desired value. Here, we define the following notations: (1) the update interval of the visualized data on map as I_{VU}, where $I_{VU} = 10$ min $= 600$ s; (2) the real sensor and fake sensor data update intervals as I_r and I_f respectively, where $I_r = 80$ s and I_f is to be determined; (3) the real PM2.5 measurement (assuming constant during pollution) and

fabricated PM2.5 measurement as P_r and P_f respectively; (4) with respect to the targeted AQI, the corresponding range of PM2.5 pollutant measurement as $R_{\text{PM2.5}} = [LB, UB]$ (obtainable via the United States EPA AQI calculation table [76]). Here, only P_f and I_f are controllable variables, and all others are given. Therefore, we define three problems for pollution methodology optimization as follows and also present the sketchy solution methodologies for these three problems.

Problem I: Under what values of P_f and I_f can the visualized AQI be the same as the targeted AQI?

Solution. To ensure the displayed AQI value is the same as desired, a necessary and sufficient condition is to select proper P_f and I_f ($0 < I_f \leq I_{VU}$) such that the average of all received PM2.5 measurements during each update interval is located within the range $R_{\text{PM2.5}} = [LB, UB]$. This can be mathematically formulated as the following inequality.

$$LB \leq \frac{P_f \cdot \frac{I_{VU}}{I_f} + P_r \cdot \frac{I_{VU}}{I_r}}{\frac{I_{VU}}{I_f} + \frac{I_{VU}}{I_r}} \leq UB. \tag{3.2}$$

■

Since smaller values of I_f are equivalent to higher message transmitting frequencies, the adversary may want to know a minimum required number of messages, which corresponds to the maximum value of I_f, for a given P_f, to ensure the targeted AQI can be achieved. This can be formulated as Problem II.

Problem II: For a given P_f, what is the maximum applicable value of I_f?

Solution. This can be viewed as an optimization problem:

$$\text{Maximize } I_f$$

Subject to:

$$\begin{aligned}
&\frac{P_f \cdot \frac{I_{VU}}{I_f} + P_r \cdot \frac{I_{VU}}{I_r}}{\frac{I_{VU}}{I_f} + \frac{I_{VU}}{I_r}} \leq UB, \\
&\frac{P_f \cdot \frac{I_{VU}}{I_f} + P_r \cdot \frac{I_{VU}}{I_r}}{\frac{I_{VU}}{I_f} + \frac{I_{VU}}{I_r}} \geq LB, \\
&I_f > 0, \\
&I_f \leq I_{VU}, \\
&P_r \notin [LB, UB].
\end{aligned} \tag{3.3}$$

Therefore, we can derive the maximum value of I_f (denoted as I_{max}) for different cases as presented by Equation (3.4):

$$I_{max} = \begin{cases} \min\left\{\frac{UB - P_f}{P_r - UB} I_r, I_{VU}\right\}, & \text{if } P_r > UB, \\ \min\left\{\frac{P_f - LB}{LB - P_r} I_r, I_{VU}\right\}, & \text{if } P_r < LB. \end{cases} \tag{3.4}$$

■

Deductions here are assuming a suitable value of P_f. Certainly, requirements exist when selecting value of P_f to make data pollution action practical. In this work, when $P_r > UB$, we opt to choose $P_f \in [LB, UB)$; and when $P_r < LB$, we opt to choose $P_f \in (LB, UB]$. This conservative yet clear choice satisfies all requirements of P_f, and more importantly, it additionally grants that we can send fabricated messages as frequently as we want (i.e. I_f can be selected for as close to 0 as we want, so long as it is less than I_{max}, targeted AQI still can be achieved). Such freedom enables the choice of any adequately desired higher fake data frequency, which can thus help to alleviate fake data receiving instability and better suppress real data and its variations, so that a better pollution result with much less fluctuation can be achieved. The case when $LB \leq P_r \leq UB$ is trivial since no

fabricated message is needed for pollution in this situation.

Furthermore, we empirically consider the possibility of deciding a uniform I_{max} for parallel multiple sensor data pollution, in which identical targeted AQI and a proper P_f are used for polluting all victim sensors in parallel. This is expressed in Problem III.

Problem III: What is the uniform I_{max} when multiple sensors are being polluted simultaneously?

Solution. Denote this uniform maximum fake sensor data update interval as I_{uni}. We assume the number of sensors being polluted simultaneously is n. The corresponding real PM2.5 measurements for these sensors are P_{r_1}, P_{r_2}, ..., P_{r_n}. Without loss of generality, we assume these n values are all in an increasing order, and among these n values, the first N_1 values P_{r_1}, ..., $P_{r_{N_1}}$ are less than LB; values $P_{r_{N_1+1}}$, ..., $P_{r_{N_2}}$ are in between LB and UB; and remaining values $P_{r_{N_2+1}}$, ..., P_{r_n} are greater than UB, where $1 \leq N_1 \leq N_2 \leq n$. This assumption is general and can always be achieved by reordering all sensors.

Since sensors in position N_1+1 through N_2 do not have any requirement on I_{uni}, we exclude them from the later portion of the solution. For the rest of the sensors, we denote corresponding I_{max} for sensor i as I_{max_i}, $S_1 = \{1, \cdots, N_1\}$, and $S_2 = \{N_2+1, \cdots, n\}$. I_{uni} can be expressed as:

$$\begin{aligned} I_{uni} &= \min\{I_{max_i} \mid i \in [1,n]\}, \\ &= \min\{I_{max_i} \mid i \in S_1 \cup S_2\}, \\ &= \min\{I_{uni-S_1}, I_{uni-S_2}\}, \end{aligned} \tag{3.5}$$

where $I_{uni-S_1} = \min\{I_{max_i} \mid i \in S_1\}$ and $I_{uni-S_2} = \min\{I_{max_i} \mid i \in S_2\}$. By Equation (3.4),

$$\begin{aligned} I_{uni-S_1} &= \min\{I_{max_i} \mid i \in S_1\}, \\ &= \min\left\{\min\left\{\frac{P_f - LB}{LB - P_{r_i}} I_r, I_{VU}\right\} \mid i \in S_1\right\}, \\ &= \min\left\{\frac{P_f - LB}{LB - \min\{P_{r_i} | i \in S_1\}} I_r, I_{VU}\right\} = \min\left\{\frac{P_f - LB}{LB - P_{r_1}} I_r, I_{VU}\right\}. \end{aligned} \tag{3.6}$$

Similarly,

$$\begin{aligned} I_{uni-S_2} &= \min\{I_{max_i} \mid i \in S_2\}, \\ &= \min\left\{\min\left\{\frac{UB - P_f}{P_{r_i} - UB} I_r, I_{VU}\right\} \mid i \in S_2\right\}, \\ &= \min\left\{\frac{UB - P_f}{\max\{P_{r_i} | i \in S_2\} - UB} I_r, I_{VU}\right\} = \min\left\{\frac{UB - P_f}{P_{r_n} - UB} I_r, I_{VU}\right\}. \end{aligned} \tag{3.7}$$

Thus,

$$I_{uni} = \min\{I_{uni-S_1}, I_{uni-S_2}\} = \min\left\{\frac{P_f - LB}{LB - P_{r_1}} I_r, \frac{UB - P_f}{P_{r_n} - UB} I_r, I_{VU}\right\} \tag{3.8}$$

That is, when launching multiple sensor pollution attacks simultaneously, a suitable uniform I_{max} exists, and is only possibly affected by the smallest and largest real PM2.5 readings among all sensors. ■

Scenario C: Enumerating MAC Addresses

The challenge of the previously stated data pollution approach in Section 3 is to obtain victim sensors' MAC addresses, which requires physical access to the geographical locations of sensors. To overcome this issue, we develop an automatic, efficient, and low-cost large-scale MAC address detection strategy. We could technically obtain MAC addresses for every PurpleAir sensor by using this strategy.

As discussed previously, responses from PurpleAir servers automatically differentiate correct and incorrect sensor MAC addresses (Table 3.4). Such responses can be exploited to screen for real MAC addresses of PurpleAir sensors. A thorough search of the entire MAC address space is theoretically possible, but practically not feasible given the enormous amount of possible MAC addresses (in the order of 16^{12}). However, PurpleAir sensors use the ESP8266 chip and there are 24 MAC address prefixes assigned to its manufacturer Espressif Systems as listed in Table 3.5. Therefore, the entire search space for sensor MAC addresses is reduced to around $24 \times 16^6 \approx 0.4$ billion.

With such a dramatically reduced search space and with possible parallel deployment of the scanning program, a large-scale automatic sensor MAC addresses screening is feasible, which could consequently enable a large-scale sensor data pollution of the entire PurpleAir system. Technically, we could first enumerate all possible MAC addresses of all PurpleAir sensors, and verify the authenticity of each MAC address individually by sending messages containing each individual MAC address to the www.PurpleAir server. If a correct MAC address is sent, the server will respond with geographic coordinates of the corresponding sensor. Using this approach, we could populate a list of every PurpleAir sensor and perform a large-scale data pollution attack on the PurpleAir system.

Algorithm 1 presents our sensor MAC address scanning algorithm. More experiment details are included in the next section.

Algorithm 1: MAC scanning algorithm

```
for p in PrefixSet do
    for i in range(16^6) do
        mac=genHexMAC(p,i);
        mac_str=macToString(mac);
        url=genURL(mac_str);
        request=sendHTTPRequestTo(url,
          "www.purpleair.com");
        response=getHTTPResponseBody(request);
        if response == "NOT FOUND" then
            mac is not a sensor MAC address;
        end
        else
            mac is a sensor MAC address;
            writeToFile(mac);
        end
    end
end
```

Results of the Three Network-based Attacks

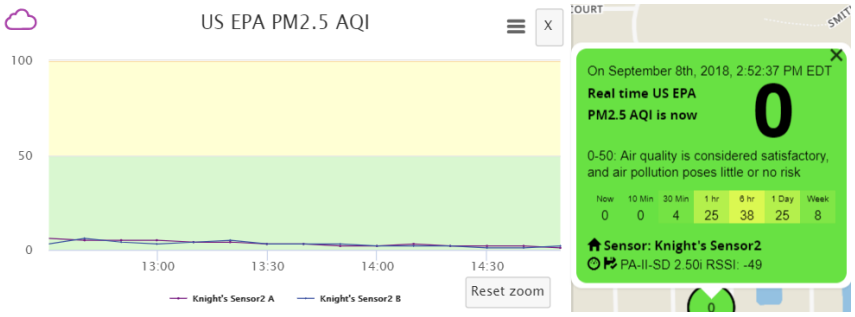
Now we present the experiment results. These experiments show the effectiveness of our data pollution approaches, including the data pollution attacks for each of the first two scenarios, the effectiveness of the wardriving attack, and a feasibility analysis of the large-scale sensor scanning attack.

Effectiveness of Pollution Attack in Scenarios A & B

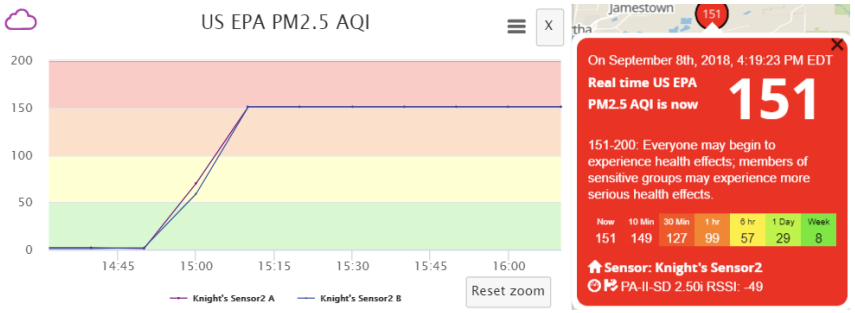
Either Scenario A or Scenario B focuses on the data pollution of a single target sensor. In this section, we present results for both scenarios.

Scenario A. In scenario A, we utilize mitmproxy to conduct the MITM attack between an PurpleAir sensor and its servers. When launching the attack, all messages from the target sensor are intercepted and modified if necessary, then are redirected to their original destinations. That is, in this experiment, we only modify data sent from a real sensor but without changing its communication pattern or frequency.

As an example, the data pollution effects for increasing PM2.5 reading and the respected AQI change can be seen in Figure 3.6. During the experiment, $AQI_{PM2.5}$ of the real sensor remains at around 0, while our intended value of $AQI_{PM2.5}$, modified by altering the PM2.5 measurement contained in the uploaded sensor data, is 151. It can be verified that the effectiveness of data pollution is apparent, as reflected in the dramatic change of AQI. We can also verify that the data pollution result is stable and experiences no fluctuation due to the complete blocking of original sensor data. Once the polluted data can be manipulated in a stable manner, we can freely control the data fluctuations so that the shape of the curve could appear more natural and less suspicious, to further camouflage the polluted data. In this way, both the sensor owner and the PurpleAir system have extreme difficulty in noticing the data pollution behavior and suffer from permanent loss of original sensor data.



(**a**)



(**b**)

Figure 3.6: AQI fluctuation for Scenario A. (**a**) Before data pollution; (**b**) After data pollution.

Scenario B. During the data pollution experiment for Scenario B, $\text{AQI}_{\text{PM2.5}}$ of the real sensor remains at around 0, whereas our modified value of $\text{AQI}_{\text{PM2.5}}$, attained by conducting the spoofing attack, is 151.

As described in Section 3, the PurpleAir servers receive two sets of data simultaneously during the spoofing attack: one authentic set of data from the target sensor, and the other fabricated data set from the fake sensor. In this case, P_r is much smaller than LB, hence we choose $P_f \in (LB, UB]$, and opt to set $P_f = UB$. We adjust the transmission interval of the fake sensor to be slightly smaller than I_{max} (computed by Equation (3.4)), but significantly smaller than I_r, and find that it is sufficient to suppress the authentic data and conceal the spoofing attack. The effects of this

optimized polluting strategy can be observed in Figure 3.7. With the optimized strategy, data fluctuations are almost entirely eliminated comparing to Figure 3.5 and the displayed AQI result indeed reaches the intended value stably. In short, we are able to successfully attain an effective pollution methodology while the servers receive both authentic and fabricated sensor data.

