SIC^2: **S**ecuring Microcontroller Based **I**oT Devices with Low-cost **C**rypto **C**oprocessors

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

***Abstract*—The popularity of Internet of Things (IoT) has raised grave security and privacy concerns. Software level attacks may compromise secret data stored in the firmware. In this paper, we explore the use of microcontrollers (MCUs) and crypto modules to secure IoT applications, and we show how developers may implement a low-cost platform that provides security to users and 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. We then present a framework termed SIC^2 (Securing IoT with Crypto Coprocessors), for secure key provisioning that protects end users' crypto chips from untrustworthy manufacturers. As a proof of concept, we pair the ESP32 with the low-cost ATECC608A cryptographic co-processor 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 that 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 [18] and may also steal credentials from consumer and medical products [2]. Botnets such as Mirai [1] and Reaper [7] 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 millions the two chips in January 2020 [23]. Particularly, ESP32 has abundant hardware security features including secure boot [25]. 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 function (i.e., `printf(.)`) is co-located with the victim code fragments in the same subroutine, and an attacker may steal sensitive information such as private keys 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. Our attacks significantly undermine the security of ESP32, and the principles may apply to other IoT chips.

To defeat software attacks, in this paper 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, **S**ecuring **I**oT devices with **C**ryptographic **C**oprocessors. Our SIC^2 protects keys from a potentially malicious manufacturer, who may want to steal those private keys. We implement a proof of concept by pairing the ESP32 with the ATECC608A [9] 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.
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, offering design and implement criterion for developers.

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 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.

A. ESP32 System Design

The ESP32 is a popular IoT MCU used by millions of consumers around the world [23]. The ESP32 contains a network stack that 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 arbitrary memory addresses, and modify security settings on the chip. JTAG allows for complete debugging of the ESP32, 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. This memory is secure because its 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 digest to validate the integrity of the bootloader, which in turn may validate the firmware and any other partitions.

B. Processor Architecture and Registers

In this section, we discuss architecture details about Xtensa LX6, a 32-bit microprocessor from Tensilica [12]. 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 [22], 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.

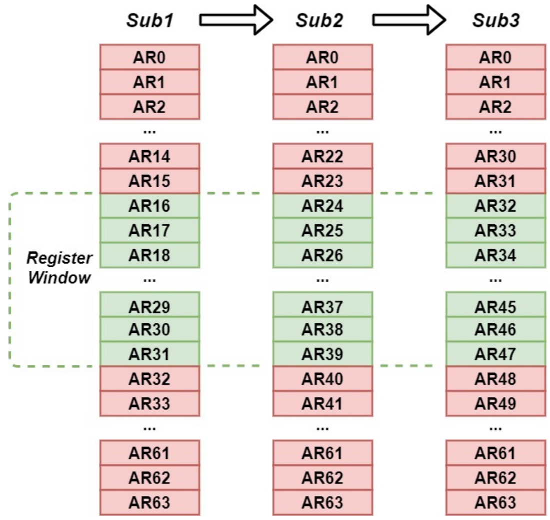

Fig. 1: Overview of the ESP32 register file. A subroutine only has access to the registers contained within the register window.

2) The Register Window: The ESP32 contains 64 general-purpose registers in the register file of the CPU. The Xtensa architecture implements a feature called **register window** which allows a subroutine to only access 16 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 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 as before.

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 several new attack proof-of-concepts against the ESP32. We have successfully launched two 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, before providing details on their implementation. We then show a proof of concept by using the web server on the ESP32 to launch the software attacks.

A. Threat Model

In the following attacks, we assume that the ESP32 may expose some kind of communication channel to the user, such as a web server. We also assume that the ESP32 stores some secret data in its firmware. The adversary may be local or remote. A local adversary cann physically access the device and use its UART port to monitor output from the device. A remote adversary has more limited capabilities since he cannot directly read the output from the device. However, it is possible for the remote attacker to use the format string attack to write contents into memory and overwrite the secret data [27]. Finally, we assume that the firmware contains some programming flaw, which is reasonable due to the abundance of software vulnerability types which are found in C [28] [29] [27] [21].

B. Attack Overview

Format string vulnerabilities [27] arise when formatting functions fail to validate a user's input format. An example of such a function is *printf()*, which accepts format string parameters 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 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 fetches the next 4 bytes in memory.

