SIC^2: Securing Microcontroller Based IoT Devices with Low-cost Crypto Coprocessors

Bryan Pearson*, Cliff Zou*, Yue Zhang†§, Zhen Ling‡ , Xinwen Fu*†
*Dept. of Computer Science, University of Central Florida, USA; bpearson@knights.ucf.edu, czou@cs.ucf.edu
‡School of Computer Science and Engineering, Southeast University; zhenling@seu.edu.cn
†Dept. of Computer Science, University of Massachusetts Lowell, MA, USA; xinwen_fu@uml.edu
§Dept. of Computer Science, Jinan University; zyueinfosec@gmail.com

***Abstract*—In this paper, we explore the use of microcontrollers (MCUs) and crypto coprocessors to secure IoT applications, and show how developers may implement a low-cost platform that provides protects private keys against software attacks. We first demonstrate the plausibility of format string attacks on the ESP32, a popular MCU from Espressif that uses the Harvard architecture. The format string attacks can be used to remotely steal private keys hard-coded in the firmware. We then 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. We have developed a prototype and performed extensive experiments to show that the ATECC608A crypto chip may significantly reduce the TLS handshake time by as much as 82% with the remote server, and it may lower the total energy consumption of the system by up to 70%. Our results indicate that securing IoT with crypto coprocessors is a practicable solution for low-cost MCU based IoT devices.**

I. Introduction

The popularity of Internet of Things (IoT) has raised grave security and privacy concerns. There is a broad attack surface against IoT, including vulnerabilities and issues in hardware, firmware/operating system, application software, networking and data. For example, hackers can force autonomous vehicles to crash [20] and may also steal credentials from consumer and medical products [2]. Botnets such as Mirai [1] and Reaper [6] exposed vulnerable networks and compromised millions of devices.

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 [25]. Particularly, ESP32 has abundant hardware security features including secure boot and flash encryption [27]. However, we have found potential threats against ESP32 by using two practical software attacks, named Same Subroutine Attack and Cross Subroutine Attack. In Same Subroutine Attack, the vulnerable instruction (i.e., `printf(.)` or `sprintf(.)`) and the victim's secret data are co-located in the same subroutine, and an attacker may steal this secret data via the attack. However, Cross Subroutine Attack is more powerful, where an attacker may extract sensitive information even if the vulnerable function and the victim code fragment are located in different subroutines. We demonstrate our attacks with a proof of concept web server, showing that an attacker may deploy the attacks remotely through the Internet.

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

To defeat software attacks, 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 or **S**ecuring **I**oT devices with **C**ryptographic **C**oprocessors, that stores private keys inside of a cryptographic coprocessor. Our SIC^2 protects keys from malicious personnel within the semiconductor manufacturing line as well as cyber attacks [13]. We implement a proof of concept by pairing the ESP32 with the ATECC608A [8] crypto coprocessor ($0.53 at Microchip), which can provide mutual authentication, encryption and integrity to a network.

Our major contributions can be summarized as follows:

1) We show that popular MCUs such as ESP32 can be compromised by multiple software attacks. Private keys can be leaked remotely.
2) We propose SIC^2, a systematic solution for manufactures 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 ECC608A.
3) 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 Amazon EC2 can reduce the overall TLS handshake time by 82% and energy consumption by up to 70%.

The rest of this paper is organized as follows. In Section II, we provide the background of the ESP32 MCU and its processor. In Section III, we present novel format string attacks against the ESP32 which compromise private keys stored on the device. In Section IV, we introduce SIC^2

978-1-7281-9074-7/20/$31.00 ©2020 IEEE
DOI 10.1109/ICPADS51040.2020.00057

and how manufacturers may securely write private keys into cryptographic coprocessors. A proof of concept of SIC^2 is discussed in Section V which combines the ESP32 with the ECC608A. We evaluate the ECC608 overhead and network performance in Section VI. Section VII discusses some related works, and Section VIII concludes the paper.

II. Background

In this section, we discuss the system design of the ESP32 and the architecture of the Xtensa processor, which is used by the ESP32.

A. ESP32 System Design

The ESP32 is a popular IoT MCU [25]. It supports WiFi, Bluetooth, and Bluetooth Low Energy (BLE) capabilities for a variety of IoT applications. It exposes Universal Asynchronous Receiver/Transmitter (UART) and Joint Test Action Group (JTAG) external debugging ports. UART communication allows users to monitor console output, upload new firmware to the chip, dump flash contents, and modify security settings on the chip. JTAG allows for complete debugging of the ESP32, including reading and modifying the entire firmware, bootloader contents, CPU registers and SRAM contents on the ESP32. Espressif has ported GDB to recognize the Xtensa architecture.