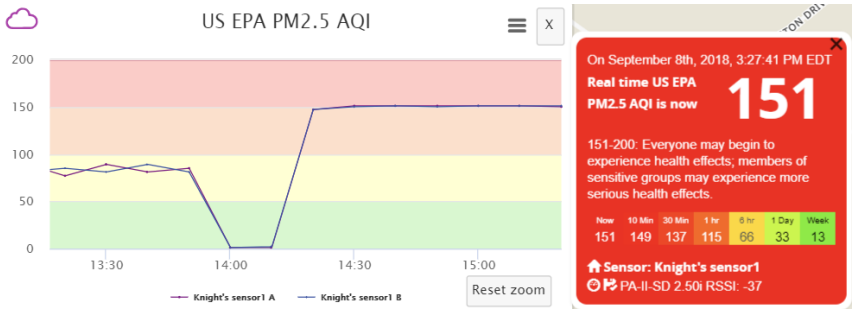
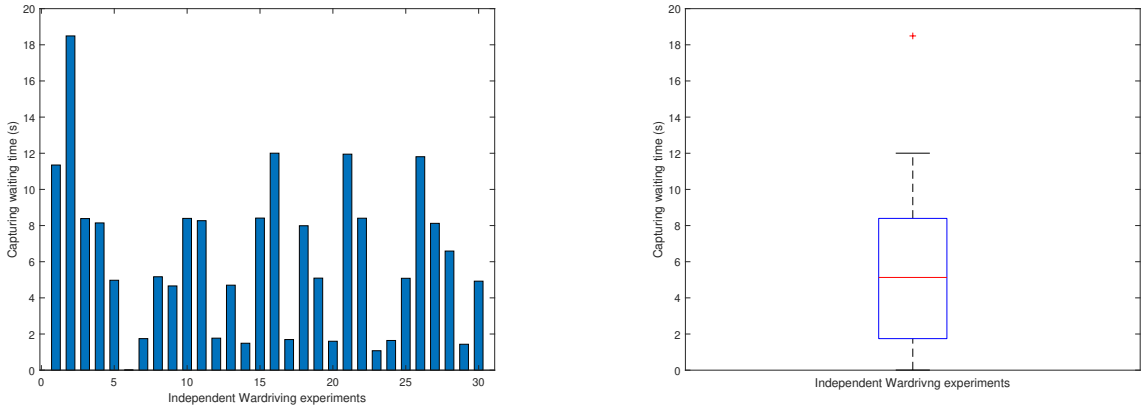


Figure 3.7: AQI fluctuation for Scenario B with optimized data pollution strategy.



(**a**) (**b**)

Figure 3.8: MAC capturing waiting time in wardriving experiments. (**a**) Time for each experiment; (**b**) Time distribution box plot.

Effectiveness of Wardriving Attack

As an auxiliary component of the data pollution approach for scenario B, the described wardriving attack is an plausible way to obtain the MAC address of a sensor by knowing only its approximate geographical location, with no requirement of other information or direct physical sensor possession.

To evaluate the effectiveness of the wardriving attack, we design the following setup: (1) All experiments are carried out by a laptop equipped with the Kismet network sniffing tool and a wireless network card. The laptop is set inside a vehicle fulfilling the wardriving technique. (2) Before the experiment, we conduct network traffic sniffing in the surrounding of the volunteer's household several times beforehand with the activated Kismet tool for the purpose of locating an area where the targeted WiFi network can be received persistently during the experiment period. Then we start the actual experiments. (3) In each experiment, we keep the wardriving vehicle consistently within the previously identified area, activate the Kismet tool, and record capturing waiting time until the targeting sensor MAC is observed. The resulting waiting time is then recorded and the experiment starts again at another arbitrary time. (4) This independent experiment is repeated 30 times for revealing its intrinsic randomness.

The consequent experiment results, including waiting time for each experiment round and a box plot of waiting time distribution, are shown in Figure 3.8. It can be observed that randomness impacted the actual MAC capturing waiting time. Nonetheless, the resulting waiting time distribution remains acceptable for the wardriving attack with an average waiting time being 6.18 s, the minimum at approximately 0.006 s, and the maximum at around 18.49 s. In fact, from in-depth observation of the sensor operating mechanism, we believe the waiting time results from a combination of two factors: First, sensor messages are being sent in a predefined periodical pattern with internal temporal separations between messages; second, wardriving sniffing is only capable of

capturing sensor MAC addresses when its communication occurs. Thus, it is reasonable to expect that the wardriving capturing waiting time experiences inevitable yet tolerable fluctuation.

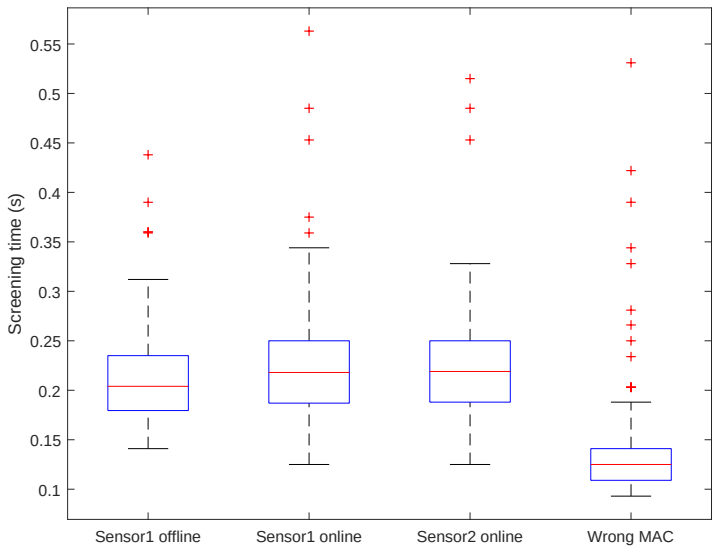


Figure 3.9: Single MAC screening time box plot figure.

Feasibility of Sensor Scanning Attack in Scenario C

We have proposed an approach for automatic large-scale sensor MAC address detection for the PurpleAir system. Even though we are able to drastically reduce the search space by leveraging prefix patterns of sensor MAC address, the efficiency in verifying one single address is still crucial in determining the overall efficiency and feasibility of the sensor scanning attack.

In Figure 3.9, we present results of an empirical experiment in estimating the time cost of scanning a single MAC address. In this experiment, we utilize real sensor MAC addresses from two PurpleAir sensors in our possession (named sensor1 and sensor2), and one invalid MAC address inside the search space altered from the MAC address of sensor1. We perform the tests under

four scenarios: (1) using sensor1's MAC when sensor1 is offline; (2) using sensor1's MAC when sensor1 is online; (3) using sensor2's MAC when sensor2 is online; and (4) using the invalid MAC address. For each scenario, we assemble the sensor message accordingly, send the message to the www.PurpleAir server, and record the time it takes for the server to respond. We repeat this test 100 times for each scenario, and plot the distributions of server response time (Figure 3.9). It is clear that the average response time is either around or less than 0.2 s for all four scenarios, hence we choose to use the average response time of 0.2 s for later analysis.

Given its nature, the process of MAC address scanning can be easily paralleled since no data exchange nor synchronization is needed among individual parallel tasks. Here we refer to the computing unit for each parallel task as a "worker", who is responsible for screening one MAC address at any moment. A worker can be a physical computing equipment, or virtual computing resource. With different number of available workers and assuming an average response time of 0.2 s, the time costs of scanning the entire MAC address space for PurpleAir sensors can be estimated as in Table 3.6.

Table 3.6 reveals that with 1000 workers, the entire brute-force scanning process could be accomplished in less than one day, thus proving the feasibility of our designed large-scale sensor MAC scanning attack. Using the popular cloud computing platform Amazon EC2 [5] as an instance, a "m5.12xlarge" instance with 48 "vCPU" , 192 GB of memory, and enhanced network connection capacity would be sufficient to provide simultaneous computing capability for at least 48 workers. Hence, the scanning process could be completed by employing only 21 "m5.12xlarge" instances for less than one day, at a cost of approximately one thousand dollars (assuming $2.304 per instance per hour according to Amazon on-demand instance price [6]). Such a large-scale attack can also be deployed over PlanetLab [59] for free.

Furthermore, although the brute-force scanning process requires no prior knowledge of the real

Table 3.6: Theoretical efficiency of MAC address scanning with different number of parallel computing workers.

NO. of Workers	5	100	300	500	800	1000
MAC Scanned Per Hour	9×10^4	1.8×10^6	5.4×10^6	9×10^6	1.44×10^7	1.8×10^7
Approximated Total Hours	4474	224	75	45	28	23

MAC address distribution among the entire search space, a screening process could, in practice, be accelerated by utilizing MAC address distribution patterns. One may conduct a sampling screening process in a selected reduced search space, with the purpose of estimating real MAC address distributions. If any distribution pattern other than a uniform one is identified, particularly those with apparent clustering features, it is possible to apply a heuristic search algorithm to accelerate the scanning process, providing that the goal of scanning is to detect a sufficiently large number of new valid sensor MAC addresses rather than a complete list of all sensor MAC addresses. In this manner, the overall scanning time could be further decreased.

Discussion

In this section, we discuss how to defend against the three attacks introduced in Section 3.

We first look at the defense to the MAC address enumeration attack. Recall when a sensor sends data to the server, the message contains the MAC address for sensor identification. However, since the MAC address is predictable, it is inappropriate for fulfilling such a purpose. Instead, a long random number can be used to identify the sensor, denoted as device ID. Even if the adversary knows one device ID, he/she cannot predict other device IDs given that the space of device IDs is too huge for enumeration.

We now look at the defense to the wardriving attack. If a random device ID is used, the wardriving

attack becomes ineffective if the network is encrypted. For example, WPA2 should be used to protect WiFi, and HTTPS should be used for end-to-end encryption. Therefore, the adversary cannot obtain the decrypted content of the communication for the device ID.

A malicious user possessing an PurpleAir sensor is dangerous. He/she may register the sensor following PurpleAir's procedure with a fake geographical location. The malicious user can also perform the MITM attack to manipulate the sensor data. Registration with a fake location could be avoided by an internal GPS module or via WiFi localization. To defeat the MITM attack, the certificate based mutual authentication can be used. That is, the sensor authenticates the servers and the servers authenticate the sensor as well. A private key has to be securely stored in the sensor. The malicious user should not be able to change the firmware of the sensor either.

In summary, we believe that a sensor should adopt the following strategies for sensor data security and integrity. Secure boot should be used to prevent the manipulation of the firmware of the sensor. With secure boot, if the firmware is changed, the sensor will not boot. Such a firmware is trustworthy. Flash encryption should be used to protect sensitive data on the flash, including the WiFi credentials. Certificate based mutual authentication with TLS should be used to defeat the MITM attack and protect the communication. With mutual authentication, the server can identify the sensor and a random device ID may not be necessary since the server only accepts data from authenticated sensors. The mutual authentication renders the MITM attack invalid and the hash of the sensor's public key can be adopted as the device ID if needed. Secure storage should be used to store the sensor's private key so that the adversary cannot obtain the private key for the MITM attack. Sensors should have different private keys so that even if one sensor is compromised, it will not affect others. The location of the sensor can be obtained from either a GPS module on the sensor or WiFi localization. A GPS module can be problematic since a dedicated adversary may replace the module with an artificial one. The GPS may not work inside buildings. The WiFi localization may be more appropriate since the trusted firmware will retrieve the WiFi information

for the purpose of localization. The server may also validate the reported location from the device via the IP location service [14], which finds the geolocation of a sensor from the IP address of the sensor while the accuracy of the IP location service is limited [16]. Secure firmware upgrade is needed in case that vulnerabilities are found in the system. The advance of hardware now actually makes the defense strategies introduced above possible for sensors using low-cost microcontrollers (MCUs) [44, 25, 49, 74].

Summary

In this paper, we perform a systematic analysis of security and data integrity of a popular air quality monitoring system, PurpleAir, which uses low-cost air quality sensors to gather and remotely manage pollutant concentration data. By analyzing the traffic of sensors, we are able to understand the architecture of the PurpleAir system and its communication protocol. We then present approaches of polluting sensor data in three scenarios: sensor in physical possession, knowing sensor MAC addresses (or geographical location), and automatic large-scale system pollution. By designing several attack methods including man-in-the-middle attack, spoofing attack, wardriving attack, and device scanning attack, we were able to successfully pollute sensor data with fabricated data. We also demonstrated that we have the capability of polluting data from any sensors of interest that are deployed globally without being detected by the owners of sensors. Guidelines on security improvement and defense mechanisms to mitigate such system vulnerabilities are also provided and discussed.

CHAPTER 4: RUNTIME SOFTWARE SECURITY OF TRUSTZONE-M ENABLED MCU-BASED IOT DEVICES

Even if software integrity can be verified at boot time via mechanisms like secure boot, protecting software of embedded devices at runtime is challenging due to the heterogeneity and constrained computational resources of MCUs.

TrustZone-M, the TrustZone extension for ARMv8-M architecture, is an emerging solution to the runtime software security of IoT devices [10, 43, 33]. Specifically, it provides resource-constrained MCUs a lightweight hardware-based solution to a trusted execution environment (TEE) for security-related software, i.e., Secure World (SW), which is isolated from the rich execution environment, i.e., Non-secure World (NSW). The NSW software cannot access the SW resources directly. TrustZone-M provides a Non-secure Callable (NSC) memory region in the SW so that functions can be defined in the NSC region as the gateway from the NSW to the SW. To the best of our knowledge, TrustZone-M has not been adopted in commercial IoT products.

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

A number of works have been done concerning the security issues in TrustZone. Cerdeira et al. [17] present systematization of knowledge (SoK) on the Cortex-A TrustZone security while our

work focuses on the Cortex-M TrustZone. Iannillo et al. [31] propose a framework for the security analysis of TrustZone-M. However, their work does not identify concrete vulnerabilities/attacks against TrustZone-M. Jung et al. [33] design a secure platform based on the Platform Security Architecture (PSA) with a brief discussion of possible attacks. Our work demonstrates five types of realistic attacks, breaching the security of TrustZone-M.

TrustZone-M

TrustZone-M. TrustZone for ARM Cortex-A processors (TrustZone-A) is a security technology that isolates security-critical resources (e.g., secure memory and related peripherals) from the rich OS and applications. An ARM system on a chip with the TrustZone extension is split into two execution environments referred to as the **Secure World** (SW) and the **Non-secure World** (NSW). Software in the SW has a higher privilege and can access resources in both the SW and the NSW, while the Non-secure software is restricted to the Non-secure resources. The NSW may communicate with the SW using the monitor mode of TrustZone-A. Recently, the TrustZone technology has been extended to the ARMv8-M architecture as TrustZone-M for some Cortex-M series processors, which are specifically optimized for resource-constrained MCUs. TrustZone-M has the SW and NSW, but differs from TrustZone-A in terms of implementation. One prominent difference is that TrustZone-M introduces a special memory region in the SW named Non-secure Callable (NSC) region to provide services from the SW to Non-secure software. Transition between the two worlds through the NSC region is achieved by NSC function calls and returns.

Runtime software security in IoT devices. IoT devices are usually capable of connecting to remote servers or controllers and transferring messages to them via communication venues such as WiFi, Bluetooth, and low-power wide-area network. MCU based IoT devices are often programmed with languages such as C and C++ because they are compact, highly efficient and have

the ability of direct memory control [68]. Such languages provide programmers a flexible platform to interact with the low-level hardware. On the flip side, they are notoriously error-prone and daunted by security issues. Attackers may perform runtime software attacks against vulnerable IoT devices with such features. Runtime software attacks can hijack the program control flow by altering the control data (e.g., return address and function pointer) or change program memory by manipulating non-control data [52]. Often in such an attack, an adversary corrupts the vulnerable memory by inputting a carefully crafted malicious payload, which eventually results in abnormal program behaviors.