Based on the experiments we performed 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 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 strings are required to access memory contents, otherwise the user will only access register 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;
    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 [20]. 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 variables 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.

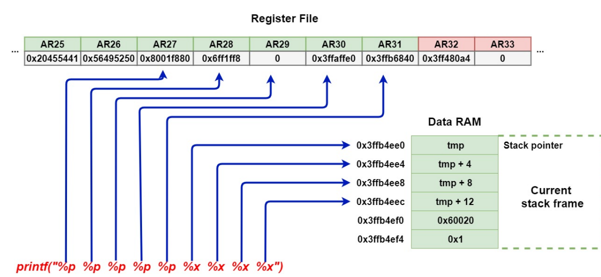

Fig. 2: Overview of the Same Subroutine Attack. The arrows show which addresses *printf()* will fetch and dereference.

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 format strings print the contents of AR27, AR28, AR29, AR30, and AR31, while the last four strings 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 50524956415445204b45590. 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 programs jumps to *sub2*, and the attacker can use the format string "%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;
    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 format string "%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.

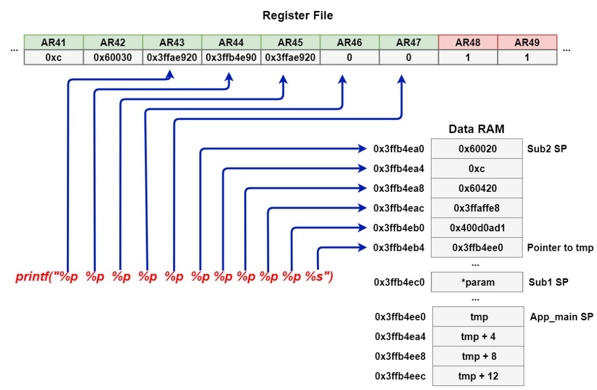

Fig. 3: Overview of the Cross Subroutine Attack. The register window and *printf()* behavior are defined with respect to *Sub2*.

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 format strings 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 format string, "%s", will cast 0x3ffb4eb4 as a char pointer and print its value. The output of this attack is shown below:

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

Unlike the payload shown in attack A, this payload prints the memory contents verbatim as long as it can be cast as a strong. This can lead to faster detection of sensitive data. One caveat is that this can cause unexpected errors if a string does not contain the null terminator, which will cause the program to read memory contents indefinitely and potentially crash.

D. Proof of Concept Using a Web Server

In this section, we show how an application that exposes a web server interface may fall victim to the format string attack. For this attack, 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 WebServer library written for the Arduino platform. This library allows a web server to process HTTP requests from the client and send responses back. Our program contains the same vulnerable format string function as described earlier. Listing 3 shows the psuedocode for this attack. This program will open a local web server at port 80, allowing clients to connect to it. When the client connects, they can pass GET parameters in the URL bar of the web browser, and the ESP32 will process the first parameter and print it to the console.

Listing 3: Web attack psuedocode.

```
#include <WebServer.h>
WebServer server(80);
void handleRequest() {
    char* param = server.arg(0).c_str();
    printf(param);
    printf("\n"); }
void setup() {
    ... // Connect to wifi, establish server
    server.on("/", handleRequest); }
void loop() { server.handleClient(); }
```

To exploit the vulnerability, the attacker can send a GET request containing an URL-encoded format string. In HTML, the "%" character is encoded as "%25". Therefore, to pass the format string "%p %x %s", the attacker can send the following GET request to the ESP32:

```
http://<IP addr>/?a=%25p%25x%%s
```

The server will decode the URL contents and recover the original payload. This will trigger the format string attack.

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. The TrustZone technology has been integrated into Arm Cortex-M processors, denoted as TrustZone-M. However, TrustZone-M can be compromised too [16]. 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 [17] 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 [19].

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. 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.

The manufacturing phase of the development lifecycle can be broken down into five key steps: wafer fabrication, wafer probing, packaging, final testing, and assembly. **Wafer fabrication** is the construction of the silicon die to connect the electrical components together. **Wafer probing** performs electrical tests to verify the functionality of the silicon chip. **Packaging** packages the die (i.e. block of semiconducting material) to protect the electrical components from any damage. The **final testing** subjects the integrated circuit to various production tests to ensure that the chip's design is reliable. Finally, **assembly** refers to the production of printed circuit boards (PCBs) and assembling the modules and chips onto the PCB.