The ESP32 contains a 1kB block of secure eFuse memory. These memory contents are accessible only to the hardware, and once an eFuse value is set, it is irreversible. The eFuse memory controls access to the communication and debugging ports. Another feature of the eFuse is the secure storage of a 256-bit flash encryption key and a 256-bit secure boot key. With flash encryption enabled, the ESP32 can use the flash encryption key with AES cipher block chaining (CBC) mode to decrypt data and instructions before being processed by the CPU. With secure boot, the ROM will calculate an AES-based SHA digest to validate the integrity of the bootloader, which in turn may validate the firmware.

B. Xtensa Processor Architecture

In this section, we discuss architecture details about Xtensa LX6, a 32-bit microprocessor from Tensilica [11]. ESP32 contains 2 Xtensa processors. We first provide some basic information about the Xtensa processor. We then discuss details about the register file and how the ESP32 can access register contents at runtime.

1) Architecture Details: Xtensa implements a modified Harvard architecture [24], with the main memory separated into instruction SRAM and data SRAM. The processor is programmable to allow manufacturers to modify instructions, the register file size, cache size, memory width, and make various other enhancements. Tensilica provides tools for mapping any configuration to the physical hardware.

2) The Register Window: The ESP32's register file contains 64 general-purpose registers. The Xtensa architecture implements a feature called **register window** and allows a subroutine to only access to 16 general-purpose registers at a time. In the register file, registers are labeled AR0, AR1, AR2, etc. The register window allocates a contiguous block of registers within the register file; for example, a subroutine may only have access to AR16, AR17, AR18, and so forth, up to AR31. When a subroutine *Sub1* calls some other subroutine *Sub2*, the register window "increments" its position in the register file, meaning new registers become available while old registers become inaccessible. The register window can increment by either 4, 8, or 12 registers. When *Sub2* returns, the register window reverts or "decrements" to the original position, allowing *Sub1* to access the same registers.

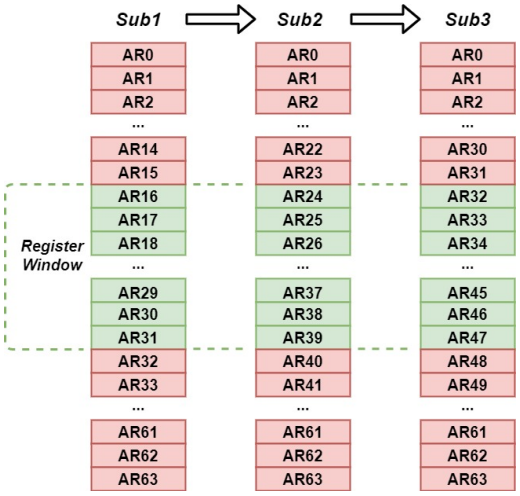

Fig. 1: Overview of the ESP32 register file. A subroutine only has access to the registers contained within the register window.

Figure 1 provides further details. Consider *Sub1*, whose register window is defined for the range AR16 to AR31. Now when *Sub1* calls *Sub2*, the register window increments by 8 registers such that *Sub2* can now access registers in the range AR24 to AR39, while registers AR16 to AR23 are no longer accessible. Similarly, when *Sub2* calls *Sub3*, the register window increments by 8 registers again such that *Sub3* can access registers in the range AR32 to AR47. On each return—from *Sub3* to *Sub2* and from *Sub2* to *Sub1*—the register window will decrement by 8 registers and allow each respective subroutine to recover the contents of its registers.

In the case where a register window attempts to allocate registers that already belong to a parent subroutine *SubP*, the CPU will initiate a *window overflow exception*. In this scenario, the CPU will dump the contents of registers into memory and allow the new subroutine to access those registers. When the program returns back to *SubP*, it will restore the register contents from memory back into the registers. In this way, register contents are never lost, even when the registers themselves must be shared among subroutines.

III. Novel Attacks against ESP32

In this section, we present two novel format string attacks on the ESP32, named the **Same Subroutine Attack** and the **Cross Subroutine Attack**. We begin by explaining the threat model of these attacks. Then we discuss implementation details, before showing a proof of concept for a remote format string attack. Additional proof-of-concepts, including

a “Serial-to-TCP” attack, are discussed in the technical report of this paper in Appendix D [21].

A. Threat Model

In the following attacks, we assume that the ESP32 may expose some kind of communication channel to the user, such as HTTP or MQTT. We also assume that the ESP32 stores some secret data in its firmware. The adversary may be local or remote. A local adversary can physically access the device and view serial output directly. In the local attack, we assume that the adversary can access the UART interface on the ESP32. This assumption is reasonable because the UART interface can be easily accessed by a micro USB cable, and the ESP32's UART interface cannot be disabled. A remote adversary can read the data transmitted by the ESP32 over the communication channel. This assumption is reasonable when the communication channel fails to authenticate the user, which is common in IoT [17]. We show that the proposed attacks can work both locally and remotely. Finally, we assume that the firmware contains some programming flaw, which is reasonable due to the abundance of software vulnerability types in C [30] [31] [29] [23].