Potential Security Issues and Pitfalls in TrustZone-M Enabled Devices

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.

Threat Model

We consider a victim IoT device using the TrustZone-M enabled MCU. It is assumed that security-related coding mistakes exist in the software of the victim device which is able to receive inputs from the Internet or peripherals. Though the SW of TrustZone-M provides a TEE that the NSW software cannot directly access, the TEE can only work normally under the assumption that Secure software is well crafted with no security-related coding mistakes. However, coding mistakes may exist in TrustZone-M's NSW, the NSC region, and the SWX region. Memory layout of a TrustZone-M based MCU, SAM L11, is shown in Figure 4.1. An adversary can exploit the coding

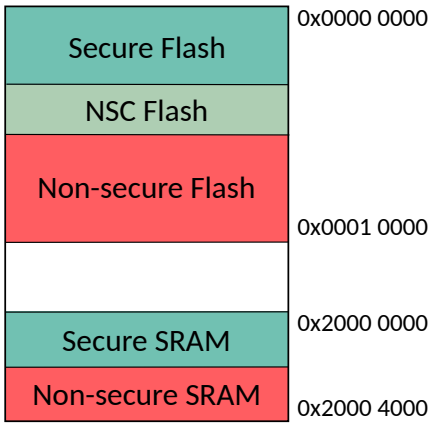


Figure 4.1: Memory layout of SAM L11. The memory is divided into the SW and NSW at the hardware level. Code in the SW (Secure Flash, NSC Flash, and Secure SRAM) can access the whole chip, while code in the NSW (Non-secure Flash and Non-secure SRAM) can only directly access resources inside the NSW.

Table 4.1: 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.

mistakes and send a malicious input (i.e. payload) to deploy software attacks. Even if the SW does not accept inputs from the Internet or peripherals and only the NSW communicates with the outside world, an attacker may compromise the NSW and feed malicious inputs into vulnerable NSC functions, which can access Secure resources. Therefore, if a NSC function is vulnerable, the entire SW may be compromised.

Runtime Software Attacks

Table 4.1 lists software attacks we have identified against the NSW, NSC and SWX of TrustZone-M. It can be observed that traditional software attacks found in other platforms such as computers and smart phones can be conducted in all regions of TrustZone-M, including code injection, return-oriented programming (ROP), heap-based buffer overflow (BOF), and format string attacks, if requisite software flaws present. We also discover potential exploits specifically targeting the NSC. Here, all attacks against the NSC refer to those deployed from the NSW. We present the details and challenges of these attacks in the context of TrustZone-M below.

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 4.1 presents an example, in which *buf[256]* will overflow if the input array is longer than 256 bytes. As a result, the extra data will overwrite the adjoining stack contents including the return address, at which the control flow will continue after the subroutine return. Adversaries may perform stack-based BOF attack for malicious code injection. The control flow can be redirected to the malicious code sent along with the payload by overflowing the local buffer and overwriting the original return address with the entry address of the malicious code.

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

Listing 4.1: Example of a function with BOF vulnerability

To specifically implement a stack-based BOF attack against the ARMv8-M architecture, we first

investigate its stack structure. A stack frame for a function in ARMv8-M consists of local variables, variable registers (R4–R7), and return address, as illustrated in Figure 4.2. 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 [39]. 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 *NOP* (no-operation) instructions, called a NOP sled, before the injected malicious code in the payload, since any hit of a NOP instruction will lead to the execution of malicious code eventually.

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

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

The second scenario refers to the null bytes in the entry address of the malicious code. In SAM L11, the malicious code has to be injected onto the stack, which is on the SRAM with a fixed range of address from 0x20000000 to 0x20004000, within which the higher halfword of any addresses is

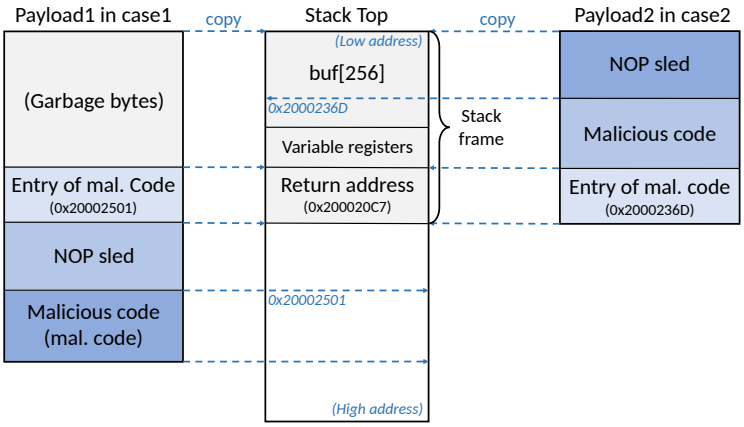


Figure 4.2: Stack-based buffer overflow attack for code injection

0x2000, containing a null byte all the time. Taking Payload1 in Figure 4.2 as an instance, since the NOP sled and malicious code are positioned after the entry address, the copy process of Payload1 will terminate when the null byte in the entry address is hit. Copying either the NOP sled or the malicious code to the stack would fail in this case. A potential solution is to construct the payload like Payload 2 in Figure 4.2, where the entry of malicious code is placed at the bottom. Because the little-endian ordering in ARMv8-M, the 0x2000 is located at the last two bytes of Payload 2 and shall be the only two bytes missing when copied to the stack. The original return address already contains 0x2000 in its upper halfword if the caller function is executed from the SRAM, in which case the BOF will still be applicable.

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

Return-oriented Programming Attack

BOF based code injection can be mitigated by security mechanisms like non-executable memory [50], which prevents code execution from certain memory region. However, an attacker can bypass such defense by leveraging code reuse attack (CRA). A representative CRA is ROP attack. Utilizing BOF to overwrite the return address, ROP redirects the control flow to a target code sequence (called a gadget) found in the existing software code. It is also possible to chain several gadgets for more complex program control. Each gadget in the chain is a code segment responsible for certain operations (e.g., arithmetic operations and load/store data) and must end with the epilogue of a subroutine for the sake of chaining the gadgets. In ARMv8-M, the instruction sequence *{POP LR, BX LR}*, which is the epilogue of leaf subroutines, pops a word to link register (*LR*), and then branches to the address specified by *LR*. Instruction *POP PC*, which is the epilogue of non-leaf subroutines, directly pops a word to program counter (*PC*).

Now we explain how to chain the gadgets utilizing the subroutine epilogue in each of them. An adversary needs to craft a "gadget stack" and sends it along with the payload. Each gadget in the chain except the last one, has a corresponding gadget frame placed on the gadget stack. A gadget frame consists of several words of data that will be popped to the operand registers of the last *POP* instruction in that gadget. Data provided by the gadget frame include the address of the next gadget, which helps to jump to the next gadget after being popped. An example of a chain of three gadgets in ARMv8-M is presented in Figure 4.3. The payload contains the entry of Gadget 1 and two gadget frames corresponding to Gadget 1 and 2. To ensure the entry of Gadget 2 will be popped to *LR*, the gadget frame for Gadget 1 contains two more words before the entry word since the second to last instruction in Gadget 1 pops the third word from the stack frame to *LR*. Two words before the entry of Gadget 2 are provided such that they will be popped to *R4* and *R5* instead. Similarly, the gadget frame for Gadget 2 provides two words of data which will be popped

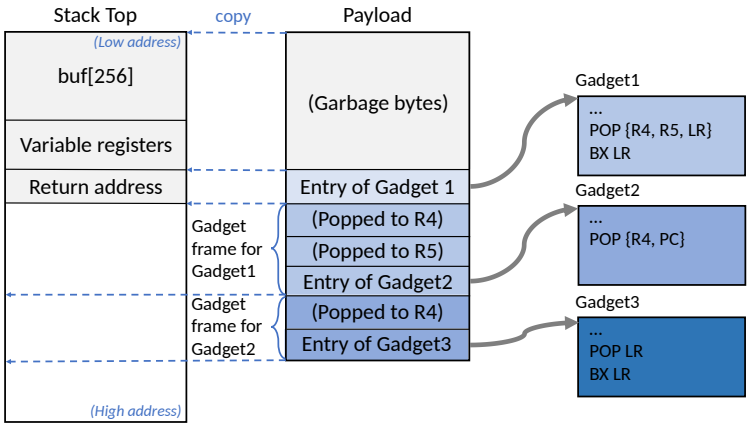


Figure 4.3: Return-oriented programming attack

to *R4* and *PC* so that execution of Gadget 3 can be routed to start.

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 [38]. 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 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 [80].

Format String Attack

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

In SAM L11, an adversary is able to exploit format string vulnerabilities for memory crash and reading/writing some values at a specific stack location by sending a malicious string input containing more format specifiers than expected. For example, by sending the string *"%x %x %x"* to the vulnerable function illustrated in Listing 4.2, in which no arguments are provided to the three specifiers in the input format string, three bytes of data following the return address on the stack will be printed in hexadecimal. However, reading or writing at an arbitrary memory location specified by an address is unachievable in SAM L11 due to the particular memory addressing as shown in Figure 4.1. 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.

```
1 void fmt_str(char *input){
2     printf(input);
3     ...
```

Listing 4.2: Example of a vulnerable format string function

Attacks against NSC Functions

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

As the gateway to the SW, the implementation of the NSC software should be particularly cautious. According to the guidance from ARM [8], hardware, toolchain, and software developers share a common responsibility to implement the NSC software securely. Though some requirements are offered in the guidelines, since the hardware and toolchain vary from vendors to vendors, there is no off-the-shelf solution to implementing trusted NSC software.

We identify two pitfalls that software developers may meet while programming the NSC functions. The first pitfall is caused by the data arguments sent from the NSW. The toolchain of SAM L11 only helps to generate the Secure gateway veneer for NSC functions but leaves the function programming to the developers. Security-related coding mistakes may present in the NSC functions as well and can be exploited by crafting Non-secure data inputs. Software exploits in the NSC

region would lead to a compromised SW. This is because the NSC region belongs to the SW and a compromised NSC program under the control of an adversary can access any resources inside the SW.

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

Listing 4.3: Example of a vulnerable NSC function

The second pitfall comes from the untrusted pointer inputs. When Non-secure software passes pointer arguments to the SW through NSC functions, NSC functions should ensure that these pointers point to the Non-secure memory. Otherwise, NSC and Secure functions may assist the Non-secure software to read or write the Secure memory. The vulnerable NSC function illustrated in Listing 4.3 can leak and corrupt the Secure memory contents at arbitrary Secure addresses if the first and the third arguments are Secure addresses and the second argument is set to 1. The violation of the principle that "Secure resources are not allowed to be accessed by the NSW" severely harms the fundamental security of the TrustZone-M implementation.

Evaluation

Table 4.2: 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.

In this section, we evaluate the five software attacks described in Section 4. We are able to successfully perform these attacks against a TrustZone-M enabled MCU, SAM L11.

Experiment Setup

We use a laptop as the attacker to continually send inputs to a SAM L11 Xplained Pro Evaluation Kit as the victim device. The laptop is connected with SAM L11 through a USB-to-UART adapter while an attacker may also inject malicious strings into an Internet connection of a SAM L11 based IoT device. In SAM L11, two UARTs are configured accordingly as a Non-secure peripheral and a Secure peripheral to receive inputs sent from the laptop to the NSW and SW respectively. For the first four attacks, we construct specific vulnerable functions in both Non-secure and Secure applications of SAM L11 and malicious payloads will be sent through the UARTs to trigger the attacks. The sizes of the payloads and experiment results are given in Table 4.2.

Experiment Results

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

As a proof-of-concept implementation of ROP, we craft a chain of gadgets with three exploitable gadgets by splitting the assembly code of a program, which prints the memory content at a given address, into three code segments. A subroutine epilogue (i.e., *POP PC*) is appended at the end of

each code segment. These gadgets are pre-stored at different locations of the flash in advance. We craft a gadget stack to chain these gadgets and send it along with the payload to SAM L11. As a result, the intended constant string is successfully printed on the adversary's terminal. A way to evaluate the feasibility of ROP against a certain program is to count up the occurrences of potential gadgets in the program. In fact, this process is equivalent to counting up the number of "*POP PC*" and "*BX LR*" instructions according to the definition of potential gadgets introduced in Section 4. We take a basic Non-secure application image, which only initializes necessary peripherals, as an example and search all the subroutine epilogues in it. The Capstone disassembly engine [62] is used to dissemble and search in the binary code. The size of the example image is 4.14KB with 1908 instructions in total. As a result, 49 "*POP PC*" and 16 "*BX LR*" are found in the image binary, representing 3.41% of the whole image.

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

As we stated in Section 4, an adversary can exploit the vulnerable format string function in SAM L11 to read out the stack contents. The payload used is "%08x %08x %08x %08x %08x" and we eventually read five sequential bytes from the stack via UARTs.

To verify the feasibility of NSC-specific attacks, we look into the example software projects provided by the vendor of SAM L11, five of which contain NSC software implementations. We statically analyze the source code of these five NSC implementations and find three to be vulnerable. These three implementations share two vulnerable NSC functions as in Listing 4.4, where two of them contain the first function and the other contains the second function. The first vulnerable

function is subject to the format string attack when it is called by the Non-secure software and the argument is a crafted format string input that can be controlled by an adversary. In our experiment, we send "%08x %08x %08x %08x %08x" as the payload and five sequential bytes from the Secure stack are eventually printed in the adversary's terminal. The second function has an information leakage problem. We call this function in the NSW with an argument which is a Secure address, as a result, the Secure memory content at the target location is then printed.

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

Listing 4.4: Vulnerable NSC functions in SAM L11 demo code

Discussion

With the increasing concerns of IoT security, more and more manufacturers start to make their MCU products support various security features such as secure boot and secure storage [58]. Runtime software attacks targeting IoT devices can be mitigated in a way with few costs by properly enabling some of the security features provided by MCUs. Taking SAM L11 as an example, the code injection attack can be neutralized via setting the RXN (RAM eXecute never) and DXN (Data eXecute never) fuse bits, via which code execution from data memory would not be allowed. In addition, SAM L11 supports secure debugging port with key authentication and allows to disable the write operation on code memory. These two security mechanisms ensure the code integrity at runtime so that adversaries cannot easily hijack the control flows by rewriting the firmware. However, other aforementioned runtime software attacks (i.e., ROP, heap-based BOF, format string exploits,

and NSC-specific attacks) are difficult to resist with the equipped security features.

The control flow integrity (CFI) is a technique for preventing runtime control-oriented attacks such as ROP. By monitoring the control flow of a program at runtime, it can detect unexpected control flow changes. [55] provides an implementation of CFI for TrustZone-M to protect the NSW. In it, a control flow graph (CFG) of the Non-secure program is constructed by static or dynamic analysis of its code and is saved in a non-writable region of the NSW. Code instrumentation is performed so that the program jumps to a *branch monitor* before any control flow changes in the original code. The branch monitor refers to the CFG and monitors control flow changes at runtime. Before a function call, the correct return address is pushed on a *shadow stack* in the SW. Since a function might be called by different callers and return to different places at runtime, the CFG cannot tell the exact return address at runtime. The shadow stack in the SW is used to record the correct return address for a certain function call. Here the stored return addresses must be fully protected from being altered. The SW, which can be seen as a trust anchor for the NSW, provides the required secure storage, namely shadow stack, for the correct return addresses and a trusted execution environment for any operations on the shadow stack.

