On Misconception of Hardware and Cost in IoT Security and Privacy

Bryan Pearson*, Lan Luo*, Yue Zhang*†, Rajib Dey*, Zhen Ling‡, Mostafa Bassiouni*and Xinwen Fu*
*College of Engineering and Computer Science, University of Central Florida, USA
Email: {bpearson,lukachan,rajibdey}@knights.ucf.edu, {yue.zhang,bassi,xinwenfu}@ucf.edu
†College of Information Science and Technology, Jinan University, China
‡Department of Computer Science, Southeast University, China
Email: zhenling@seu.edu.cn

***Abstract*—The popularity of IoT has raised grave security and privacy concerns. There is a misconception that security and privacy issues of IoT systems are caused by the hardware and its cost. In this paper, we will explore the use of microcontrollers (MCUs) and crypto modules in IoT applications and demonstrate that hardware and cost may not be the bottleneck of IoT security and privacy in various application domains. We discuss how to implement hardware security, system/firmware security, network security, and data security with the low-cost Espressif's ESP32, TI's CC3220 and Microchip's cryptographic co-processor ATECC608A. We perform extensive experiments to validate the performance of cryptographic and networking operations of IoT devices based on those and other MCUs and crypto modules. We are the first to perform a comprehensive measurement and comparison of cryptographic and networking performance of these modern IoT MCUs and modules.**

I. Introduction

Internet of Things (IoT) interconnects everything including physical and virtual objects together through communication protocols. IoT has broad applications in digital healthcare, smart cities, transportation, agriculture, logistics and many other domains. The global IoT market is booming. According to Forbes [2], the IoT Market will reach $520B by 2021.

The popularity of IoT has raised grave concerns about security and privacy [12], [14]. When medical devices are connected to the Internet, compromised medical devices may endanger the lives of patients. Hacked autonomous cars may crash. Hackers exploited default passwords and user names of webcams and other IoT devices, and installed the *Mirai* botnet [1] on compromised IoT devices. The huge botnet was then used to deploy the DDoS attack against Dyn DNS servers. The IoT *reaper* botnet was discovered in 2017 [5] and exploited newly found vulnerable IoT devices.

There is a misconception that the security and privacy issues of IoT are caused by incapable hardware and the associated cost. For example, it is believed that it is hard to adopt secure hardware and achieve the desired security such as public key cryptography based mutual authentication while preserving decent networking performance for smart home products. In this paper, we will explore how to secure low-cost microcontrollers (MCUs) based IoT applications. Sensor nodes in various smart systems such as smart home, smart health and smart grid can use MCUs to process commands and automatic control.

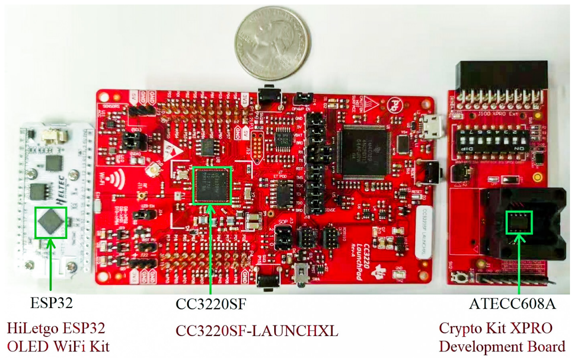
ESP32
HiLetgo ESP32 OLED WiFi Kit
CC3220SF
CC3220SF-LAUNCHXL
ATECC608A
Crypto Kit XPRO Development Board

Fig. 1: ESP32, CC3220 and ATECC608A and development boards

Our major contributions can be summarized as follows. First, we discuss how to implement hardware security, system/firmware security, network security, and data security through Espressif's ESP32 ($3.45 at aliexpress) [4], TI's CC3220SF ($6.79 at TI) [13] and Microchip's ATECC608A ($0.55 at Microchip) [6]. Figure 1 shows these modules and the corresponding development boards. ATECC608A is a crypto co-processor module with AES, HMAC and ECC (elliptic curve) hardware acceleration and secure key storage; it can be used with a MCU or microprocessor such as ESP32 and CC3220F to provide public key cryptography based mutual authentication and communication secrecy and integrity. Second, we perform extensive experiments to demonstrate the performance of cryptographic and networking operations of those and other MCUs and modules, and show that the low cost MCUs and crypto chips will be able to meet the security and privacy requirements in domains where MCUs are used. We are the first to perform a comprehensive measurement and comparison of cryptographic and networking performance of these modern IoT MCUs and modules.We advocate the use of these low-cost and low-power modules to secure IoT systems.