B. Attack Overview

Format string vulnerabilities [29] arise when formatting functions fail to validate a user's input format. An example of such a function is *printf()*, which accepts format string characters as its input. Typically, if the program were to execute an instruction such as *printf("%s", name)*, it would simply print the contents of *name*. However, if *name* is not provided to the function, the program will print the contents of a different memory location, which may leak sensitive data. Every format string character passed to the format function will fetch the value of the next consecutive memory address and cast it accordingly. On the ESP32, which has a 4-byte address width, this means every format string character fetches the next 4 bytes in memory.

Based on our experiments on the ESP32, we found that when no input parameters are provided to *printf()*, it will begin by fetching the last five registers in the subroutine's register window. Afterwards, it will fetch the value at the stack pointer (SP), then the value at SP + 4, then SP + 8, and so forth. This means that on the ESP32, at least 6 format string characters are required to begin accessing memory contents. In this way, the format string attack may be used on the ESP32 to leak arbitrary data from memory.

C. Format String Attacks

*1) **Same Subroutine Attack**:* In the **Same Subroutine Attack**, the format string instruction and the private data exist within the same subroutine. We begin by discussing the setup of this subroutine. We then describe the details of the registers and memory. Finally, we show how an adversary may exploit this program and obtain the private data.

Listing 1: Same Subroutine Attack psuedocode.

```
void app_main() {
    char tmp[16] = "PRIVATE KEY";
    char* params = malloc(128);
    accept_user_input(&params);
    printf(params); }
```

As shown by Listing 1, the program defines a local variable called *tmp* in a subroutine called *app_main*. In our example, *tmp* is a 16-byte char array set to the string "PRIVATE KEY". *printf()* will print some arbitrary input that is provided by the user in *accept_user_input()*.

We used JTAG debugging on the ESP32 to determine details about this program. Communication with JTAG requires the addition of OpenOCD, an open-source software project that can communicate with the JTAG interface [22]. We attached a GDB client to the OpenOCD session in order to debug our application.

From JTAG debugging, we determined the following details. The stack pointer address of *app_main* is 0x3ffb4ee0. On the ESP32, local char arrays are always defined starting at the stack pointer address, so *tmp* is defined from 0x3ffb4ee0 to 0x3ffb4eef. The register window in the subroutine is defined between AR16 and AR31. The contents of the last five registers in the register window (namely, AR27 to AR31) are 0x8001f880, 0x6ff1ff8, 0x0, 0x3ffaffe0, and 0x3ffb6840.

Fig. 2: Overview of the Same Subroutine Attack. The arrows show which addresses *printf()* will access.

To perform the attack, the user must provide the following format string as input to *printf()*: **"%p %p %p %p %p %x %x %x %x"**. The first five characters print the contents of AR27, AR28, AR29, AR30, and AR31, while the last four characters print the contents of *tmp*. The attack behavior is illustrated in Figure 2. The output to UART is shown below:

```
0x8001f880 0x6ff1ff8 0x0 0x3ffaffe0
0x3ffb6840 56495250 20455441 59454b
0
```

As shown above, the first five values correspond to the register contents of AR27 through AR31. The next 16 bytes correspond to the stack contents, beginning with the stack pointer. Recall that *tmp* is written at the stack pointer address. Since the bus architecture of the ESP32 is little-endian, the bytes must be reversed to recover the original data. For example, the value 56495250 must be changed to 50524956. After doing this for all values, the user can obtain the desired value 505249564154452045 4b59 0. A hex-to-ascii converter shall reveal the contents of this data to be "PRIVATE KEY".

*2) **Cross Subroutine Attack**:* In the **Cross Subroutine Attack**, the format string instruction and the private data are located in different subroutines. This attack is much more powerful than the Same Subroutine Attack, since it can steal data from any previous subroutine in the call stack.

Again, we begin by discussing the setup requirements of the program, followed by the details of the program including memory and register contents. We conclude by showing the exploit and how the private data may be recovered.

Listing 2 shows that the format string function and the private data are located in different subroutines. we have a local variable *tmp* defined in *app_main*, but we also have two new subroutines, *sub1* and *sub2*. In the expected program flow, *app_main* calls *sub1*, which calls *sub2*, which calls the vulnerable *printf* function. The attack will leverage the behavior of the window overflow exception in the ESP32, where register contents are dumped to memory when the program transfers control to a new subroutine. Namely, the address of *app_main*'s stack pointer will dump to memory when the program executes *sub2*, and the attacker can use the character "%s" at this location to recover the stack pointer address, cast it as a string, and print its value.