The CFI for protecting the control flow of the NSW is not sufficient for the overall system security. It can be observed from Table 4.1 that all software attacks may occur in both the SW (including NSC and SWX) and NSW. Recall that the CFI mechanism requires a trusted execution environment and a secure storage. In the case of TrustZone-M, the SW is supposed to play the role of such a trust anchor. If the SW itself is insecure and vulnerable to potential software attacks at runtime, it cannot provides the indispensable secure storage and trusted execution environment required by CFI for the NSW. Thus the effectiveness of the CFI enforcement for the NSW would be harmed. Another issue of CFI is that it may not defeat the heap-based BOF or format string attacks if control flows are unchanged but sensitive data are modified.

For the overall security of TrustZone-M based IoT devices, the following strategies may be adopted. First, the SW shall be very cautious about authenticity and security of the Internet connection in order to avoid remote exploits of SW software vulnerabilities. Second, the arguments sent from the NSW to an NSC function may be an address or application data. If it is an address, the NSC function must verify that it is not a Secure address before passing the address argument to the SW, since an NSW program shall not access the SW resources directly. If it is application data, input validation and sanitization shall be carefully performed. Third, security mechanisms including CFI and onboard executable space protection of TrustZone-M shall be applied to the NSW for control flow integrity. Finally, secure coding, code review and penetration testing are critical to the overall system security and the best practice shall be adopted [12]. Runtime software attacks may also be mitigated by programming the MCUs with security oriented languages such as Java [32, 71] and Rust [64].

Summary

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

CHAPTER 5: A SECURITY FRAMEWORK FOR TRUSTZONE-M ENABLED IOT DEVICES

IoT devices are suffering a broad attack surface including attacks from five dimensions, namely hardware, boot-time software, runtime software, network, and over-the-air (OTA) update. In work we propose a security framework for TrustZone-M enabled IoT devices to secure the devices from the five dimensions. The security framework utilizes standard hardware or software protection provided by commercial MCUs. Besides, to defend against code reuse attacks such as ROP, we design and implement an image based address space layout randomization (ASLR) scheme for IoT devices, denoted as iASLR. iASLR is unique since it relocates an image every time the device boots while the image layout is randomized only one time in related work [22]. 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 non-executable RAM and data flash, secure NSC functions and control flow integrity (CFI) for the overall system security of TrustZone-M enabled IoT devices.

Security Framework

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 Rich Execution Environment (REE), namely the NSW. Compared to the SW, the NSW contains most of the program code and tends to communicate with the outside world

Table 5.1: 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.

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 5.1, 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.

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 like UART (Universal Asynchronous Receiver/Transmitter) or SPI (Serial Peripheral Interface) for updating software locally, is universal in microcontroller-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 Non-secure 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 Non-secure 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.

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, 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 non-volatile memory in both the SW and NSW.

Runtime Software Security

Runtime software security is a critical issue for IoT devices, for which C or C++ are preferable programming languages. 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.

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

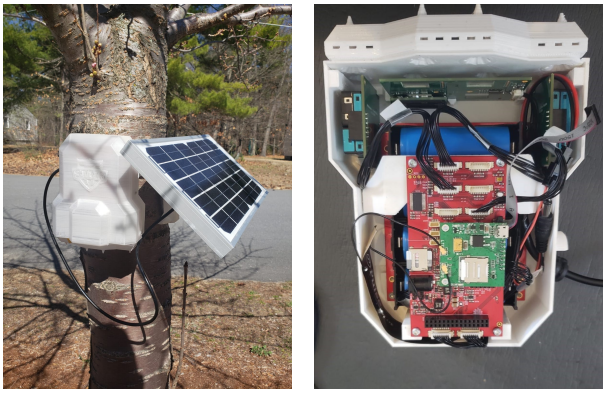
clients are authenticated before the connection is established, the integrity of messages is checked upon being received, and network traffic is encrypted during transmission.

Over-the-air (OTA) Update

OTA update is the process of distributing new software from 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 5, employing secure network protocols is the solution.

Firmware rollback or downgrade attack [20] 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; (2) Validating the signature and version before overwriting the current software. The first stage involves secure network protocol that has been discussed



(**a**)STAIR in the field (**b**)Internals of STAIR

Figure 5.1: Secure and Trustworthy Air Quality Monitoring Device (STAIR).

in the previous subsection. 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.

Implementation: a 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. Figure 5.1 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 ECDSA). It is reasonable to consider

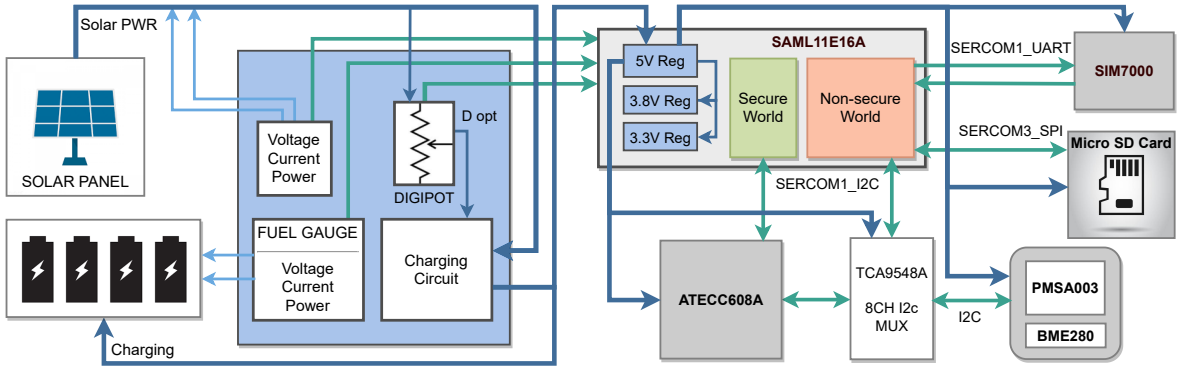


Figure 5.2: Block diagram of the air quality monitoring device – STAIR.

that compromising the hardware and well-known cryptographic algorithms are difficult. Therefore, these mechanisms are trustworthy if they are implemented without errors.

Device Design

Figure 5.2 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 crypto-authentication 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 on-board 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 on-board components use I2C to communicate and allow the main controller board to access data regarding the battery states.

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 Non-secure 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 row can only be programmed and read by Secure

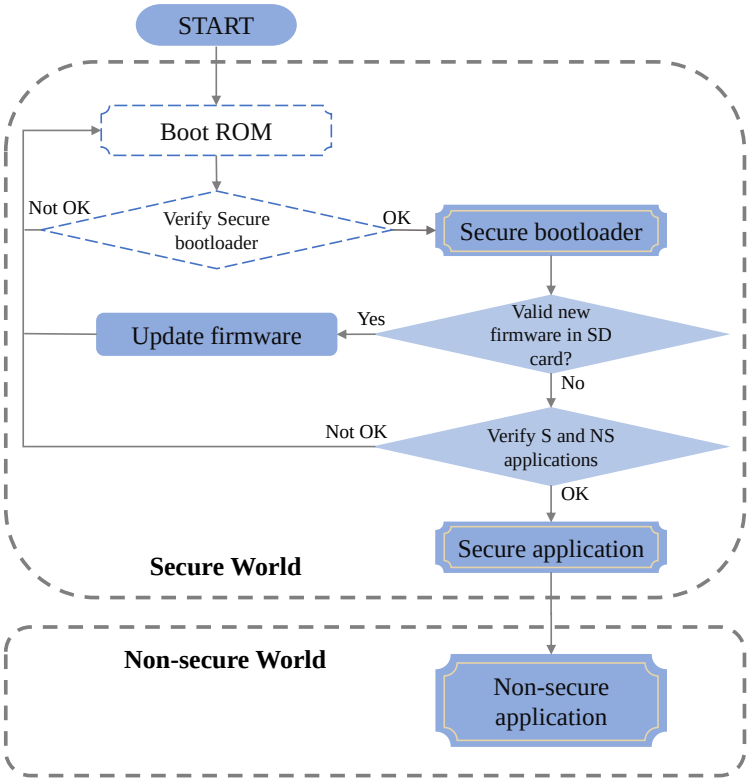


Figure 5.3: Control flow of secure boot. The hollow boxes represent security checks provided by SAM L11, while other boxes are security checks designed by us.

software or external debugger with DAL2. In this way, only one who obtains CEKEY0 is able to access Non-secure memory and one who owns CEKEY0 and CEKEY1, or CEKEY2 can access both Secure and Non-secure memories. And 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.

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 pre-installed 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 Non-secure applications at boot, by which a chain of trust is constructed as presented in Figure 5.3.

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 Non-secure applications before their executions. We implement this validation by using ECDSA (Elliptic Curve Digital Signature Algorithm) signature verification. Specifically, the digests of the applications are calculated by SAM L11's on-device crypto accelerator using 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.

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; (2) Checking any inputs from Non-secure 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

non-executable data. Therefore, we set SAM L11's SRAM and data flash to be non-executable in order to avoid code injection attacks. This is achieved by setting two BOCOR row fuses, namely DXN (Data Flash is eXecute Never) and RXN (RAM is eXecute Never), to 1. In addition, the Secure bootloader will set the Secure and Non-secure code flash to be non-writable, using the memory protection unit (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 4, 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 Non-secure 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 5.

Besides, we introduce two possible security techniques against code-reuse attacks (CRAs) such as ROP. One of them, namely the Address Space Layout Randomization (ASLR) for microcontroller, is devised and implemented in our system.

Control flow integrity

Control flow integrity (CFI) [3] 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. [55] provide an implementation of CFI for TrustZone-M to protect the NSW. In it, a control flow graph (CFG) of the Non-secure program is constructed by static or dynamic analysis of its code and is saved in a non-writable region of the NSW. Code instrumenta-

tion is performed so that the program jumps to a *branch monitor* before any control flow changes in the original code. The branch monitor refers to the CFG and monitors control flow changes at runtime. Before a function call, the correct return address is pushed on a *shadow stack* in the SW. Since a function might be called by different callers and return to different places at runtime, the CFG cannot tell the exact return address at runtime. The shadow stack in the SW is used to record the correct return address for a certain function call. Here the stored return addresses must be fully protected from being altered. The SW, which can be seen as a trust anchor for the NSW, provides the required secure storage, namely shadow stack, for the correct return addresses and a trusted execution environment for any operations on the shadow stack.

The CFI for protecting the control flow of the NSW is not sufficient for the overall system security. It can be observed from Table 4.1 that all software attacks may occur in both the SW (including NSC and SWX) and NSW. Recall that the CFI mechanism in [55] requires a trusted execution environment and a secure storage. In the case of TrustZone-M, the SW is supposed to play the role of such a trust anchor. If the SW itself is insecure and vulnerable to potential software attacks at runtime, it cannot provide the indispensable secure storage and trusted execution environment required by CFI for the NSW. Thus the effectiveness of the CFI enforcement for the NSW would be harmed. Another issue of CFI is that it may not defeat the heap-based BOF or format string attacks if control flows are unchanged but sensitive data are modified. 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 paper.

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [77] 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 5.

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.

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.

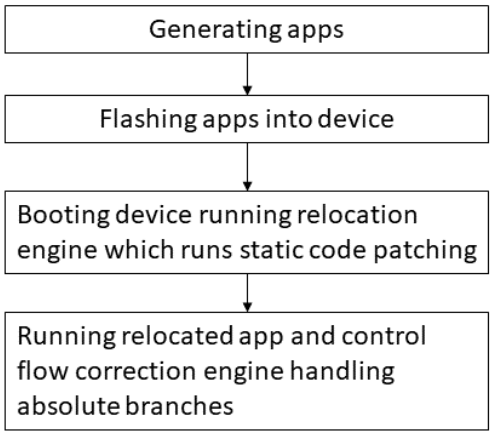


Figure 5.4: Workflow of iASLR.

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.

Workflow and Challenges

Fig. 5.4 gives the workflow of iASLR and Fig. 5.5 illustrates the memory (flash) layout of the iASLR powered system. Recall we run the system directly from the flash. Our image based ASLR always keeps one copy of Non-secure world app image at the start of the Non-secure flash (denoted as base app) and relocates the base app within the free Non-secure flash every time the device boots. In a TrustZone-M enabled system, although we can relocate the Secure app in the Secure world, we assume the Secure app is secure and will focus on the design of relocating the Non-secure (NS) app located in the Non-secure world.

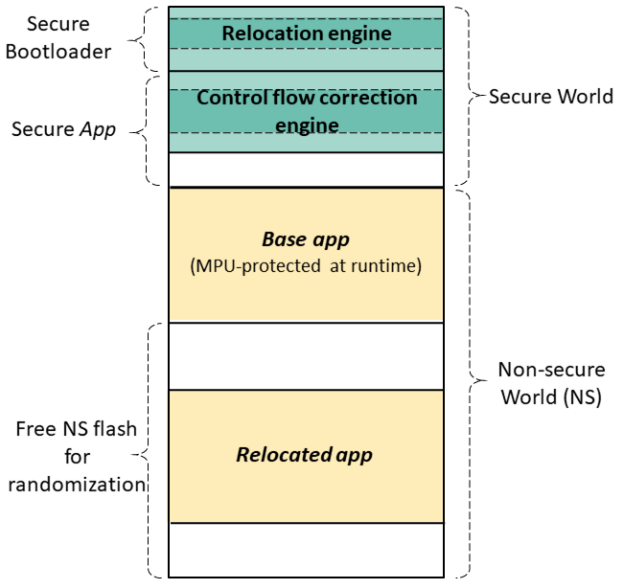


Figure 5.5: Flash layout of an iASLR-enabled system.

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 Non-secure flash (i.e., the Non-secure 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 is stored in the Secure world 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: NSC calls

that call NSC functions in the secure world and 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 Non-secure 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 Non-secure flash. The relocation engine sets the base app image to be non-executable 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 Non-secure app images in our system: the base app image that is always located at the start of the Non-secure flash and the relocated app image somewhere in the rest of the Non-secure flash. The relocation engine also performs **static code patching** to patch NSC calls in the relocated app as detailed in Section 5.

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 5.

Static code patching

Static code patching is applied to all NSC calls in the relocated image. A 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 the NSC call.

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 non-executable, any attempt of executing instructions within the base app image leads to an 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.

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, 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: the base app image and relocated app image. Third, the boot time of an iASLR powered system will increase because of the relocating operations.