Our secure key provisioning framework is shown in Figure 4. It is composed of six main entities. The **crypto facility** manufactures the crypto coprocessors and distributes it to the secure facility. The **secure facility** is in charge of provisioning crypto chips with private keys and certificates. The secure facility has a self-signed root CA which can be used to sign device certificates. The **original device manufacturer**, or ODM, is tasked with creating the hardware of the end device. The **build server** creates the firmware and software for the end device. The **end user / end device** is the final IoT product / the owner of the final IoT product. Finally, the **runtime server** serves as the application server and and authenticates the end device's public key and certificate.

1) Key Provisioning & Prerequisites: In **Step 1A**, the crypto facility manufactures the unprovisioned crypto coprocessor within its own secure facility. In **Step 1B**, the build server develops the application for the end device

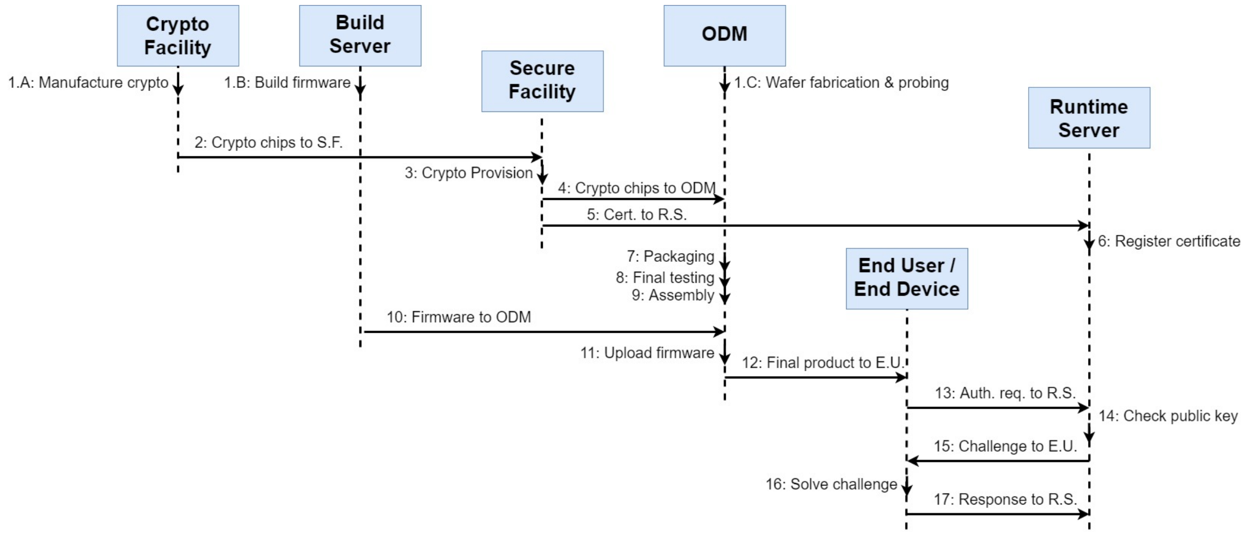

Fig. 4: Secure key provisioning framework.

and performs testing on development kits. In **Step 1C**, the manufacturer performs the wafer fabrication and wafer probing test of the silicon die. In **Step 2**, the crypto facility distributes crypto chips to the secure facility. Secure key provisioning is performed by the secure facility in **Step 3**. The secure facility will provision each crypto chip to internally generate a private key. Additionally, the secure facility will generate and store a device certificate into the device. Finally, 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.

2) Finalizing the Hardware: In **Step 4**, the secure facility distributes the provisioned crypto chips to the ODM, and in **Step 5**, it distributes the certificates to the runtime server. The runtime server shall save these certificates to a registry in **Step 6**. Next, in **Step 7**, the manufacturer performs the packaging process of the end device. In **Step 8**, the chip undergoes the final testing phase within the ODM to ensure that the hardware meets production requirements. In **Step 9**, PCBs are printed and the chip and modules (sensors, etc.) are assembled onto those PCBs. The ODM obtains the firmware from the build server in **Step 10** and uploads the firmware to each end device in **Step 11**. This creates the final product for the end user.