The rest of this paper introduces readers to ESP32 security features in Section II, discusses differences between ESP32 and CC3220 and capabilities of the ATECC608A in Section III, and presents our evaluation of the cryptographic and networking performance of these devices in Section IV. Section V concludes the paper.

II. Secure MCU Based IoT System Via ESP32

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

A. Security Requirements of IoT Systems

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

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

In order to secure network traffic to and from the IoT device, we can use SSL/TLS (which we will refer to simply as TLS) to establish mutual authentication, message encryption, and message integrity between the device and a server. *Mutual Authentication* is necessary and critical for any IoT system. We have explored various systems and found that those without mutual authentication often have various vulnerabilities [8], [7], [11], [9], [10], [3]. Without client authentication, a fake client may solicit security credentials from the server or a smartphone app. Without server/user authentication, a fake server or a fake user can cheat on the clients and collect sensitive information. Certificate based mutual authentication based on public key crypto is often the most feasible and simple implementation of mutual authentication. In TLS' certificate based mutual authentication, a client verifies the server's certificate and identity. The server performs similar operations to authenticate the client.

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

B. Hardware Security: Disabling JTAG and UART

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

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

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

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

The primary purpose of UART with respect to the ESP32 is to upload an application or firmware to the flash. Other possibilities with UART include monitoring output from the console, and reading or modifying direct memory addresses. The UART bootloader is implemented through an external interface known as *esptool*. If flash content is encrypted by the encryption key stored in the eFuse, then the UART bootloader will transparently decrypt this content before reaching the serial monitor. Similarly, the UART bootloader will transparently encrypt data when flashing it via *esptool*.

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

- DISABLE_DL_ENCRYPT: disables data encryption in UART bootloader mode
- DISABLE_DL_DECRYPT: disables data decryption in UART bootloader mode
- DISABLE_DL_CACHE: disables cache access in UART bootloader mode

Afterwards, the UART bootloader cannot read or write to the encrypted flash. If the UART bootloader tries to read data, it will find that everything is encrypted. Similarly, if the UART bootloader tries to upload plaintext data, this data will not function correctly, since the flash controller will transparently "decrypt" the data, effectively corrupting it. Malicious users cannot override the encryption properties of the flash, since they are set and enforced by the hardware.

C. System and Firmware Security

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

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

TABLE I: ESP32's security-related eFuse fields (*Size* in bits)

Name	Description	Size
FLASH_CRYPT_CNT	Flash encryption counter	8
FLASH_CRYPT_CONFIG	Flash encryption config.	4
CONSOLE_DEBUG_DISABLE	Disable ROM console	1
ABS_DONE_0	Enable secure boot	1
JTAG_DISABLE	Disable JTAG	1
DISABLE_DL_*	Disable UART	3
BLK1	Flash encrypt key	256
BLK2	Secure boot key	256
BLK3	Undefined	256

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

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

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

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

The first step in secure boot is validation of the software bootloader. When compiled into flash, the software bootloader contains three attributes: (i) *Binary*: the image executed by the software bootloader; (ii) *ECDSA public key*: used to verify the application signature; (iii) *AES digest*: a hash-based message authentication code (HMAC) used by the ROM bootloader to validate the software bootloader. Upon reading the software bootloader, the ROM bootloader will calculate its own AES digest to validate the integrity of the software bootloader. If the digest is valid, the ROM executes the software bootloader. However, if the digests fail to match, the ROM refuses to execute the software bootloader, and the system halts.

The second step of secure boot is validation of the firmware. Every firmware image and partition table must contain an ECDSA signature appended to their binaries. When loaded into flash, the application firmware contains two attributes: (i) *Binary*: the image executed by the firmware; (ii) *ECDSA signature*: generated using the binary and the ECDSA private key. The software bootloader uses its ECDSA public key to verify the ECDSA signature in the application firmware with the binary as the input.

D. Data Security

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

When flash encryption is enabled, application-based flash partitions, i.e. factory and over-the-air (OTA) partitions, are encrypted by default. From there, decryption can only occur at runtime via the flash controller. The flash controller is a hardware component that can use the AES key to perform the following operations: (i) Decryption of memory-mapped read accesses to flash; (ii) Encryption of memory-mapped write accesses to flash.

It is also possible to encrypt other flash partitions by manually setting an "encrypt" flag for a partition. This requires generating a custom partition table rather than using the default table (which only encrypts factory and OTA partitions). All partitions have the option for their content to be encrypted, with the exception of *nvs*, or non-volatile storage (NVS), which persists through the power cycle.