Evaluation

Evaluation of the Implemented Air Quality Monitoring Device

Security Evaluation

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 microcontroller, has been installed, to the debug interface through a 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 SHA256 hash function with an authentication key. To evaluate its effectiveness, we 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 non-executable. 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 Non-secure data flash or Non-secure RAM. In both cases, fault exceptions are generated when control flows are diverted. Moreover, Secure flash and Non-secure flash are non-writable 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.

```
1 bool __attribute__((cmse_nonsecure_entry)) nsc_sha256(uint8_t* input,
    uint32_t input_length, uint32_t* output){
2     if((input < NS_FLASH_START) || (input > NS_FLASH_END && input <
    NS_RAM_START)
3     || (output < NS_FLASH_START) || (output > NS_FLASH_END && input <
    NS_RAM_START)){
4         return false;
```

```
5     }
6     if((input < NS_FLASH_END && input_length > NS_FLASH_END - input) ||
    input_length > NS_RAM_END - input){
7         return false;
8     }
9     sec_sha256(input, input_length, output);
10    return true;}
```

Listing 5.1: The NSC function for providing Secure SHA256 algorithm to the NSW.

Defense against NSC-specific attacks: The STAIR sensor application contains one NSC function as shown in Listing 5.1. 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 a 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.

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 Figures 5.6-5.8. The 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.

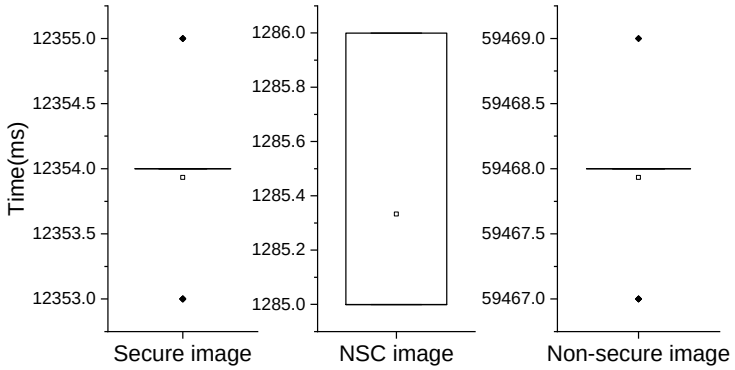


Figure 5.6: Overheads of updating Secure image (5.59KB), NSC image (32B), and Non-secure image (29.8KB) through secure OTA.

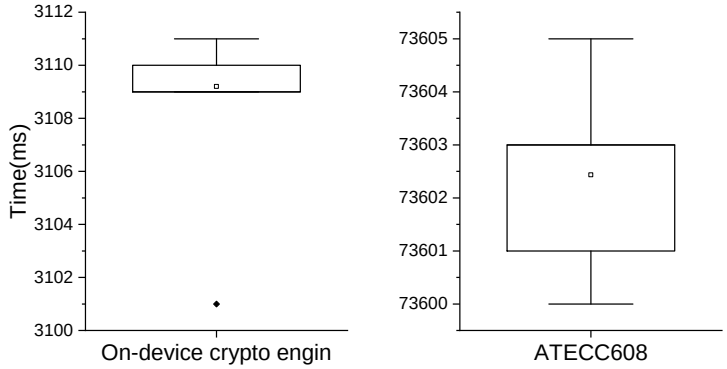


Figure 5.7: Overheads of secure boot using on-device crypto engine or ATECC608 for hashing.

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 Figure 5.6. It is observed that the time taken by validation is approximately proportional to the 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 algo-

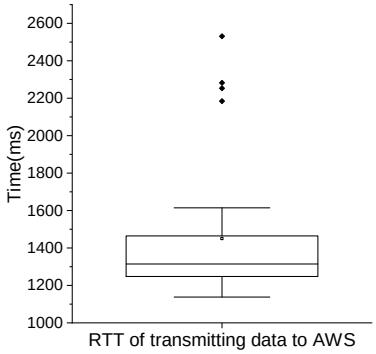


Figure 5.8: RTT of transmitting data to AWS IoT through MQTTS protocol.

rithm, we evaluate the performance of secure boot with each of them. As a result, the on-device hardware crypto accelerator shows a much better performance with average time efficiency of 3.109s than the ATECC608, which needs 66.913s in average.

While sending data to the AWS IoT platform, we use the MQTTS protocol, which requires mutual authentication and traffic encryption. It takes 1300ms 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.

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 [46].

Security analysis

Since iASLR is based on image relocation, any adversary who wants to reuse the image code has to guess the base address of the relocated image every time the device restarts. 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 Equation (5.1), 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.

$$S = \log_2 \frac{f-i}{2} < \log_2 \frac{f}{2} < \log_2 \frac{f_s}{2} = \log_2 f_s - 1 \tag{5.1}$$

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

Table 5.2 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.

Table 5.2: Entropy Bound of ARM Cortex-M23/M33 Enabled Boards

MCU/Development Board	Flash Size	Entropy Bound (bits)
Microchip SAML11 [48]	64KB	15
ARM musca b1 [9]	4MB	21
Nordic nRF5340 [65]	1MB	19
NXP LPC55S69 [54]	640KB	18.32
STM32L562E-DK [70]	512KB	18
NuMicro M2351 [53]	512KB	18

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 5.3, 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 introduces runtime delay. All three applications in Table 5.3 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 0.5ms) 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 change the size of the Non-secure application. For most of the cases in Table 5.3, the overhead is no greater than 10% while there is one with overhead at 17%.

Table 5.3: 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

Summary

In this paper, we present 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 implement the proposed security framework for an air quality monitoring device and evaluate 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.

CHAPTER 6: fASLR: Function-Based ASLR for Resource-Constrained IoT Systems

With the booming IoT industry, there are rising concerns on the security and privacy of IoT devices. IoT application code is often written in unsafe programming languages such as C and C++, and is vulnerable to memory corruption attacks [19]. One typical memory corruption attack is the code reuse attack (CRA), which hijacks the control flow and reuses the application code [13]. Memory corruption attacks and defenses have been actively studied for mainstream operating systems such as Windows, macOS, Lniux, Android and iOS.

In this paper, we focus on defending against memory corruption attacks for resource-constrained IoT devices, particularly those running on micrcontrollers (MCUs). It is an intuitive idea to port existing security schemes to IoT platforms. We study the use of ASLR in memory-constrained IoT devices to mitigate CRAs such as the return-oriented programming (ROP) by randomizing the memory layout of code and data. Modern operating systems often implement the following ASLR scheme. When an executable is loaded into RAM, its base (start) address is randomly chosen while the executable structure is kept almost intact. Fine-grained ASLR strategies have been proposed and randomize executable code at fine levels of basic blocks, functions, or instructions [67] within a loaded application image. However, porting ASLR to resource-constrained IoT devices is a great challenge due to the limited memory space.

We propose a function-based ASLR scheme (fASLR) based on the ARM Cortex-M processor with TrustZone-M enabled [10]. The runtime fASLR is located in the Secure World (SW) of TrustZone-M. The application is in the Non-secure World (NSW) flash and denoted as the NS app, which is protected by the memory management unit (MPU). When a function call occurs, MPU redirects the function call to the runtime fASLR for callee randomization. fASLR can run an application

even if the application on flash is too large to be completely loaded into RAM.

Our major contributions are summarized as follows.

(i) We propose a *function-based ASLR* scheme for resource-constrained IoT devices with limited RAM and flash. fASLR dynamically loads only needed functions into RAM and randomizes their entry addresses so as to achieve large randomization entropy. (ii) Novel schemes are designed for fASLR to perform memory management and addressing. Finished functions are efficiently removed from RAM when there is no RAM for storing more functions. We carefully address the issue of addressing since functions are randomly moved around. (iii) We implement fASLR with a TrustZone-M enabled MCU, SAM L11, and validate the feasibility and performance of fASLR with 21 applications. fASLR incurs a runtime overhead of less than 10% for all the applications.

System Architecture

In this section, we first present the threat model and design goals of our ASLR scheme—fASLR. We then introduce the architecture of a fASLR-enabled system and the workflow of fASLR. At last, we discuss challenges for implementing a practical fASLR.

Threat Model

fASLR leverages ARM Cortex-M processors and hardware-based techniques including TrustZone-M, memory management unit (MPU), and exception handling mechanism. Based on the hardware isolation provided by TrustZone-M, on-device system resources are divided into two worlds, namely the Secure World (SW) and the Non-secure World (NSW).

We assume a TrustZone-M enabled device has the following security features. (i) Main compo-

nents of fASLR reside in the SW and can be fully trusted. The application (denoted as NS app) is located in the NSW and may be vulnerable. (ii) The NS app is located at a fixed address in the NS flash and is executed from the flash (instead of RAM) by default. (iii) The device supports the memory protection mechanisms such as the MPU.

We assume an adversary has the following capabilities. (i) The NS app may be subject to CRAs such as the ROP attack. (ii) The adversary can obtain the binary of the NS app, disassemble the binary, and obtain code gadgets for CRAs.

Design Goals

fASLR is designed to achieve the following goals.

- **Mitigating CRAs**. The scheme shall provide dynamic function-level code randomization for resource-constraint IoT devices to mitigate CRAs, which require a certain chain of gadgets found in the NS app. The randomization shall achieve high entropy to defeat brute-force guessing attacks.
- **Usability**. The scheme shall be user-friendly and will not add much burden of programming.
- **Low runtime overhead**. The proposed scheme cannot introduce large overhead in terms of time and space and affect the NS app performance much.

System Architecture

As illustrated in Fig. 6.1, fASLR has three key components. (i) the ***Static Preprocessing Module (SPM)*** for compilation time preparation, (ii) The ***Boot Engine (BE)*** for boot time configuration, and (iii) The ***Function Randomization Engine (FRE)*** for runtime function-level randomization.

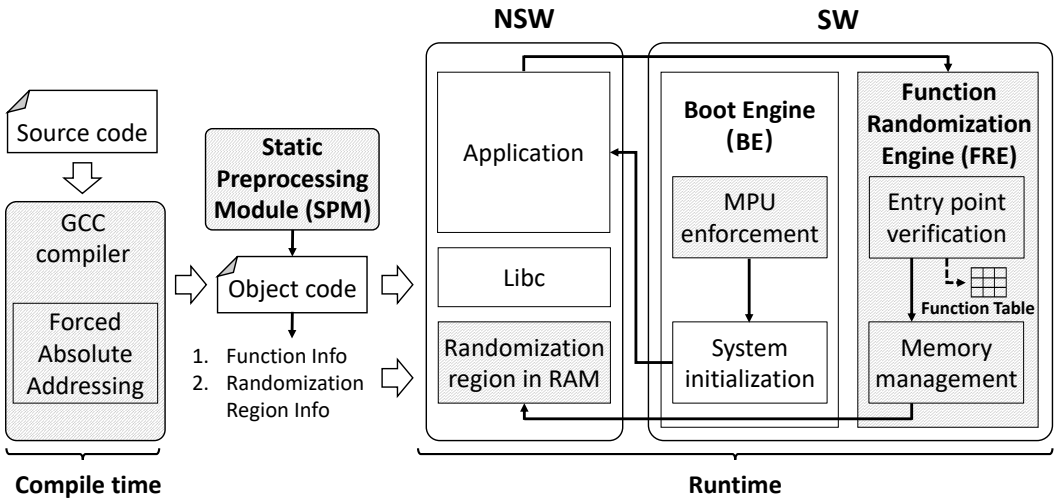


Figure 6.1: fASLR architecture.

Static Preprocessing Module (SPM). The *SPM* serves two major purposes: (i) Creating the *Function Table*. Once the NS app code is compiled via a GCC compiler, the SPM tries to extract needed information of all functions in the ELF object file, including function entry point addresses and function sizes from the symbol table, and function stack frame sizes from `.debug_frame` section of the ELF file. Function information is recorded in a data structure called *Function Table*. (ii) Profiling randomization region. The *SPM* determines the size and location of the *randomization region* that is a free memory chunk available in the NS RAM.

Boot Engine (BE). When a TrustZone-M enabled device boots, the boot flow is the Secure bootloader, Secure app, and then the NS app. The NS app starts with the `reset` handler that calls the first function, e.g., `main()`. The *BE* is a part of the Secure bootloader stored in the SW flash. It configures and enables the MPU to mark the NS app code in the flash as non-executable for two purposes: (i) MPU prevents the NS app in the NS flash from being exploited by CRAs. (ii) Once the NS code is set as non-executable, any attempt to execute the NS code triggers a hardware exception, which is handled by the `HardFault` exception handler in the SW [7].

Function Randomization Engine (FRE). The *FRE* is a part of the `HardFault` exception handler

and handles invoked functions in the NSW flash protected by the MPU. It serves two purposes, i.e., **function entry point verification** and **memory management**.

When a `HardFault` exception occurs, the *FRE* fetches the return address of the exception, which is the entry point of the invoked function, through the NSW exception stack frame. Then the **function entry point verification** is performed by comparing the return address to all legitimate function entry point addresses in the *Function Table* until there is a match. After a match, the *FRE* obtains the size of the corresponding function from the *Function Table* for later use in randomization. A `HardFault` exception may be caused by other reasons, for instance, memory access violation when an adversary launches CRAs trying to execute an instruction not at legitimate function entry points. In such a case, a security alert shall be raised.

After the function entry point verification, **memory management** is performed. Specifically, the invoked function is randomly relocated to a RAM region within the *randomization region*. After the function randomization, we need to carefully handover the control flow from the exception handler to the relocated function. We deliberately manipulate this by overwriting the return address of the exception handler on stack with the new function entry point so that the execution mode will change back to the mode of the NS app with correct privileges.

Workflow

Offline—compilation and flashing. After the NS app is compiled and linked by the GCC compiler at compile time, the *SPM* creates the *Function Table* offline according to the NS app ELF file. The *Function Table* and NS app image are then flashed to the SW and NSW flash respectively. The *BE* and *FRE* are also flashed to the SW flash.

Runtime. Fig. 6.2 shows the program flow, which is an iterative sequence of function calls (e.g.,

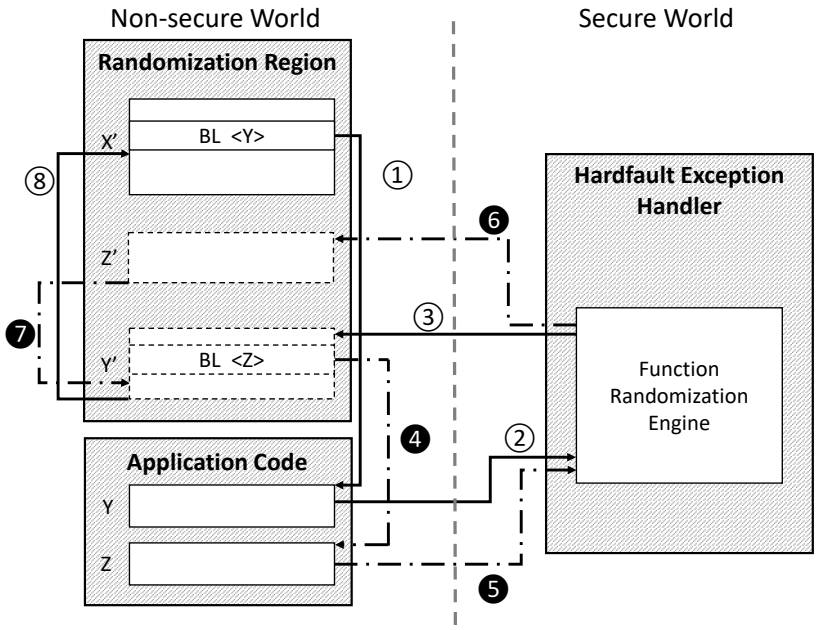


Figure 6.2: Program flow of function X, Y and Z. For any function F in the NS app, we use F' to represent its corresponding copy in the randomization region.