3) Device Registration: After the completion of the manufacturing process, the final IoT product is shipped to the end user and authenticates to the runtime server. In **Step 12**, the end user obtains the final product from the ODM. Now, the end user turns on the end device and connects to the Internet. In **Step 13**, the end device sends an authentication request to the runtime server which includes the public key of the crypto coprocessor. Note that for mutual authentication to succeed, the firmware (or crypto coprocessor) should also store the CA certificate of the runtime server to authenticate it successfully.

In **Step 14**, the runtime server searches its certificate registry to ensure that the given public key is valid. Then it will initiate a challenge-response procedure to ensure that the end device owns the associated private key. The runtime server generates a random number *N* and encrypts it using the given public key. Denote this encrypted number $E_{dev}(N)$. The runtime server sends $E_{dev}(N)$ to the end device in **Step 15**. In **Step 16**, the end device feeds $E_{dev}(N)$ into a decryption function within the crypto chip that uses the private key to solve the challenge, obtaining $D_{dev}(E_{dev}(N)) = N$. The end device sends the authentication response *N* back to the runtime server in **Step 17**. At this point, the runtime server may verify that the received *N* matches the original *N*, thus proving the end device's ownership of the private key. At this point, the runtime server and end user can proceed with the normal application.

C. Security Analysis

With our defense enabled, all attacks in this paper will fail. Recall that the payload of both attacks in Section III will leak a private key that is stored in the firmware. SIC^2 shall store such a private key in the hardware of the crypto coprocessor, thus preventing any access to this key via the format string attack. In addition, control flow attacks [29] and code injection attacks [3] can never execute code which reads the key value, because this code does not exist. In fact, due to the hardware isolation between the MCU and crypto coprocessor, *all* software attacks shall fail to recover the private key.

When considering the secure key provisioning 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 an ODM may manufacture many different products, and the runtime server must only accept certificates which belong to its own products. To address this, as shown in to **Steps 5 and 6** of the provisioning framework, the runtime server will receive certificates from the secure facility and can know ahead of time which certificates to trust. In this way, even if the ODM swaps crypto coprocessors in the packaging phase, the runtime server will reject certificates that have not been registered.

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. 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 [1].

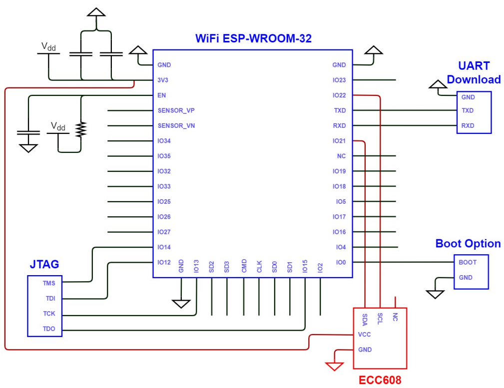

Fig. 5: ESP-WROOM-32 with ECC608 schematic.

A. ATECC608A Overview

The ECC608 comes packaged in the Small Outline IC (SOIC) format. In a production environment, the SOIC may be directly soldered to a printed circuit board (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 for signature generation and verification, and extraction of the public key, but restricts reading/write access to the private key. The ECC608 is also capable of generating a certificate signing request (CSR) from the private key. This is necessary for attaining a valid X.509 certificate. To prevent malicious configuration, the *configuration* memory zone can be locked. Similarly, to prevent overwriting a key or certificate, the user may lock the *data* memory zone.

A device can communicate with the ECC608 via the CryptoAuthLib software library [11]. We have successfully ported this library to ESP-IDF. In addition to setting and locking the configuration and data zones, CryptoAuthLib can be used to send commands to the ECC608. The host MCU communicates with the ECC608 via the I^2C protocol. The host MCU and ECC608 may also share a mutual IO secret, which obscures the I^2C traffic by encrypting data with the secret value. This results in a safer communication channel.

To achieve network communication, we use *MbedTLS* [8], 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 authenticate the chip. We will investigate these use cases in our future work.

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.

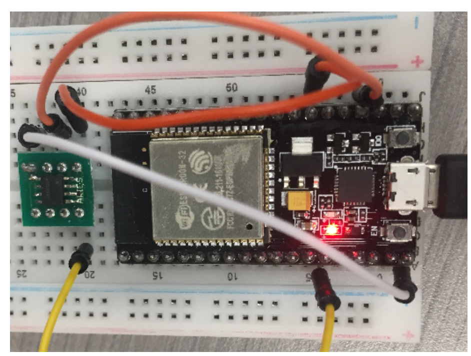

Fig. 6: ESP32 wired to an ECC608.