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

E. Network Security

The challenge to implement TLS on an IoT device is often the cost and efficiency of implementing the public key based cryptographic functionalities. As shown in this paper, the hardware and cost may no longer be the bottleneck. ESP32 has cryptographic hardware acceleration for RSA and Random Number Generator (RNG), while ECC hardware acceleration is limited based on our experiments. Our extensive experiments show that the performance of TLS is satisfactory in various application domains. ESP32 also has cryptographic hardware acceleration for AES and

SHA-2 in addition to RSA and RNG so that TLS can be fully implemented. Therefore, AES encryption can be implemented for communication secrecy, and HMAC will achieve communication integrity.

F. Secure Over-the-Air Updates (OTA)

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

III. Discussion

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

A. Differences from the TI CC3220

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B. Microchip ATECC608A

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

IV. Evaluation

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

A. Experiment Setup

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

Figure 1 shows some of the development boards we use to program those modules. Note that the development board is a device that contains a chip such as the ESP32 and is used to evaluate the chip. For the ESP32, we use the HiLetgo ESP32 OLED WiFi Kit ($18.99 at Amazon) while one without the OLED display costs around $10.99 at Amazon and around $5 at aliexpress. The programming environment is Espressif IoT Development Framework (esp-idf), Arduino integrated development environment (IDE), or the Mongoose OS firmware development framework. For ESP8266, we use a NodeMCU development board (around $6.50 at Amazon). We program in Mongoose, running ESP8266 at 160MHz. For CC3220, we use TI's CC3220SF-LAUNCHXL development board ($49.99 at TI) and run at 180 MHz. The programming environment is the Code Composer Studio (CCS) IDE. For ECC608, we use Microchip's Crypto Kit UDFN Socketed XPRO Development Board ($85 at Microchip). The programming environment for the AES132 and ECC608 is Atmel Studio 7; additionally, these crypto chips can be programmed through the ESP32 or ESP8266.

B. Summary of Measurement Results

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

C. AES, HMAC, ECC, and RSA

We now show the box plots of these measurements. We first show the performance of AES key generation, encryption, and decryption. Figures 2 and 4 showcase these results, respectively. We use a key size of 256 bits and cipher block chain (CBC), except in the cases of the ESP8266 and AES132. ESP8266 only implements 128-bit AES operations due to RAM constraints, while the AES132 is restricted to the 128-bit key size in Counter with CBC-MAC (CCM) mode. For all other chips, we choose to measure AES-256 in

TABLE II: Summary of cryptography metrics (unit μs) for the ESP32, CC3220, ECC608, AES132, and ESP8266.

Evaluation	ESP32 (240MHz)	CC3220 (180MHz)	AES132A	ESP8266 (160MHz)	ECC608 Standalone	ESP32 with ECC608	ESP8266 with ECC608
AES encryption	4.05	38.8	10.0×10^3	153	6.10×10^3	N/A	N/A
AES decryption	4.12	39.5	10.0×10^3	145	6.70×10^3	N/A	N/A
HMAC	154	45.1	N/A	182	25.9×10^3	N/A	N/A
ECC sig. gen.	9.29×10^4	3.87×10^5	N/A	2.48×10^5	90.2×10^3	N/A	N/A
ECC sig. verify	3.32×10^5	7.09×10^5	N/A	6.97×10^3	45.1×10^3	N/A	N/A
RSA sig. gen.	159	N/A	N/A	N/A	N/A	N/A	N/A
RSA sig. verify	2.27×10^3	N/A	N/A	N/A	N/A	N/A	N/A
MQTT conn. establishment	2.30×10^6	6.99×10^5	N/A	2.85×10^6	N/A	1.10×10^6	1.40×10^6
MQTT RTT	3.99×10^4	4.79×10^4	N/A	8.32×10^4	N/A	5.90×10^4	5.22×10^4

CBC mode because it is the same algorithm used to encrypt the flash contents on the ESP32.

In our runs of AES key generation, CC3220 performed approximately 226 μs faster than ESP32. Conversely, for AES encryption and decryption, the ESP32 performed faster than CC3220 and ESP8266 by a large margin, and slightly faster than the ECC608 and AES132. The AES132A showed the worst performance at 10 ms for encryption and decryption. Encryption and Decryption operations performed considerably faster than key generation.