① and ❹), MPU violation exception (e.g., ② and ❺), runtime function randomization, function execution (e.g., ③ and ❻), and function return (e.g., ❼ and ⑧).

Once we turn on the device power supply, the *BE* boots the system and then the NS app initiates the `reset` handler [81]. After a sequence of initialization operations, the `reset` handler branches to the main application code, i.e., *main()*. Both attempts of executing the reset handler and *main()* trigger runtime fASLR, and their code is loaded by the *FRE* to the *randomization region* in the RAM for execution. During the execution of *main()* function, the control flow can divert to a callee on the MPU-protected flash only if the callee is invoked by *main()*.

In a fASLR-enabled system, when a function call occurs in the randomization region, it jumps to the entry point of the callee in the original MPU-protected application, and thus triggers the MPU violation exception. The *FRE* then conducts runtime function randomization for the callee and diverts the control flow to the callee relocated in the RAM. During the execution of the callee, any

function call in the callee can also trigger a MPU violation exception. The *FRE* handles function calls and randomization in such an iterative way above. The control flow returns to the caller in the RAM once its callee is finished. fASLR does not interfere with the function return mechanism, and the function in the RAM returns normally as functions in a system without fASLR do. Note that in a fASLR enabled system, a callee returns to the relocated caller in the randomization region since that is where the function call really occurs.

Fig. 6.2 presents an exemplary program flow for the call path $X \rightarrow Y \rightarrow Z$ of functions X, Y and Z. A call path illustrates the calling relationship. Starting from the leftmost one, each function in the path calls the function right after it. Suppose that function X has been loaded to the randomization region and the program flow starts from the relocated function X'. When X' calls Y, the attempt of executing Y (①) results in a MPU violation exception (②), handled by the *FRE* inside the `HardFault` exception handler. Y is then relocated to the randomization region as Y' and consequently, the control flow is redirected to Y' (③). During the execution of Y', Y' attempts to call Z (❹), and the MPU violation exception (❺) is triggered again and is then handled by the *FRE* (❻). Finally, the control flow returns from Z' to Y' (❼) and Y' to X' (⑧) when Z' and Y' finish execution.

Challenges

A practical fASLR faces the following challenges. We address these challenges in detail in Section 6.

Memory management. We target MCUs with limited RAM and the whole NS app may not be loaded into the RAM for execution. Therefore, a memory management strategy is needed to dynamically trim finished functions. *Ancestor functions* are defined as direct or indirect callers of the current running function. Such functions are awaiting returns from some ongoing function

calls, and shall not be trimmed before their descendants return. *Finished functions* are those that have finished execution and are not ancestors of any running function. They can be disposed safely. The runtime *FRE* needs to distinguish finished functions and select an appropriate timing to trim them from the randomization region.

Memory Addressing. A function in the randomization region may contain branches that use absolute or relative addresses. All absolute branches in the ARMv8-M architecture compiled by GCC are function calls, which would not be affected by the relocation and will function normally. Relative branches within a function can work normally as well since a relative position would not change when the function is relocated as a whole. However, relative branches may be used to jump between functions. In a fASLR-enabled system, those relative branches can lead the control flow to branch to an unexpected destination as function-based randomization changes the relative position of two functions.

Memory Management and Addressing

In this section, we discuss the challenges of fASLR and present our key memory management and addressing schemes.

Memory Management

We devise a function cleaning scheme that dynamically cleans up finished functions from the *randomization region* with the following mechanisms: (i) *Call stack unwinding*. Finished functions are found through unwinding the Non-secure call stack. (ii) *Cleaning on demand*. Finished functions are cleaned up only if the available randomization region space is not large enough for the callee. (iii) *Call instruction rewriting*. We further reduce the runtime overhead by overwriting a

call instruction in a loaded function if the callee of that call instruction has already been loaded into RAM.

Call stack unwinding. The key of function cleaning is to distinguish finished functions from all loaded functions in RAM. However, it is difficult to trace all finished functions at runtime because fASLR runtime does not capture any function return information. Instead, our approach finds ancestor functions of the current callee, and records all loaded functions. *Any function that is a loaded function but not an ancestor function is a finished function that can be disposed.* Now the problem is decomposed to record all loaded functions and find all ancestor functions.

A queue structure named *Loading Queue* is used to store meta data, denoted as function record, of all loaded functions in the RAM. The *FRE* pushes a function record into the *Loading Queue* when an unloaded function is called. A function record includes the following information of the callee, (i) *loadAddress*–the new entry point of the callee in the randomization region, (ii) *size* of the callee, and (iii) *stack frame size* of the callee.

We also need to find out all ancestor functions. Modern computer system uses the call stack to retain return addresses of functions that have been called but have not returned yet. Such functions are direct or indirect callers of the current running function, which is also the callee when program execution is trapped in the *FRE* in our system. Therefore, functions which have their stack frames in the call stack are ancestor functions of the callee. To figure out all functions in the call stack, **stack unwinding** is needed. Basically, stack unwinding helps to locate **all return addresses in the call stack**. A return address then tells where the caller is within the RAM. If we can obtain all return addresses on the call stack, by comparing each return address with the function records in the *Loading Queue*, we are able to identify all ancestor functions.

Frame pointer is an intuitive approach of unwinding the call stack [30]. However, the Armv8-M architecture only implements the Thumb instruction set, which does not support the frame pointer

Algorithm 2: Call Stack Unwinding

```
nsSp = getNSSp() returnAddress = readExceptionStackFrame(nsSp)
 funcSp = nsSp + sizeof(ExceptionStackFrame) for i = 1; i < loadingQueue.size; i++ do
   if (returnAddress ≤ loadingQueue[i].endAddress) & (returnAddress ≥
    loadingQueue[i].loadAddress) then
      funcRecord = loadingQueue[i]
      funcRecord.state = UNFINISHED
      funcSp = funcSp + funcRecord.callFrameSize
      returnAddress = getReturnAddress(funcSp)
   end
end
```

mechanism. To achieve stack unwinding without frame pointers, we devise a stack unwinding method utilizing the stack top address and the stack frame sizes of all functions to resolve return addresses on the call stack. Recall that the stack frame size of each function is extracted offline by the *SPM* from `.debug_frame` section of the ELF file and stored in the *Function Table*.

In ARM, by convention return address is the first object pushed onto the stack when there is a function call, and is at the bottom of the callee's stack frame. Once a function call triggers the `HardFault` exception and the program execution is trapped by the *FRE*, the top stack frame is the exception stack frame of the `HardFault` exception handler and has a fixed length s_e. The current stack top can be obtained through the `SP` register of the NSW. The frame top of the first function f_1 (namely the current caller) is $T_1 = SP + s_e$. According to the `LR` register, which stores an address within f_1, *FRE* is able to search the frame size s_1 of f_1 from the function records in the *Loading Queue*. To access the return address of f_1, the frame bottom B_1 is calculated by $B_1 = T_1 + s_1$. Following this procedure, the *FRE* is able to resolve return address of every stack frame from the stack top to bottom. Algorithm 2 presents our stack unwinding procedure.

Not that recursion is compatible with our function cleaning strategy. A recursive function is the function that calls itself. In our compilation environment, a recursive function uses a relative branch instruction. Therefore, when a recursive function is loaded to the randomization region, it

can still call itself with the relative branch without triggering the MPU violation exception.

Cleaning on demand. The *FRE* removes functions only when the randomization region does not have enough space to load a new function. Before loading a function to RAM, the *FRE* checks if there is enough memory space for it. If not, the *FRE* first recovers rewritten call instructions in the loaded functions as introduced below. It then unwinds the call stack, finds out all ancestor functions, and marks those functions in the *Loading Queue* as unfinished. According to the marked *Loading Queue*, the *FRE* disposes all finished functions and updates the *Loading Queue*.

Call instruction rewriting. Function call rewriting optimizes the memory management scheme so that finished but not disposed functions in RAM can be called again without triggering the `HardFault` exception. Specifically, when a function call occurs and the control flow is trapped in the `HardFault` exception handler, the *FRE* overwrites that call instruction (in the loaded caller) to change the destination address of the call (i.e., the entry point of the callee in flash) with the entry of the loaded callee in RAM. Thus, the caller will directly jump to the loaded callee next time this call instruction executes. The rewriting history, including which instruction is rewritten and what the original instruction is, is recorded in the *Rewriting List*. Such records are used to recover the call instructions with callees' flash entry points before function cleaning, since the loaded callees of those call instructions might be disposed.

Memory Fragmentation Management. In the randomization region, each loaded function occupies a function block. A disposed function block becomes a free block. If there are any adjacent free blocks, the *FRE* merges them into one big free block. All free blocks are managed by a linked list. A function block, as presented in Fig. 6.3 (a), consists of a two-word metadata, a payload and padding bytes for memory alignment. The metadata contains the size of the block and the pointer which points the next free block. The payload region is used to stores the randomized function.

When the system starts, fASLR initializes the whole randomization region as a big free block

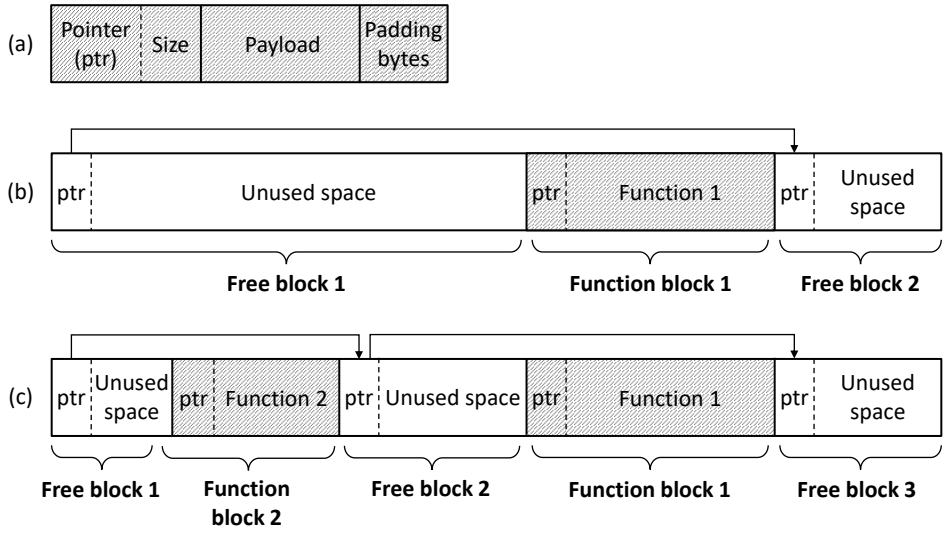


Figure 6.3: Memory Management. (a) Structure of the randomization region; (b) Memory layout before loading Function 2; (c) Memory layout after loading and randomizing Function 2

since no function has been allocated yet. Once a function is called in the NSW, the *FRE* allocates a function block for the target function. Specifically, it first scans the linked list and finds out all free blocks larger than the target function in the payload region. The *FRE* randomly selects one block among the discovered free blocks and then randomly allocates the target function to the selected free block. After the allocation, new free blocks may be generated and the linked list will be updated accordingly. Fig. 6.3 (a) and (b) illustrate the case of randomizing Function 2 when there are two free blocks. Function 2 is consequently allocated to the middle of free block 1. The new free block 1 and free block 2 are then formed.

Memory Addressing

Control flow instructions using relative addresses in the ARMv8-M instruction set include *B* (branch), *BL* (branch with link), and *CBNZ/CBZ* (conditional branches), among which the *BL* (branch with link) is used to branch between functions. It is difficult for fASLR to handle such

relative addressing without instruction patching, namely runtime instruction update. Note that the relative positions of two functions changes after function randomization. Recalculating all the relative addresses used in the randomized function and updating the related instructions with the new relative addresses will result in unacceptable overhead in performance.

fASLR eradicates relative addressing at compile time. A user can compile the app (including libraries) with specific compilation flags (i.e., *-mlong-calls*, *-fno-jump-tables*). As a result, all relative function calls use absolute addressing.

Security and Performance Analysis

In this section, we analyze the effectiveness of fASLR against ROP, a representative code reuse attack. Entropy is computed to quantify the randomness of gadgets required for the ROP attack, which indicates the difficulty of guessing the gadget locations in a brute-force way. We also study time and memory overheads introduced by fASLR.

Effectiveness against ROP

The prerequisite of ROP is that the adversary knows where the ROP gadgets are. In a fASLR enabled system, an adversary can only use ROP gadgets in randomized functions relocated to the randomization region. Gadgets in the NS app stored in flash are non-executable, so it is hard for adversaries to use them. Recall that any MPU violation triggers the `HardFault` exception. As discussed in Section 6, the *FRE* validates the return address of the exception by using the *Function Table*. Therefore, the *FRE* is incapable of identifying exceptions triggered by a ROP attack if the adversary targets the entry point of a function since normal function calls will trigger such exceptions as well. In other words, the adversary will succeed in reusing a whole function as a

gadget for ROP attack. However, such gadgets are often of very low quality [27, 15] containing too many instructions. It is almost impossible for an adversary to assemble a chain of gadgets with such low quality gadgets to achieve certain malicious goal.

An adversary may also guess the addresses of randomized functions in a brute-force way. However, our runtime randomization approach rebases a function every time as long as it has not been loaded into RAM and achieves high randomization entropy as analyzed below.

Randomization Entropy

fASLR mitigates the brute-force guessing attack from two aspects. On the one hand, fASLR restricts the number of functions that can be reused at a time. This is achieved by configuring the whole app image as non-executable. The only code snippets that can be utilized are functions relocated to the randomization region in the RAM.

On the other hand, even if all the required gadgets can be found from the relocated functions, the adversary has to guess locations of all those functions at once. Formula (6.1) gives the total number (denoted as C) of possible function layouts in the randomization region.

$$C = k! \binom{V+k}{k}, \tag{6.1}$$

where k is the number of functions in the randomization region, and V is the size of unused randomization space divided by two since the ARMv8-M architecture only allows even function addresses.