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 (SCL) 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 GUI that can communicate with the ECC608 via an external programmer, such as the ATSAMD21 board [10]. As an alternative to

[1]Available at https://github.com/PBearson/ESP32-With-ECC608.

ACES, CryptoAuthLib can be used to configure these parameters.

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” to the root directory. The file includes 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^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. 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 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.

VI. Evaluation

In this section, we explore the improvements to speed and energy consumption provided by the integrated ECC608 crypto chip. We also discuss the area overhead of ECC608 that is added to an MCU. For performance assessment of the ESP32 security features, please refer to Appendix C of the technical report [19].

A. ECC608 Area Overhead

We have paired an ECC608 with our ESP32 development board based on the ESP-WROOM-32 module. However, in production, manufacturers may wish to connect these components together onto a PCB. We have calculated the size of the ECC608 and WROOM and determined the area overhead of the 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, the overhead of the ECC608 is minimal and shows that a manufacturer can integrate these crypto chips into their hardware without much sacrifice.

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 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 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 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.

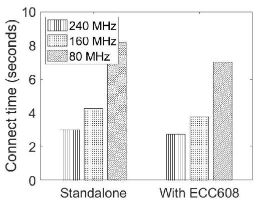

Fig. 7: AWS handshake time.

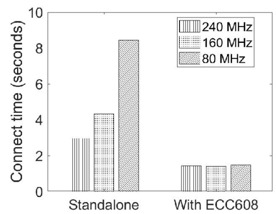

Fig. 8: EC2 handshake time.

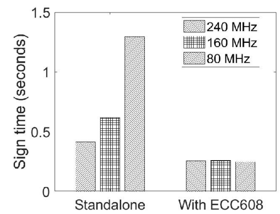

Fig. 9: ECDSA sig. gen. time.

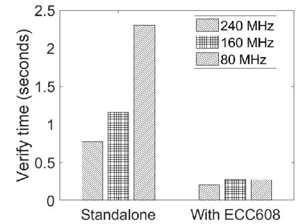

Fig. 10: ECDSA sig. verify time.

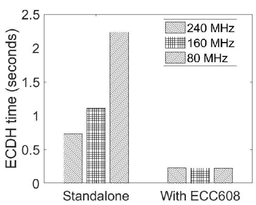

Fig. 11: ECDH time.

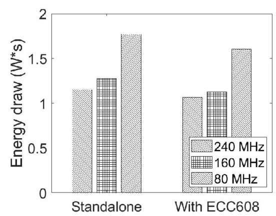

Fig. 12: AWS handshake energy draw.

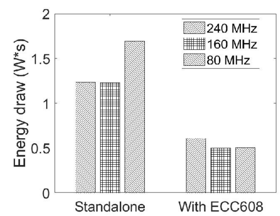

Fig. 13: EC2 handshake energy draw.

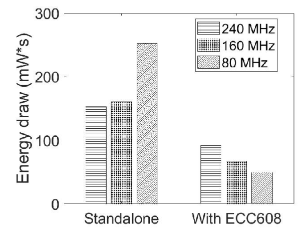

Fig. 14: ECDSA sig. gen. energy draw.

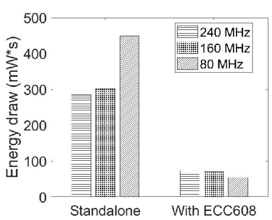

Fig. 15: ECDSA verify energy draw.

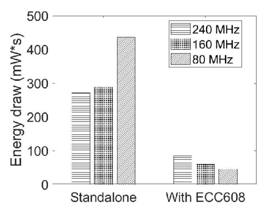

Fig. 16: ECDH energy draw.

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 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, and with the different clock speed settings.

VII. Related Work

In this section, we discuss some software and hardware attacks that have targeted ESP32 and ESP8266 [24] in recent years. The ESP8266 is a popular MCU from Espressif and the predecessor to ESP32.

Hardware Exploits. Researchers have exposed critical hardware vulnerabilities on ESP32-based smart devices. 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 [14]. A similar attack was performed on WIZ smart bulb [15]. Researchers also performed a voltage glitching attack on the ESP32 ROM with full security settings enabled, triggering a full readout of the security keys [13]. The latter attack cost several hundred dollars and can only be addressed with a full hardware revision. Espressif has already announced a new ESP32 chip that mitigates against fault injection [26]. These kinds of attacks and responses show the importance of hardware security in modern IoT systems.