Listing 2: Cross Subroutine Attack psuedocode.

```
void sub2(char* x) { printf(x); }
void sub1(){
    char* params = malloc(128);
    accept_user_input(&params);
    sub2(params); }
void app_main() {
    char tmp[16] = "PRIVATE KEY";
    sub1(); }
```

Again, we used JTAG to debug the program and discovered the following information. First, the stack pointer of *app_main*, *sub1*, and *sub2* are 0x3ffb4ee0, 0x3ffb4ec0, and 0x3ffb4ea0, respectively. As *tmp* is a local buffer defined in *app_main*, *tmp*'s address is also 0x3ffb4ee0. The ESP32's application startup sequence makes several subroutine calls prior to reaching *app_main*, and our experiments show that the register file has already been exhausted by the time the program reaches *app_main*. The call to *sub1* and *sub2* both shift the register window by 8 registers. Therefore, due to the window overflow exception, 8 registers must be dumped into memory on both calls. The register window for *app_main* is defined from AR16 to AR31. For *sub1*, it is defined from AR24 to AR39. And for *sub2*, it is defined from AR32 to AR47. When jumping to *sub1*, registers AR16 through AR23 are saved to memory; when jumping to *sub2*, registers AR24 through AR31 are saved to memory. The stack pointer address is always stored in the second register of the register window; in the case of *app_main*, AR17 contains the stack pointer value 0x3ffb4ee0. Our experiments revealed that the second register is always dumped to the memory location that is 12 bytes behind the new stack pointer. In particular, this means that when the program reaches *sub1*, the stack pointer address of *app_main* is saved to 0x3ffb4eb4, exactly 12 bytes behind *sub1*'s stack pointer.

If *printf* can be manipulated to point to 0x3ffb4eb4, the character "%s" will cast this address as a char pointer and print its value accordingly. This will cause the contents of *tmp* to be leaked. However, as noted above, the stack pointer address of *sub1* is 0x3ffb4ec0, while the stack pointer address of *sub2* is 0x3ffb4ea0. This means that the format string attack cannot be used in *sub1*, because the format string pointer can only be moved forward in memory starting from the stack pointer; it cannot be moved backward. Fortunately, *sub2*'s stack pointer precedes 0x3ffb4eb4, so the format string attack is feasible in this subroutine.

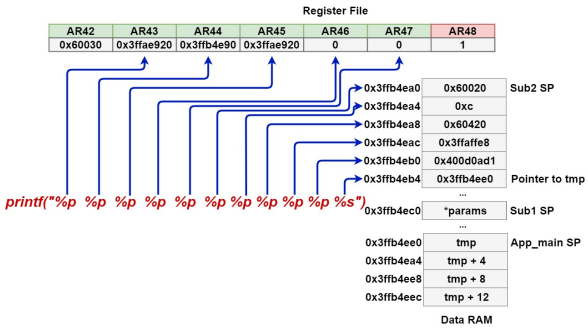

Fig. 3: Overview of the Cross Subroutine Attack.

To perform the attack, the user must provide the following format string as input to *printf()*: **"%p %p %p %p %p %p %p %p %p %p %s"**. Note that *sub2*'s stack pointer precedes 0x3ffb4eb4 by 20 bytes. The first 5 characters print the registers AR43 through AR47, while the next 5 print the first 20 bytes of the stack frame, specifically the contents in the range 0x3ffb4ea0 to 0x3ffb4eb3. The final character, "%s", will cast 0x3ffb4eb4 as a char pointer and print its value. The attack behavior is illustrated in Figure 3. The output of this attack is shown below:

```
0x3ffae920 0x3ffb4e90 0x3ffae920
0x0 0x0 0x60020 0xc 0x60420
0x3ffaffe8 0x400d0ad1 PRIVATE_KEY
```

Unlike the Same Subroutine Attack, this payload prints the memory contents verbatim as long as it can be cast as a string. This leads to faster detection of sensitive data while it may also lead to unexpected errors if a string does not contain the null terminator, which will cause the program to read memory contents indefinitely and potentially crash.

*3) **Attack Limitations**:* Both the Same Subroutine Attack and the Cross Subroutine Attack have limitations. Most notably, the attacks assume that the private key is stored as a local variable so that the adversary can read its contents from the stack. Not all programs may behave this way; for example, the private key may be stored as a global variable within the data section. The data section is located *before* the stack in the ESP32's address map. Therefore, the format sting attacks described above cannot access a private key which is stored as a global variable, because those attacks can only read memory contents that are located after the stack pointer.