Formula (6.1) can be proved as follows. Counting the number of possible layouts of a set of functions in the randomization region is a combinatorial problem. That is, given a set of k loaded functions $f_1, f_2, ...f_k$ and a free randomization space with a size of $2 * V$, how many combinations

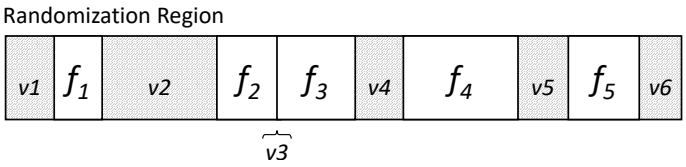


Figure 6.4: Randomization layout with 5 loaded functions $f_1, f_2, ..., f_5$. The 5 functions form 6 vacancies $v_1, v_2, ..., v_6$ where free randomization units can be placed.

do we have to arrange the loaded functions and the free randomization space? Note the ARMv8-M architecture only allows an function to be loaded to an even address. Therefore, the randomization space can be treated as V free randomization units and each unit is 2 bytes. The problem now is how to count the combinations of the k functions and V free units. We solve this counting problem based on the "Stars and Bars" problem [75, 26]. Fig. 6.4 illustrates a randomization layout with 5 loaded functions. The shaded portions in the randomization region are free spaces. Analogous to the bars in "Stars and Bars", the V free units can be placed in the vacancies (denoted as v_i) between the k loaded functions (analogous to stars), before the first function, and after the last function. That is, we need to allocate V free units to $k+1$ vacancies (i.e., $v_1, v_2, ..., v_{k+1}$). For each vacancy, it can contain at least zero and at most V free units. Therefore, the combinations can be counted by the binomial coefficient $\binom{V+k}{k}$ multiplied by $k!$ since the k loaded functions are distinguished. In the example where $k = 5$ and $V = 6$, there are $5!\binom{6+5}{5} = 55440$ combinations.

The probability of a layout is the reciprocal of C, i.e., $P = 1/C$. Formula (6.2) gives the entropy H of function randomization.

$$H = -\sum_{i_1=1}^{C} P \log_2 P = -\sum_{i_1=1}^{C} \frac{1}{C} \log_2 \frac{1}{C} = \log_2 C \tag{6.2}$$

Time Overhead

fASLR introduces runtime overhead when it hijacks a function call for function randomization via hardware exception. According to fASLR runtime mechanism, we consider three factors that affect the program runtime performance, namely the number of function calls N_c that trigger `HardFault` exceptions, function randomization processing time for the ith function call denoted as $T_R(i)$, and hardware exception processing time T_E. Formula (6.3) gives the relationship between the time overhead TO and the three factors.

$$TO = \sum_{i=1}^{N_c} (T_R(i) + T_E). \tag{6.3}$$

Intuitively, $T_R(i)$ would be much larger than T_E since T_R involves several time-consuming operations such as memory write and table scanning, while T_E is accomplished by hardware. The overhead from the function randomization process primarily comes from the following aspects: (i) address verification, which involves looking up the *Function Table*; (ii) function cleaning on demand, which looks up the call stack and cleans up finished functions; (iii) randomization, which selects a free block to rebase the callee; (iv) function loading, which reads and writes the function body; (v) function rewriting, which overwrites the destination of the call instruction with the entry point of loaded function.

Memory Overhead

The components of fASLR deployed in the SW include the *BE* code, *FRE* code, *Function Table*, *Loading Queue*, and *Rewriting List*. The *Function Table* is a static table with three 4-byte attributes and its size is linear to the total number of functions in the NS app. The *Loading Queue* and *Rewriting List* are dynamic data structures that contain function records and rewriting records

respectively. Each function record has three 4-bytes metadata and a rewriting record contains double 4-bytes data. The maximum number of records that the *Loading Queue* may use at runtime is equal to the number of functions in the NS app, while the maximum number of rewriting records in the *Rewriting List* is the total number of call instructions. Formula (6.4) presents the size of the *Function Table* (i.e., MO_t), *Loading Queue* (i.e., MO_q), and *Rewriting List* (i.e., MO_l),

$$MO_t = N_f \times 3 \times 4 = 12N_f, \tag{6.4}$$

$$MO_q = N_p \times 3 \times 4 = 12N_p, \tag{6.5}$$

$$MO_l = N_c \times 2 \times 4 = 8N_p, \tag{6.6}$$

where N_f is the number of functions in the NS app, N_p is the number of functions in the longest NS call path, N_c is the number of function calls in the NS app.

Evaluation

In this section, we first present the experiment setup. We then present evaluation of randomization entropy, runtime overhead and memory overhead.

Experiment Setup

fASLR is implemented and deployed on the SAM L11 Xplained Pro Evaluation Kit, a MCU development board using the ARM Cortex-M23 core with TrustZone-M enabled. SAM L11 has a 64KB flash and a 16KB SRAM.

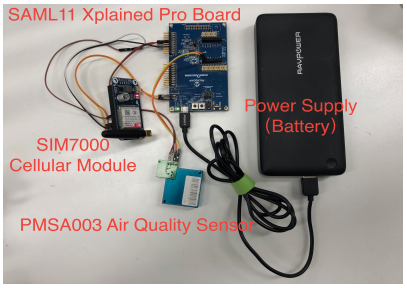


Figure 6.5: Air quality monitoring device.

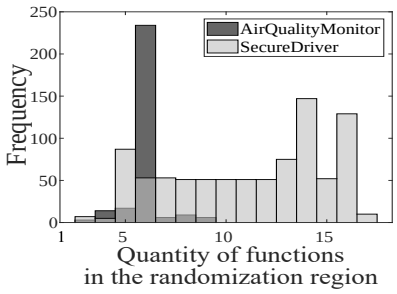


Figure 6.6: Histogram of function quantity in the randomization region.

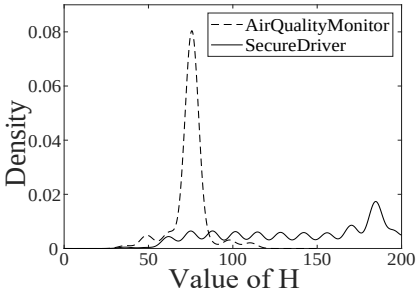


Figure 6.7: Kernel distribution of the global entropy.

Software in SAM L11 is built with the GNU Arm Embedded Toolchain. User code, namely the NS app code, is compiled with two flags, `-mlong-calls` and `-fno-jump-tables`, to eliminate instructions using relative addressing. We recompile the C library with the same compiler flags. A Python script runs during the compilation time to collect function metadata and saves them in the *Function Table*. fASLR program and the *Function Table* are part of the Secure application placed in the SW flash, while the user app is deployed in the NSW flash.

We evaluate the performance of fASLR with 21 applications, including our own air quality monitoring system *(AirQualityMonitor)*. The air quality monitoring device, as shown in Fig. 6.5, consists of a SAM L11 development board, a PMSA003 air quality sensor module, and a SIM7000 cellular module. The NS app in SAM L11 periodically receives air quality data from PMSA003 and sends the data to SIM7000, which then transfers the data to the AWS IoT platform via secure MQTT protocol. The other twenty apps including the CoreMark benchmark [23], two micro benchmarks *Cache Test* and *Matrix Multiply* created based on [61], nine benchmarks of BEEBS [57], and eight SAM L11 demo apps obtained from Atmel Start [47].

Randomization Entropy

The entropy of function randomization changes dynamically when a function call occurs. We explore the changing entropy with two representative apps by examining the functions in the randomization region after each function call and collect k and V in Formula (6.1). Fig. 6.6 presents the histogram of k in the randomization region. We can see that for the *AirQualityMonitor* application, most values of k are 6, while for the *SecureDriver* application, values of k spread are more diverse. This is because *SecureDriver* contains more long call paths than *AirQualityMonitor*.

For each pair of k and V, we calculate the corresponding entropy of function randomization according to Formulas (6.1) and (5.1). Fig. 6.7 shows the distribution of H based on kernel based density estimation. It can be observed from Fig. 6.7 that fASLR achieves good randomization entropy ranging from 30 to 100 for *AirQualityMonitor*, and 50 to 170 for *SecureDriver*.

Runtime Overhead

fASLR introduces runtime overhead since it intercepts every function call of the NS app for function randomization. We evaluate the time overhead by measuring and comparing the execution time of an application with and without fASLR. We use the internal `systick` timer of the Cortex-M core to record the execution time with precision of 0.01s. Since the main program of an IoT application is usually a big loop, in the experiments we measure the execution time of 1000 loops for each testing application. We comment out all `delay` functions inside the loop for better estimation of time overhead introduced by fASLR. Table 6.1 presents the total execution time of 1000 loops for each application. The runtime overhead of fASLR is less than 10% for all apps, and 14 apps achieve time overheads below 5%. We also count for the occurrence of function cleaning for each app. The result shows that 19 apps have exhausted memory space during execution and

Table 6.1: Total Execution Time (in second) of 1000 Loops and Overheads.

Application	**# of cleanings**	**Baseline**	**with fASLR**	**Overhead**
AirQualityMonitor	1	324.79s	327.50s	0.83%
CoreMark	4	15.62s	15.78s	1.24%
SecureDriver	1	12.56s	12.64s	0.64%
ADC Event System	2	12.41s	12.54s	1.04%
Calendar	0	50.36s	50.33s	0.06%
Light Sensor	1	24.77s	25.36s	2.38%
Low Power for SAML1X	0	14.60s	14.60s	0%
ADP Hello	1	9.93s	10.88s	9.57%
SAML11-CRYA	1	6.79s	7.35s	7.07%
SAML11-TrustRAM	1	1.14s	1.25s	9.65%
Cache Test	2	2.13s	2.26s	6.10%
Matrix Multiply	1	24.47s	26.13s	6.78%
Beebs-crc	1	7.44s	7.73s	3.90%
Beebs-aha-mont64	1	7.30s	7.73s	3.56%
Beebs-aha-compress	1	4.50s	4.67s	3.78%
Beebs-bs	1	0.28s	0.29s	3.57%
Beebs-bubblesort	1	0.33s	0.35s	6.06%
Beebs-compress	2	2.02s	2.18s	7.92%
Beebs-md5	2	0.42s	0.44s	4.76%
Beebs-levenshiein	1	17.42s	17.97s	3.27%
Beebs-edn	2	15.96s	16.24s	1.75%

triggered at least one function cleaning.

Memory Overhead

For each tested application, we measure the total number of functions and the memory overhead of NS apps before and after deploying fASLR, as illustrated in Table 6.2. In the SW, the code overhead is caused by the program of fASLR with a fixed code size of 3.45KB, and the data overhead is mainly introduced by the static *Function Table*, dynamic *Loading Queue* and *Rewriting List*, and thus depends on the number of functions in the NS app. The size of the NS app is changed

Table 6.2: NS App Size (in byte) and Overheads.

Application	**# of Functions**	**Baseline**	**with fALSR**	**Overhead**
AirQualityMonitor	148	41092	43036	4.73%
CoreMark	174	46048	47648	3.47%
SecureDriver	139	39544	41184	4.14%
ADC Event System	173	43036	44640	3.72%
Calendar	97	36780	36808	0.08%
Light Sensor	132	40496	40528	0.08%
Low Power for SAML1X	67	34136	34164	0.08%
ADP Hello	99	38072	38316	0.64%
SAML11-CRYA	143	41368	43012	3.97%
SAML11-TrustRAM	142	39896	41500	4.02%
Cache Test	140	40228	41844	4.02%
Matrix Multiply	145	40728	42404	4.12%
Beebs-crc	138	39944	41492	3.88%
Beebs-aha-mont64	142	40476	42028	3.83%
Beebs-aha-compress	140	39944	41492	3.88%
Beebs-bs	137	39344	40896	3.94%
Beebs-bubblesort	137	39932	40892	2.40%
Beebs-compress	143	40808	42360	3.80%
Beebs-md5	137	41552	43100	3.73%
Beebs-levenshiein	138	39708	41348	4.13%
Beebs-edn	144	42112	43736	3.86%

because of the compilation with specific compiler flags. Table 6.2 shows little memory overhead below 5% for all tests.

Discussion

In this section, we first present the memory limitation of fASLR, i.e., the size requirement of randomization region for a certain application. We also discuss how fASLR can be used together with other security defense mechanisms.

Compatibility with Other Security Mechanisms

Data Execution Prevention. FAIRS can work with the data execution prevention (DEP), which is supported by many commercial MCUs. DEP prevents execution from the data region such as stack and heap, and can effectively mitigate against the code injection attack including the buffer overflow attack. It is widely believed that combining ASLR with DEP is more effective than using ASLR or DEP alone [73].

The ARM's DEP technology is called eXecute-Never (XN) bits [1] which are used to mark certain memory regions as non-executable. Any attempt of executing an instruction from the region tagged as XN will result in a `HardFault` exception. Our scheme requires an executable randomization region for runtime function randomization and execution. Therefore, any other memory region can be set to XN for security concerns. Although the violation of XN triggers the same fault exception in ARMv8-M architecture, it is trivial for FAIRS to distinguish such exceptions at the verification stage by judging if the exception return address is within the application code region or not since FAIRS only handles exceptions triggered by attempts of executing functions in the application region.

Control Flow Integrity. Control flow integrity (CFI) is another countermeasure against CRAs. Unlike ASLR which makes the locations of gadgets unpredictable, CFI prevents the control flow hijack by monitoring any control flow changes, ensuring branch instructions branch as intended. CFI technique is not as mature as ASLR and has not been widely used in industry due to its runtime overhead. Although some CFI implementations have been deployed, they cannot mitigate all CRAs independently. For example, Windows 10 supports both ASLR and control flow guard (Windows' implementation of CFI) as exploit protection mechanisms [2].

CaRE [55] is a CFI implementation for IoT devices based on the Cortex-M processors. CaRE

intercepts any direct and indirect branches by replacing those instructions with the supervisor call (*svc*) so that the control flow is trapped to its monitor code. FAIRS can work compatibly with CaRE. When combined with FAIRS, the supervisor calls resulted by branches will be processed by FAIRS and the monitor code of CaRE will be randomized. These extra supervisor call of CaRE, as we analyzed in Sections 6 and 6, introduce additional time and memory overheads.

Summary

In this work, we propose fASLR, a function-based ASLR scheme for runtime software security of resource-constrained IoT devices, particularly those based on microcontrollers. fASLR leverages hardware-based security provided by the TrustZone-M technique as the trust anchor. It uses MPU and prevents direct code execution of the application image in the Non-secure world flash. Instead, it traps control flow in an exception handler and relocates functions to be executed to a randomly selected location within the RAM. A memory management strategy is designed for allocating and cleaning up functions in the randomization region. fASLR is user friendly and only requires a user compiling the app with specific flags. We implement fASLR with a TrustZone-M enabled MCU—SAM L11. fASLR achieves high randomization entropy with acceptable overheads. We will release fASLR to GitHub for broad adoption and refine the implementation to further reduce the overhead.