Software Exploits: Researchers have reported several vulnerabilities that affect ESP32 and ESP8266 software libraries. The *Zero PMK Installation* vulnerability affects the EAP authentication framework [6]; 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 [4]. 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 [5]. Note that ESP-IDF version 3.3

and NONOS version 3.1 address all of the aforementioned vulnerabilities.

It is shown in [22] that some applications may be vulnerable to a stack-based buffer overflow attack, or simply BOF. On the ESP32, the return address of a subroutine *Sub1* is saved to the first register in a register window, called A0. On a subroutine call to *Sub2*, A0 is dumped to memory 16 bytes behind *Sub2*'s stack pointer. If the new function uses *strcpy* or *strcat* to copy a user input into a local buffer, an attacker can overwrite this buffer and modify the return address, thereby controlling the return address of *Sub1*.

These attacks all show that software can often contain many security vulnerabilities. As we illustrated in Section III, only a single vulnerable line of code may be enough to cause destruction to an application. Hence, it is the important that the user's most critical data, i.e., their private keys, is always protected from such software vulnerabilities, which can be accomplished using crypto coprocessors.

VIII. Conclusion

In this paper, 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. We demonstrate two 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.

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] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. CoRR, abs/0901.3482, 2009.

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

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

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

[7] 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.

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

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

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

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

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

[13] 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.

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

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

[16] L. Luo, Y. Zhang, C. C. Zou, X. Shao, Z. Ling, and X. Fu. On runtime software security of trustzone-m based iot devices. https://arxiv.org/abs/2007.05876, Jul 2020.

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

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

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

[26] 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.

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

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

[29] 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.

Appendix A
Full MCU Dataset

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 I displays our full MCU dataset. 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
ONE SCRIPT SECURITY FOR THE ESP32

If developers decide to utilize the ESP32 boards in their projects, they can take additional steps to protect hardware and firmware by incorporating flash encryption and secure boot into their applications. To complement this, we have provided a software solution for IoT device developers that may integrate the ESP32's security features into their projects. We label this software solution "One Script Security" or OSS. The flash encryption and secure boot functions of the ESP32 are enforced by the eFuse secure memory block as described in Section II.

Traditionally, in order to enable flash encryption and secure boot, developers can navigate through an interactive menu using the *idf.py* tool. This tool will configure various settings into a *sdkconfig* text file that is used to compile some features into the firmware. Moreover, developers can utilize the *espsecure.py* and *espefuse.py* to generate and set the flash encryption and secure boot keys. If the developer enables flash encryption or secure boot without first providing external keys, the ESP32 will internally generate these keys and they will be permanently inaccessible to the developer.

Internally generated security keys are considered the safest approach but can result in a soft brick of the device. For instance, the ESP32 supports only three reflashings of plaintext firmware while allowing for unlimited OTA updates; if the developer neglects to add OTA support into the firmware, then he cannot update the device any further. This restriction is enforced with the help of the *FLASH_ENCRYPT_CNT* eFuse. Without the secure boot key, developers cannot modify the software bootloader and produce a valid digest; however, many configuration options in *sdkconfig* may modify the bootloader without the user's knowledge, leading to accidental tampering of the chain of trust. Enabling flash encryption will also modify the bootloader, so these features must be enabled simultaneously.

We write OSS with the previous pitfalls in mind. **The software will externally generate the secure boot and flash encryption keys and read/write protect them to the current user**. The software will also encrypt and upload the firmware and bootloader at the proper address offsets. The secure boot key is derived from a digest of the private ECDSA *secure boot signing key*; in this way, the developer can always use the ECDSA private key to generate the secure boot key, which in turn will be used to generate a valid digest of the software bootloader. Since the flash encryption key is accessible to the developer, he can encrypt the firmware *before* uploading the ESP32; this can be done an unlimited number of times. As long as the flash encryption key and secure boot signing key are not deleted, a developer can always produce a valid software image. Additionally, OSS will disable the JTAG and UART debugging ports.

Importantly, our solution integrates components from ESP-IDF such as *idf.py* and *espsecure.py*, and depends on the general structure of the build directory (which stores plaintext elf and binary files). As a result, OSS can be imported into existing projects based on ESP-IDF, and developers do not have to migrate away from this platform. Furthermore, if the developer wishes, he may still upload firmware physically.