D. Proof of Concept Remote Format String Attack

We now demonstrate how format string attacks can be performed remotely to steal an IoT device's secrets. An application that exposes a communication channel such as a web server interface may fall victim to the format string attack. We have written two vulnerable programs using the Arduino IDE, an alternative to the ESP-IDF development platform provided by Espressif. To serve as a web server,

ESP32 uses the WebServer library of the Arduino platform, which allows a web server to process HTTP requests from the client and send responses back. The ESP32 application contains a format string vulnerability based on the *sprintf()* function in C, which sends the formatted output to a string rather than stdout. The expected syntax for this instruction is *sprintf(buf, "%s", param)*, where *buf* is a string and *param* is formatted as a string before being sent to *buf*. However, the instruction *sprintf(buf, param)* is vulnerable to the format string attack, because *param* is now treated as the format parameter and can lead to memory leakage if controlled by an attacker. If *buf* is passed remotely to an adversary, he can observe the output of the format string attack.

1) Remote Same Subroutine Attack: This attack exploits the Same Subroutine Attack through the *sprintf()* vulnerability to leak a private key on a web server. Listing 3 provides pseudocode, which starts a web server at port 80. The *handleReq()* subroutine is called whenever a user visits the root index of the server. This subroutine stores a key and parses any HTTP GET request sent from the client, calls *sprintf()* to format the request, and passes the formatted request back to the client in an HTTP response.

Listing 3: Remote Same Subroutine Attack psuedocode.

```
WebServer server(80);
void handleReq() {
    char key[32] = "THIS IS A PRIVATE KEY";
    char* res = malloc(128);
    char* param = server.arg(0).c_str();
    sprintf(res, param);
    server.send(200, "text/plain", res); }
void setup() {server.on("/", handleReq); }
void loop() { server.handleClient(); }
```

To send a payload, the adversary can use a web browser to send the following GET 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 the *res* buffer. The server will send the buffer back to the adversary to read in the HTTP response. The attacker can then derive the key from the payload by decoding to ascii.

2) Remote Cross Subroutine Attack: This attack exploits the Cross Subroutine Attack through the *sprintf()* vulnerability. Similar to the *printf()*-based Cross Subroutine Attack, the adversary will pass a format string containing the "%s" character to cast the private key as a string. The web server will pass this output to a buffer that is sent back to the adversary in an HTTP response. The full details of this attack are available in Appendix D of the technical report [21].

IV. SIC^2: Securing IoT with Crypto Coprocessors

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

A. Need of Crypto Co-processors

From our discussion in Section III, MCUs with secure boot can be compromised and leak cryptographic keys if these keys have no hardware protection. While the TrustZone technology has been integrated into Arm Cortex-M processors, denoted as TrustZone-M, it can be compromised too [18]. 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 a number of IoT development boards and solutions. Only Microsoft's Azure Sphere [19] and TI's CC3220 and CC3100MOD 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 Appendix A of the technical report for this paper [21].

B. 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 [13]. 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 4. 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

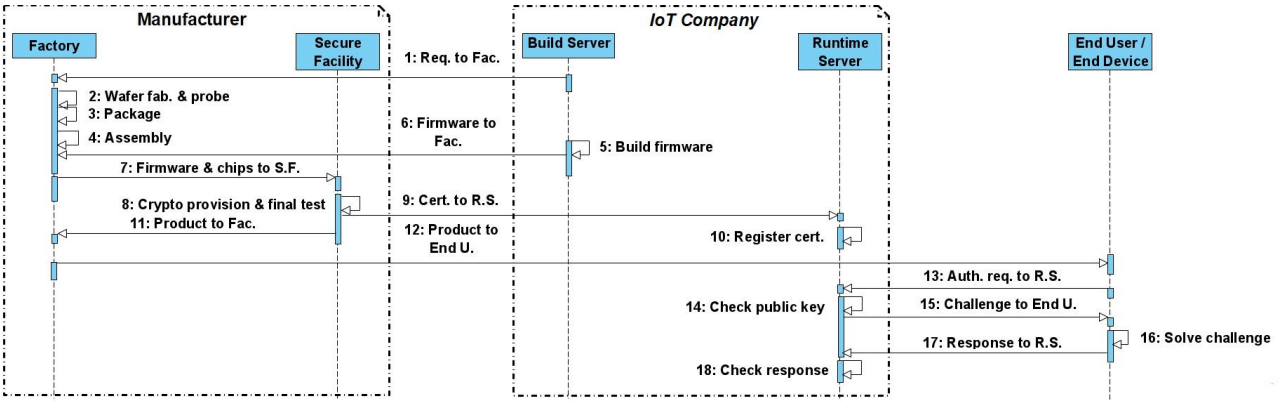

Fig. 4: Secure key provisioning framework.

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**.

1) Manufacturing Phase (Steps 1-4): 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.

2) Key Provisioning Phase (Steps 5-10): 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 a 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.

3) Device Authentication Phase (Steps 11-18): 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 orders the product from the factory, 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.