Next, we measure HMAC, whose results can be seen in Figure 5. For ESP32, CC3220, and ESP8266, we use a key size of 112 bits, while the ECC608 uses a 256-bit key size due to hardware restrictions. All chips use the SHA-256 hash function. The final HMAC is 256 bits. Our tests indicate that ESP32 executes HMAC slower than CC3220 and ECC608 by approximately 100 μs. CC3220 showed the strongest performance at only 45.1 μs. The ECC608 performed the worst at 25.9 ms. Similar to AES, all metrics, except the ECC608, are on the order of μs, likely due to SHA-2 hardware acceleration.

For ECC, we first use ECDH (Elliptic Curve Diffie-Hellman), followed by ECDSA (Elliptic Curve Digital Signature Algorithm) to generate and verify the digital signature. We use the SECP256R1 curve and a 256-bit sized key. ECC is particularly advantageous over RSA in terms of speed and key size. The results of ECC performance on the MCUs can be observed in Figures 3 and 6. ESP32 outperformed the CC3220 in all three benchmarks. The ESP8266 showcased a median run time of approximately 0.25 seconds for signature generation and 0.07 seconds for verification. It is observed that ECC operations are several orders of magnitude slower than AES and HMAC. This behavior is expected and well-documented.

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

Figures 7 and 8 plot the results of RSA key generation, signature generation, and signature verification on the ESP32, in comparison to ECC. We continue to use the SHA-256 hash function for consistency with ECC. RSA key generation variance was significant. ECC key generation performed faster and more consistently; however, RSA signature operations fared much better than ECC due to hardware acceleration. As expected, all operations fell on the order of seconds, with key generation performing at least ten times slower than signature operations in most cases.

D. *MQTT*

In our setup, we use the Amazon AWS IoT broker in the North Virginia region to measure MQTT connection establishment and round-trip delay. We publish messages with a quality of service (QOS) level of 1, ensuring that AWS will acknowledge our messages by responding with *PUBACK* message packets. The run times for ESP32 and CC3220 can be seen in Figures 9 and 10. We also measure performance of these chips when leveraging the ECC608's hardware acceleration.

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

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

V. CONCLUSION

In this paper, we study modern MCUs and crypto co-processors including Espressif's ESP32, TI's CC3220 and Microchip's ATECC608A in terms of their cryptographic and networking operation performance. It can be observed that these MCUs and modules can provide satisfactory hardware security by disabling the I/O interfaces, system/firmware security through secure boot, network security through SSL/TLS (including mutual authentication that is required by Amazon AWS IoT), and data security through flash/file encryption and Over-the-Air (OTA) firmware upgrade through wireless or HTTPS. The very low cost ATECC608A can be added to various MCUs and microprocessors as a crypto co-processor to secure the overall IoT system, and meet the performance requirements of networking.