CHAPTER 7: CONCLUSION

In this dissertation, we perform comprehensive security analysis on low-cost and resource-constrained IoT systems. We investigate potential networking issues in a real-world sensor network. According to our analysis, we devise network attacks in three scenarios, and are able to pollute sensor data with fabricated data via man-in-the-middle attack, spoofing attack, wardriving attack, and device scanning attack. We also propose corresponding defense strategies. We figure out five attack vectors aiming IoT systems and present a well-rounded security framework for IoT devices which use TrustZone-M enabled microcontrollers. We systematically analyze runtime security issues of such IoT devices and present potential pitfalls of TrustZone-M programming. To avoid pitfalls and defend against runtime software attacks, guidelines for the overall system security of TrustZone-M enabled IoT devices are presented. The proposed security framework is implemented for an air quality monitoring device. We also perform evaluation on the performance of the implementation and the proposed security mechanism. fASLR is proposed for runtime software security of MCU-based IoT devices. Employing TrsutZone-M as the trust anchor, we deploy runtime fASLR in the Secure World to relocate functions at runtime. A memory management mechanism is devised to distinguish and clean up finished functions on demand. fASLR achieves large randomization entropy with the cost of acceptable overheads.

LIST OF REFERENCES

[1] Arm11 mpcore processor technical reference manual.

[2] Apply mitigations to help prevent attacks through vulnerabilities, 2021.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), November 2009.

[4] Tigist Abera, Lucas N. Asokan, Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[5] Amazon. Amazon ec2. `https://aws.amazon.com/ec2/`, 2018.

[6] Amazon. Amazon ec2 pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`, 2018.

[7] ARM. Armv8-m fault handling and detection.

[8] Arm. Armv8-m secure software guidelines.

[9] ARM. Musca-b1 test chip board. Last accessed 2021-9-30.

[10] Arm. Trustzone for cortex-m.

[11] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, 2010.

[12] Berkeley Information Security Office. Secure coding practice guidelines, 2020.

[13] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. Mitigating code-reuse attacks with control-flow locking. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 353–362. ACM, 2011.

[14] Brand Media, Inc. Where is geolocation of an ip address? `https://www.iplocation.net/`, 2018.

[15] Michael D Brown and Santosh Pande. Is less really more? why reducing code reuse gadget counts via software debloating doesn't necessarily indicate improved security. *arXiv preprint arXiv:1902.10880*, 2019.

[16] Center for Applied Internet Data Analysis. Internet protocol address (ip) geolocation bibliography. `http://www.caida.org/projects/cybersecurity/geolocation/bib/`, 2018.

[17] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*, pages 18–20, 2020.

[18] Yassine Chahid, Mohamed Benabdellah, and Abdelmalek Azizi. Internet of things security. In *Wireless Technologies, Embedded and Intelligent Systems (WITS), 2017 International Conference on*, pages 1–6. IEEE, 2017.

[19] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*, pages 378–387. IEEE Computer Society, 2005.

[20] Yue Chen, Y. Zhang, Z. Wang, and Tao Wei. Downgrade attack on trustzone. *ArXiv*, abs/1707.05082, 2017.

[21] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, 2017.

[22] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, 2017.

[23] EEMBC Embedded Microprocessor Benchmark Consortium. Cpu benchmark – mcu benchmark – coremark.

[24] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy. `https://mitmproxy.org/`, 2010–. [Version 4.0].

[25] Espressif Systems (Shanghai) PTE LTD. Esp-idf programming guide. `https://docs.espressif.com/projects/esp-idf/en/latest/`, 2018.

[26] William Feller. *An Introduction to Probability Theory and Its Applications*. Number v. 2 in An Introduction to Probability Theory and Its Applications. Wiley, 1950.

[27] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets. *International Symposium on Engineering Secure Software and Systems*, pages 155–172, 2016.

[28] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660, 2013.

[29] Pengfei Guo, Yingjian Yan, Chunsheng Zhu, and Junjie Wang. Research on arm trustzone and understanding the security vulnerability in its cache architecture. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*, pages 200–213. Springer, 2020.

[30] S.M. Hejazi, C. Talhi, and M. Debbabi. Extraction of forensically sensitive information from windows physical memory. *Digital Investigation*, 6:S121–S131, 2009. The Proceedings of the Ninth Annual DFRWS Conference.

[31] Antonio Ken Iannillo and Radu State. A proposal for security assessment of trustzone-m based software. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 126–127. IEEE, 2019.

[32] Sérgio Akira Ito, Luigi Carro, and Ricardo Pezzuol Jacobi. Making java work for microcontroller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001.

[33] Junyoung Jung, Jinsung Cho, and Ben Lee. A secure platform for iot devices based on arm platform security architecture. In *2020 14th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–4. IEEE, 2020.

[34] Uri Kanonov and Avishai Wool. Secure containers in android: the samsung knox case study. In *The 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–12, 2016.

[35] Kismet. Kismet: A wireless network detector, sniffer, and intrusion detection system. `https://www.kismetwireless.net/`, 2018.

[36] Nikolaos Koutroumpouchos, C. Ntantogian, and Christos Xenakis. Building trust for smart connected devices: The challenges and pitfalls of trustzone. *Sensors (Basel, Switzerland)*, 21, 2021.

[37] J Sathish Kumar and Dhiren R Patel. A survey on internet of things: Security and privacy issues. *Int. J. Comput. Appl.*, 90(11), 2014.

[38] Azeria Labs. Arm heap exploitation.

[39] Azeria Labs. Return oriented programming (arm32).

[40] Stephen S. Lim, Theo Vos, Abraham D. Flaxman, Goodarz Danaei, Kenji Shibuya, Heather Adair-Rohani, Markus Amann, H. Ross Anderson, Kathryn G. Andrews, Martin Aryee, Charles Atkinson, Loraine J. Bacchus, Adil N. Bahalim, Kalpana Balakrishnan, John Balmes, Suzanne Barker-Collo, Amanda Baxter, Michelle L. Bell, Jed D. Blore, Fiona Blyth, Carissa Bonner, Guilherme Borges, Rupert Bourne, Michel Boussinesq, Michael Brauer, Peter Brooks, Nigel G. Bruce, Bert Brunekreef, Claire Bryan-Hancock, Chiara Bucello, Rachelle Buchbinder, Fiona Bull, Richard T. Burnett, Tim E. Byers, Bianca Calabria, Jonathan Carapetis, Emily Carnahan, Zoe Chafe, Fiona Charlson, Honglei Chen, Jian Shen Chen, Andrew Tai Ann Cheng, Jennifer Christine Child, Aaron Cohen, K. Ellicott Colson, Benjamin C. Cowie, Sarah Darby, Susan Darling, Adrian Davis, Louisa Degenhardt, Frank Dentener, Don C. Des Jarlais, Karen Devries, Mukesh Dherani, Eric L. Ding, E. Ray Dorsey, Tim Driscoll, Karen Edmond, Suad Eltahir Ali, Rebecca E. Engell, Patricia J. Erwin, Saman Fahimi, Gail Falder, Farshad Farzadfar, Alize Ferrari, Mariel M. Finucane, Seth Flaxman, Francis Gerry R Fowkes, Greg Freedman, Michael K. Freeman, Emmanuela Gakidou, Santu Ghosh, Edward Giovannucci, Gerhard Gmel, Kathryn Graham, Rebecca Grainger, Bridget Grant, David Gunnell, Hialy R. Gutierrez, Wayne Hall, Hans W. Hoek, Anthony Hogan, H. Dean Hosgood, Damian Hoy, Howard Hu, Bryan J. Hubbell, Sally J. Hutchings, Sydney E. Ibeanusi, Gemma

L. Jacklyn, Rashmi Jasrasaria, Jost B. Jonas, Haidong Kan, John A. Kanis, Nicholas Kassebaum, Norito Kawakami, Young Ho Khang, Shahab Khatibzadeh, Jon Paul Khoo, Cindy Kok, Francine Laden, Ratilal Lalloo, Qing Lan, Tim Lathlean, Janet L. Leasher, James Leigh, Yang Li, John Kent Lin, Steven E. Lipshultz, Stephanie London, Rafael Lozano, Yuan Lu, Joelle Mak, Reza Malekzadeh, Leslie Mallinger, Wagner Marcenes, Lyn March, Robin Marks, Randall Martin, Paul McGale, John McGrath, Sumi Mehta, George A. Mensah, Tony R. Merriman, Renata Micha, Catherine Michaud, Vinod Mishra, Khayriyyah Mohd Hanafiah, Ali A. Mokdad, Lidia Morawska, Dariush Mozaffarian, Tasha Murphy, Mohsen Naghavi, Bruce Neal, Paul K. Nelson, Joan Miquel Nolla, Rosana Norman, Casey Olives, Saad B. Omer, Jessica Orchard, Richard Osborne, Bart Ostro, Andrew Page, Kiran D. Pandey, Charles D H Parry, Erin Passmore, Jayadeep Patra, Neil Pearce, Pamela M. Pelizzari, Max Petzold, Michael R. Phillips, Dan Pope, C. Arden Pope, John Powles, Mayuree Rao, Homie Razavi, Eva A. Rehfuess, Jürgen T. Rehm, Beate Ritz, Frederick P. Rivara, Thomas Roberts, Carolyn Robinson, Jose A. Rodriguez-Portales, Isabelle Romieu, Robin Room, Lisa C. Rosenfeld, Ananya Roy, Lesley Rushton, Joshua A. Salomon, Uchechukwu Sampson, Lidia Sanchez-Riera, Ella Sanman, Amir Sapkota, Soraya Seedat, Peilin Shi, Kevin Shield, Rupak Shivakoti, Gitanjali M. Singh, David A. Sleet, Emma Smith, Kirk R. Smith, Nicolas J C Stapelberg, Kyle Steenland, Heidi Stöckl, Lars Jacob Stovner, Kurt Straif, Lahn Straney, George D. Thurston, Jimmy H. Tran, Rita Van Dingenen, Aaron Van Donkelaar, J. Lennert Veerman, Lakshmi Vijayakumar, Robert Weintraub, Myrna M. Weissman, Richard A. White, Harvey Whiteford, Steven T. Wiersma, James D. Wilkinson, Hywel C. Williams, Warwick Williams, Nicholas Wilson, Anthony D. Woolf, Paul Yip, Jan M. Zielinski, Alan D. Lopez, Christopher J L Murray, and Majid Ezzati. A comparative risk assessment of burden of disease and injury attributable to 67 risk factors and risk factor clusters in 21 regions, 1990-2010: A systematic analysis for the global burden of disease study 2010. *Lancet*, 380(9859):2224–2260, 12 2012.

[41] Zhen Ling, Kaizheng Liu, Yiling Xu, Yier Jin, and Xinwen Fu. An end-to-end view of iot security and privacy. In *Proceedings of the 60th IEEE Global Communications Conference (Globecom)*, Singapore, December 2017.

[42] Zhen Ling, Junzhou Luo, Yiling Xu, Chao Gao, Kui Wu, and Xinwen Fu. Security vulnerabilities of internet of things: A case study of the smart plug system. *IEEE Internet Things J. (Iot-J)*, 2017.

[43] Liang Liu, Jun Ma, Cuiping Zhang, Ting Chong, Haifeng Zhang, and Yuanyi Dong. Security software system design and implementation for microcontrollers based on trustzone. *DEStech Transactions on Computer Science and Engineering*, (cisnrc), 2019.

[44] Sergey Lyubka. The two-dollar secure iot solution: Mongoose OS + ESP8266 + ATECC508 + AWS IoT. `https://mongoose-os.com/blog/mongoose-esp8266-atecc508-aws/`, December 2016.

[45] Carlo Maria Medaglia and Alexandru Serbanati. An overview of privacy and security issues in the internet of things. In *The Internet of Things*, pages 389–395. Springer, 2010.

[46] Microchip. Atmel start.

[47] Microchip. Atmel start.

[48] Microchip. Saml11 xplained pro evaluation kit.

[49] Microchip Technology Inc. Saml11 xplained pro evaluation kit. `http://www.microchip.com/DevelopmentTools/ProductDetails/dm320205`, 2018.

[50] Microsoft. Nx bits - microsoft wiki - fandom.

[51] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Netw.*, 10(7):1497–1516, 2012.

[52] Abhinav Mohanty, Islam Obaidat, Fadi Yilmaz, and Meera Sridhar. Control-hijacking vulnerabilities in iot firmware: A brief survey. In *The 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 2018.

[53] Nuvoton. Numicro m2351 series – a trustzone empowered microcontroller series focusing on iot security. Last accessed 2021-9-30.

[54] NXP. Lpc55s69-evk: Lpcxpresso55s69 development board. Last accessed 2021-9-30.

[55] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 259–284. Springer, 2017.

[56] Colin O'Flynn and Alex Dewar. On-device power analysis across hardware security domains. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 126–153, 2019.

[57] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, 2013.

[58] Bryan Pearson, Lan Luo, Yue Zhang, Rajib Dey, Zhen Ling, Mostafa Bassiouni, and Xinwen Fu. On misconception of hardware and cost in iot security and privacy. In *ICC 2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019.

[59] PlanetLab. Planetlab: An open platform for developing, deploying, and accessing planetary-scale services. `https://www.planet-lab.org/`, 2017.

[60] C Arden Pope III and Douglas W Dockery. Health effects of fine particulate air pollution: Lines that connect. *J. Air & Waste Manag. Assoc.*, 56(6):709–742, 2006.

[61] Heather Quinn. Microcontroller benchmark codes for radiation testing.

[62] Nguyen Anh Quynh. The ultimate disassembly framework – capstone – the ultimate disassembler.

[63] Dan Rosenberg. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat conference*, page 26, 2014.

[64] Rust. Rust: Embedded devices, 2020.

[65] Nordic Semiconductor. nrf5340. Last accessed 2021-9-30.

[66] Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. Security, privacy and trust in internet of things: The road ahead. *Comput. Netw.*, 76:146–164, 2015.

[67] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 574–588. IEEE Computer Society, 2013.

[68] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 1275–1295. IEEE, May 2019.

[69] Dug Song. Dsniff. `https://www.monkey.org/~dugsong/dsniff/`.

[70] ST. Stm32l562e-dk discovery kit. Last accessed 2021-9-30.

[71] STMicroelectronics. Stm32 ide.

[72] Hui Suo, Jiafu Wan, Caifeng Zou, and Jianqi Liu. Security in the internet of things: A review. In *Computer Science and Electronics Engineering (ICCSEE), 2012 international conference on*, volume 3, pages 648–651. IEEE, 2012.

[73] swiat. On the effectiveness of dep and aslr, 2010.

[74] Texas Instruments Incorporated. Cc3220 simplelink wi-fi and iot, single-chip wireless mcu solution. http://www.ti.com/product/CC3220?keyMatch=cc3220sf&tisearch=Search-EN-Everything, 2018.

[75] Wikipedia. Stars and bars (combinatorics).

[76] Wikipedia. Air quality index - wikipedia. https://en.wikipedia.org/wiki/Air_quality_index, 2018.

[77] Wikiwand. Address space layout randomization.

[78] Wireshark. Wireshark oui lookup tool. https://www.wireshark.org/tools/oui-lookup.html.

[79] Wireshark. Wireshark. https://www.wireshark.org/, 2018.

[80] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems*, pages 260–269, Oct 2003.

[81] Joseph Yiu. Chapter 2 - getting started with cortex-m programming. In Joseph Yiu, editor, *Definitive Guide to Arm® Cortex®-M23 and Cortex-M33 Processors*, pages 19–51. Newnes, 2021.