C. Security Analysis

With our defense enabled, all software attacks in this paper will fail because private keys will no longer be stored in the firmware. 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.

V. 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 [12]. A prototype of our defense can be found in Figure 5. This project was written in ESP-IDF version 4.0 and is publicly available on Github *.

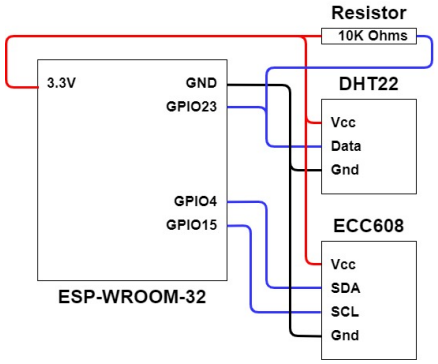

Fig. 5: Schematic of ESP32 with ECC608 and DHT22.

A. 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 illustrates the pairing of an ESP32-based development board with the ECC608 on a socket adapter.

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 attaining 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 [10]. CryptoAuthLib allows an MCU to communicate with the ECC608 via the I^2C 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^2C traffic by encrypting data with the secret value. This results in a safer I^2C channel.

To achieve network communication, we use *MbedTLS* [7], 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 case the server provides an ECC-based certificate. 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.

B. 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 provide implementation details with the DHT22 in the technical report [21].

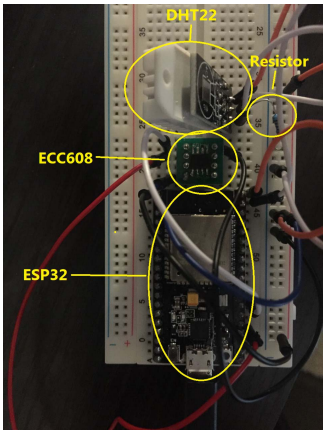

Fig. 6: ESP32 with the ECC608 and DHT22 on a breadboard.

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 illustrates our hardware setup on a breadboard.

We have used Atmel Crypto Evaluation Studio (ACES) to set the configuration parameters. ACES is a programming software that can communicate with the ECC608 via an external programmer, such as the ATSAMD21 board [9].

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" (a file executed by CMake which describes the build instructions for a project) to the root directory. The file includes the source and header files of this library. The library contains a hardware

*Available at https://github.com/PBearson/ESP32-With-ECC608.

abstraction layer that specifies communication settings with many devices including the ESP32 over I^2C; this setting is included as a compile option in "CMakeLists.txt".

In addition, we have developed an app that connects with a remote server via Message Queueing Telemtry Transport (MQTT) over TLS. Our app integrates the CryptoAuthLib and Espressif's 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. *atcab_sign* and *atcab_verify_extern* will provide the required operations. In ECDH, we overload the public key generation and shared key generation functions. *atcab_genkey* will generate a key in the temporary key slot, while *atcab_ecdh_tempkey* will establish the shared key.

C. 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, as discussed in Section IV.

The details for enabling the ESP32's security features are as follows. 1) **Flash Encryption (FE)**: The programmer should use Espressif's build framework to compile the bootloader to support FE. During the boot sequence, the bootloader will detect FE is supported, and the hardware will generate a key to store in the eFuse. Then, the chip will encrypt the complete flash contents. 2) **Secure Boot (SB)**: The programmer should compile the bootloader to support SB. On first boot, the ROM will generate a SB key to store in the eFuse. Then, the ROM generates an AES-based SHA digest over the bootloader using the SB 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.

VI. Evaluation

In this section, we discuss the area overhead of the ECC608 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, please refer to Appendix C of the technical report [21].

A. ECC608 Area Overhead

PCB size is an important factor when considering 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, 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.

B. AWS 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**, or simply AWS, is an IoT management cloud service. AWS can generate certificates for the end user that are signed by the Amazon Root CA. AWS 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 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.

The difference in the network connection between AWS and EC2 lies in their server certificates. During the TLS handshake, AWS will present a server certificate signed by a RSA private key, while EC2 is 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 and negotiate the session key, since ECC608 only supports ECC for public key cryptography. However, the ECC608 can be used to verify EC2's certificate and take advantage of the hardware acceleration.

C. 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 vs. 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 and an EC2 server, the latter of which uses an ECC-based certificate and can perform ECDH with our ESP32. Each benchmark was executed 100 times, and we recorded the average runtime.

Figure 7 shows the total handshake time when connecting to AWS, while Figure 8 measures the EC2 handshake time. Connecting to AWS does not impact the connection time

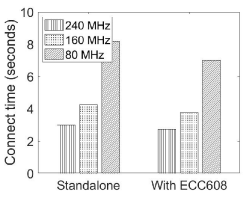

Fig. 7: AWS handshake time.