A. OSS Implementation

Our OSS solution is written as a shell script that can directly access components of ESP-IDF. The script assumes that ESP-IDF and the Xtensa toolchain have already been installed. The developer will effectively work in place of *idf.py flash*, which is the typical method for building and uploading an app to the ESP32. The source code is available at Github [2].

The functionality of OSS is split into two phases: *first time setup* and *repeat setup*. In *first time setup*, the ESP32 has not yet configured security settings and eFuses, while in *repeat setup*, the security settings have already been configured. OSS can predict which setup to execute by analyzing the ESP32's eFuse contents; if the flash encryption key and secure boot key read out "????...????", then this implies that the security keys have been set already and that the *repeat setup* is appropriate. On the author hand, a reading of "0000...0000" suggests that the security keys have not been set yet, and therefore *first time setup* should execute. Alternatively, we provide configuration options that allow the user to manually run the desired setup.

The *first time setup* works as follows. First, *espsecure.py* will generate a new flash encryption key and secure boot signing key; these keys will be read/write protected to the current user. Then *espefuse.py* will burn and write-protect the following eFuses: *BLK1* (flash encryption key), *BLK2* (secure boot key), *FLASH_CRYPT_CNT*, *FLASH_CRYPT_CONFIG*, *ABS_DONE_0*, *JTAG_DISABLE*, *DISABLE_DL_ENCRYPT*, *DISABLE_DL_DECRYPT*, and *DISABLE_DL_CACHE*. Recall that the secure boot key is derived from the signing key. Finally, the script will proceed with building, signing, and encrypting the application before finally uploading it to the ESP32.

The *repeat setup* skips the process of configuring eFuses and proceeds directly to the build step. In this case, the bootloader is flashed to an offset of 0x0 rather than 0x1000, which accommodates for the newly added bootloader digest that is stored at the start of the bootloader image. To properly encrypt and upload applications, OSS must determine the appropriate address offset of the firmware, bootloader, partition table, and if applicable, the OTA data partition. The bootloader offset is strictly 0x0 or 0x1000, while the firmware and OTA data offsets can be determined from running *make partition_table*. The partition table offset can be determined from the *sdkconfig* text file.

To ensure that the bootloader and firmware will correctly compile in the security features into the binary, OSS will peak at the *sdkconfig*, and if the corresponding security settings have not been configured, OSS will not run. Without

[2]Available at https://github.com/PBearson/ESP32-One-Script-Security

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	Cortex 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	Yes	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
STM32L5	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%)

TABLE I: The full MCU dataset for hardware security of IoT devices.

flash encryption support, an application will not have the necessary code to read or write to external flash over the SPI channel. Without secure boot support, the compiler will not generate the bootloader digest, and the bootloader will not contain the code necessary to validate the firmware.

APPENDIX C
ESP32 SECURITY EVALUATION

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 II shows insight into the binary sizes when flash encryption and secure boot are enabled/disabled. 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.

TABLE II: Binary size insight (in *kilobytes*).

Binary	App size	Insecure	Secure
Bootloader	Small	25.38	37.39
Bootloader	Large	27.36	39.38
Firmware	Small	144.32	196.60
Firmware	Large	1407.10	1441.78

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 III: Build time benchmarks (unit *seconds*).

Benchmark	App size	Insecure	Secure
Compile	Small	33.94	34.65
Compile	Large	52.90	53.74
Encrypt	Small	N/A	3.21
Encrypt	Large	N/A	19.53
Upload	Small	4.55	20.78
Upload	Large	25.03	134.87
All	Small	38.50	58.64
All	Large	77.92	208.14

Table III shows our evaluation metrics. 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.

TABLE IV: Run time benchmarks (unit μs).

Benchmark	Insecure	Secure
int8 add	146	147
int8 div	374	377
int32 add	109	109
int32 div	274	275
int64 add	445	447
int64 div	4478	4481
float sin	1.11×10^4	3.06×10^4
float div	2506	2510
double sin	1.10×10^4	1.11e+4
double div	3.16×10^4	3.16×10^4
double sqrt	4786	4793
copy string	3150	3150
matrix mult	1.57×10^5	1.57×10^5
read to flash	7.30×10^4	7.26×10^4
write to flash	5.23×10^7	4.77×10^7

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 including 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.

Figure IV shows the results of each benchmark. 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.