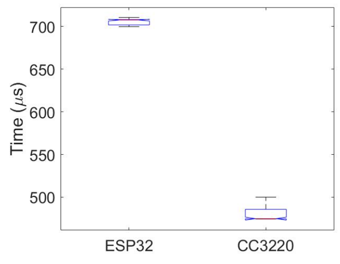

Fig. 2: AES (256) key generation

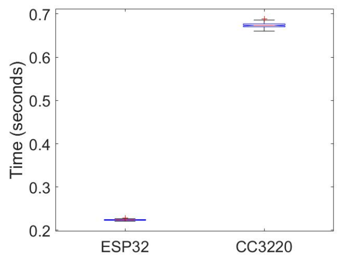

Fig. 3: ECC (SECP256R1) key generation

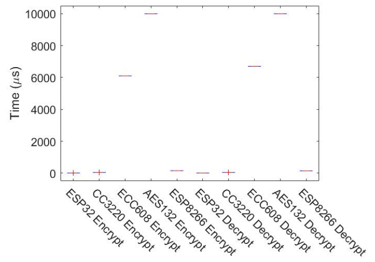

Fig. 4: AES encryption and decryption

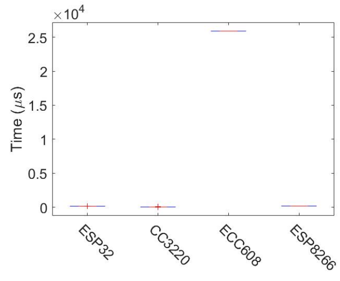

Fig. 5: HMAC with SHA-256

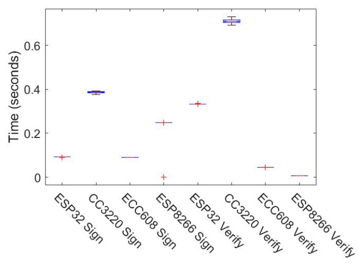

Fig. 6: ECC signature operations

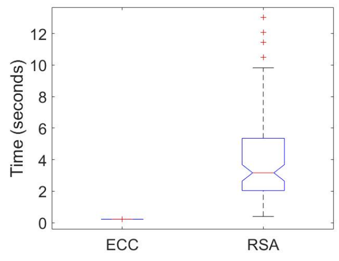

Fig. 7: ECC (256) and RSA (1024) keypair generation for the ESP32

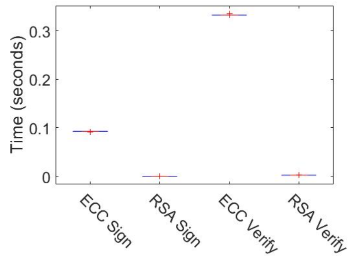

Fig. 8: ESP32 signature operations using ECC (256) and RSA (1024).

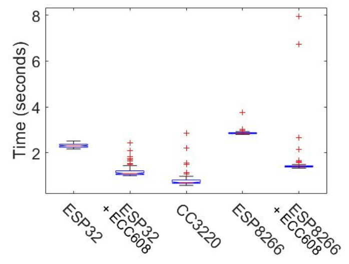

Fig. 9: AWS IoT connection establishment

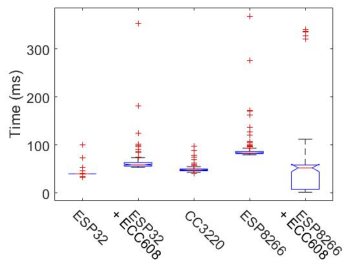

Fig. 10: Round-trip delay of MQTT packets

Acknowledgement

This work was supported in part by National Key R&D Program of China 2018YFB0803400 and 2017YFB1003000, by the US National Science Foundation under Grant Nos. (1642124 and 1547428), by National Natural Science Foundation of China under grants 61572130, 61532013, and 61632008, by Jiangsu Provincial Key Laboratory of Network and Information Security under grants BM2003201, by Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under grants 93K-9 and by Collaborative Innovation Center of Novel Software Technology and Industrialization. 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 *Proceedings of the 26th USENIX Security Symposium (Security)*, 2017.

[2] L. Columbus. Iot market predicted to double by 2021, reaching $520b. https://www.forbes.com/sites/louiscolumbus/2018/08/16/iot-market-predicted-to-double-by-2021-reaching-520b/, Aug 2018.

[3] N. Dhanjani. Security evaluation of the philips hue personal wireless lighting system. http://www.dhanjani.com/docs/Hacking%20Lighbulbs%20Hue%20Dhanjani%202013.pdf, 2013.

[4] Espressif Systems (Shanghai) Pte., Ltd. Esp32. https://en.wikipedia.org/wiki/ESP32, 2018.

[5] A. Greenberg. The reaper iot botnet has already infected a million networks. https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/, Oct. 2017.

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

[7] Z. Ling, K. Liu, Y. Xu, Y. Jin, and X. Fu. An end-to-end view of iot security and privacy. In *Proceedings of the 60th IEEE Global Communications Conference (Globecom)*, Singapore, December 2017.

[8] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu. Security vulnerabilities of internet of things: A case study of the smart plug system. *IEEE Internet of Things Journal (IoT-J)*, 2017.

[9] J. Molina. Learn how to control every room at a luxury hotel remotely. https://www.defcon.org/images/defcon-22/dc-22-presentations/Molina/DEFCON-22-Jesus-Molina-Learn-how-to-control-every-room-WP.pdf, 2014.

[10] J. Molina. Learn how to control every room at a luxury hotel remotely: The dangers of insecure home automation deployment. In *Proceedings of Defcon*, 2014.

[11] J. Obermaier and M. Hutle. Analyzing the security and privacy of cloud-based video surveillance systems. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security (IoTPTS)*, 2016.

[12] X. Peng, J. Ren, L. She, D. Zhang, J. Li, and Y. Zhang. Boat: A block-streaming app execution scheme for lightweight iot devices. *IEEE Internet of Things Journal*, 5(3), June 2018.

[13] Texas Instruments Incorporated. Cc3220. http://www.ti.com/product/CC3220, 2018.

[14] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu. Iot security techniques based on machine learning: How do iot devices use ai to enhance security? *IEEE Signal Processing Magazine*, 35(5), September 2018.