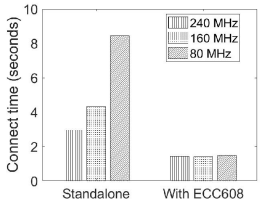

Fig. 8: EC2 handshake time.

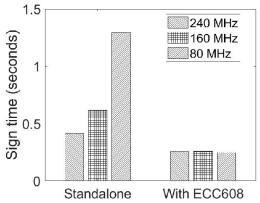

Fig. 9: ECDSA sig. gen. time.

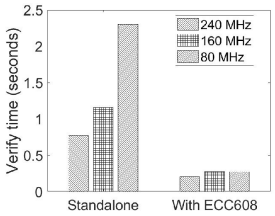

Fig. 10: ECDSA sig. verify time.

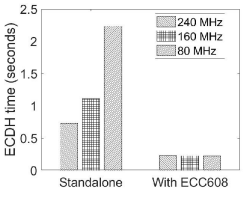

Fig. 11: ECDH time.

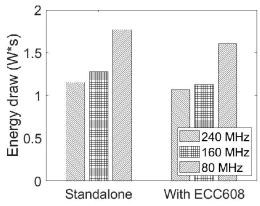

Fig. 12: AWS handshake energy draw.

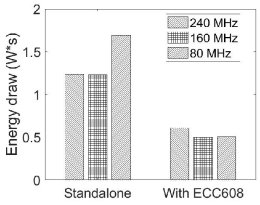

Fig. 13: EC2 handshake energy draw.

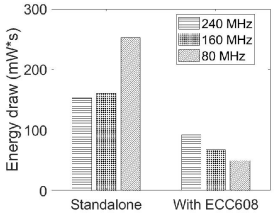

Fig. 14: ECDSA sig. gen. energy draw.

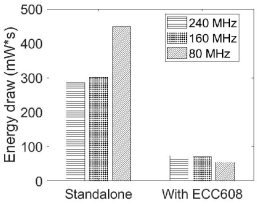

Fig. 15: ECDSA verify energy draw.

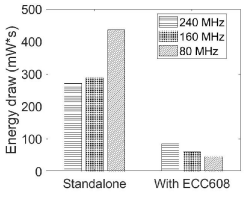

Fig. 16: ECDH energy draw.

so drastically, since the ECC608 can only use the signature generation function to prove ownership of its certificate. 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. It can be observed that these operations form the majority of computation, as the CPU clock speed has almost no impact on the handshake performance when the ECC608 is in use.

Figures 9 and 10 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 to verify a signature. By comparison, the ECC608 can consistently perform signature generation and verification in about 0.25 seconds.

Finally, we measure the time delay of ECDH which establishes the session key among the client and the server. Figure 11 shows these results. 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.

D. Energy Consumption

To complement our performance metrics, we have also measured the energy consumption of ESP32 when performing the TLS handshake, ECDSA, and ECDH operations. Figures 12, 13, 14, 15, and 16 showcase these measurements. Note that the ECC608 itself also contributes to the total energy consumption, since it draws power from the ESP32. Despite this, our results indicate the ECC608 reduces energy consumption of the whole system. When using the crypto chip, the EC2 handshake requires only 0.6 Watt-seconds of energy, whereas a standalone ESP32 may require up to 1.7 Watt-seconds. At 80 MHz, the crypto chip reduces power usage 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 megawatt-seconds, while the standalone ESP32 will draw 252 megawatt-seconds under this operation. These results are consistent with the signature verification and ECDH key exchange benchmarks.

VII. Related Work

In this section, we discuss software and hardware attacks that have targeted ESP32 and ESP8266 [26] in recent years.

Hardware Exploits. 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 [15]. A similar attack was performed on WIZ smart bulb [16]. Researchers also performed a voltage glitching attack on the ESP32 ROM with full security settings enabled, triggering a full readout of the security keys [14]. The latter attack cost several hundred dollars and could only be addressed with a hardware revision. [28]. By comparison, the crypto coprocessor discussed in this paper, the ATECC608A, employs several anti-tampering mechanisms to defend against such hardware exploits.

Software Exploits: The *Zero PMK Installation* vulnerability affects the EAP authentication framework [5]; attackers could force the Pairwise Master Key (PMK) to default to 0 and hijack a connection. 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 [3]. 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 and the Pairwise Suite Count value. A malicious AP can send an arbitrarily large AKM packet and trigger a crash [4]. Note that ESP-IDF version 3.3 and NONOS version 3.1 address all of the aforementioned vulnerabilities. Carel Van Rooyen and Philipp Promeuschel [24] 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. Our format string attacks differ from the above related works because our attacks are more general in nature (i.e., not specifically tied to any libraries), and the ESP32 provides no formal protection against format string attacks.

VIII. Conclusion

In this paper, we explore how low-cost 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. Software attacks are a major concern on IoT devices. We demonstrate two remote 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.

Acknowledgements

This research was supported in part by National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, and 2017YFB1003000, US National Science Foundation (NSF) Awards 1643835, 1931871 and 1915780, US Department of Energy (DOE) Award DE-EE0009152, National Natural Science Foundation of China (Grant Nos. U1736203, 61877029, 62022024, 61972088, 61532013), Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] 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, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In 26th USENIX Security Symposium (USENIX Security 17), Aug. 2017.

[2] C. Cimpanu. Hacker leaks passwords for more than 500,000 servers, routers, and iot devices. https://www.zdnet.com/, 2020.

[3] M. E. Garbelini. Esp32/esp8266 eap client crash (cve-2019-12586. https://matheus-garbelini.github.io/home/post/esp32-esp8266-eap-crash/, 2019.

[4] M. E. Garbelini. Esp8266 beacon frame crash (cve-2019-12588. https://matheus-garbelini.github.io/home/post/esp8266-beacon-frame-crash/, 2019.

[5] M. E. Garbelini. Zero pmk installation (cve-2019-12587. https://matheus-garbelini.github.io/home/post/zero-pmk-installation/, 2019.

[6] A. Greenberg. The reaper iot botnet has already infected a million networks. https://www.trendmicro.com/vinfo/pl/security/news/cybercrime-and-digital-threats/millions-of-networks-compromised-by-new-reaper-botnet, October 2015.

[7] A. Holdings. Arm mbed. https://tls.mbed.org/, 2020.

[8] M. T. Inc. Atecc608a. https://www.microchip.com/wwwproducts/en/ATECC608A, 2018.

[9] M. T. Inc. Atsamd21g18. https://www.microchip.com/wwwproducts/en/ATsamd21g18, 2020.

[10] M. T. Inc. Cryptoauthlib - microchip cryptoauthentication library. https://github.com/MicrochipTech/cryptoauthlib, 2020.

[11] T. Inc. Xtensa Instruction Set Architecture (ISA) Reference Manual.

[12] A. Industries. Dht22 temperature-humidity sensor + extras. https://www.adafruit.com/product/385.

[13] M. Korolov. What is a supply chain attack? why you should be wary of third-party providers. https://www.csoonline.com/article/3191947/, Jan 2019.

[14] LimitedResults. Pwn the esp32 forever: Flash encryption and sec. boot keys extration. https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/, 2019.

[15] LimitedResults. Pwn the lifx mini white. https://limitedresults.com/2019/01/pwn-the-lifx-mini-white/, 2019.

[16] LimitedResults. Pwn the wiz connected. https://limitedresults.com/2019/02/pwn-the-wiz-connected/, 2019.

[17] 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 Things J., 4(6):1899–1909, 2017.

[18] L. Luo, Y. Zhang, C. C. Zou, X. Shao, Z. Ling, and X. Fu. On runtime software security of trustzone-m based iot devices. In Proceedings of IEEE Global Communications Conference (GLOBECOM), 2020.

[19] Microsoft. What is azure sphere? https://docs.microsoft.com/en-us/azure-sphere/product-overview/what-is-azure-sphere, 03 2020.

[20] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. http://illmatics.com/Remote\%20Car\%20Hacking.pdf, August 2015.

[21] B. Pearson, C. Zou, Y. Zhang, Z. Ling, and X. Fu. sic^2: Securing microcontroller based iot devices with low-cost crypto coprocessors. https://bpearson.net/papers/ICPADS_2020.pdf, 2020.

[22] D. Rath. Open on-chip debugger. http://openocd.org/files/thesis.pdf.

[23] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., 15(1):2:1–2:34, 2012.

[24] C. V. Rooyen and P. Promeuschel. Exploitation: Arm & xtensa compared. http://nullcon.net/, 2018.

[25] E. Systems. Espressif achieves the 100-million target for iot chip shipments. https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments, Jan. 2018.

[26] E. Systems. Esp8266 overview. https://www.espressif.com/en/products/socs/esp8266/overview, 2019.

[27] E. Systems. Secure boot. https://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html, 2019.

[28] E. Systems. Security advisory concerning fault injection and efuse protections (cve-2019-17391). https://www.espressif.com/en/news/Security_Advisory_Concerning_Fault_Injection_and_eFuse_Protections, 2019.

[29] S. University. Format string vulnerability. http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf.

[30] ViperEye. Heap overflow: Vulnerability and heap internals explained. https://resources.infosecinstitute.com/heap-overflow-vulnerability-and-heap-internals-explained/, Jun. 2013.

[31] B. Watters. Stack-based buffer overflow attacks: Explained and examples. https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/, Feb. 2019.