Secure IoT Development: A Maker's Perspective

Sian En OOI, Razvan BEURAN, Yasuo TAN
School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi City, Ishikawa Prefecture, 923-1292 Japan
{sianen.ooi, razvan, ytan}@jaist.ac.jp

***Abstract*—Efforts to tackle the challenges in securing Internet-of-Things (IoT) across the entire stack have been growing as IoT spreads over many domains. Applications of IoT in various domains are further accelerated by the Maker movement, as anyone with "googling" ability, the right tools and skill sets can develop a product. This creates an even larger attack surface, as security is generally not the main focus of a maker.**

To reconstruct the development process of a platform, we created a maker-oriented IoT hardware, MkIoT, and implemented an end-to-end (E2E) application prototype by leveraging existing off-the-shelf embedded hardware, open-source code, examples and tutorials provided by maker communities. The development process allowed us to investigate the challenges in securing both device and E2E communication, and in implementing life-cycle management using the public cloud. This paper examines and demonstrates the stumbling blocks and pain points of implementing a secure IoT application from the unique perspective of a maker, and serves as a reference for IoT makers, developers and researchers alike.

***Keywords*—Internet of Things (IoT), secure development, IoT hardware platform, IoT life-cycle, maker movement**

I. Introduction

As technological development advances, it makes technology more affordable and accessible to mass consumers. This trend is also observed in the IoT space, where the market for smart devices is increasing at a fast pace. While many technologies are technically complex and may require expertise in the field for development, another trend has emerged to empower the layperson: the "Maker" movement. The Maker movement has not only instilled "Do It Yourself" (DIY), hacking and tinkering culture into the general public, but found its way into the educational curriculum, to foster a better linkage between theory and practical applications [1], [2].

"You have a 9/10 chance that somebody already made it and that it's posted somewhere on the site. You would be dumb not to use it" [3]. This statement summarizes a maker's approach, where prototyping first is preferred to planning. Moreover, most maker-related discussions and tools do not place much emphasis on security [4], [5]. While various parties made efforts to promote secure IoT systems,such as best-practice security guidelines outlined by public cloud providers, semiconductor companies, the burden still falls on the maker to properly design and secure their creations. Furthermore, it is also difficult to practically validate and verify security in IoT due to its heterogeneous nature. While there is no one-size-fits-all approach when it comes to practical IoT experiments to investigate the pitfalls in proper implementation of security, in this paper, we attempt to analyse such issues by reconstructing the development process of an IoT application from a maker's perspective. This involves the design, development and implementation of a reference IoT system, MkIoT in order to capture the "behind the scenes" process. Thus, the core contributions of this paper are:

- Design an IoT hardware platform, MkIoT, to facilitate secure IoT development for makers
- Use MkIoT to demonstrate the challenges in securing the device and end-to-end communication
- Investigate the secure life-cycle management for public cloud applications

The remainder of the paper is organized as follows. Section 2 introduces the process of deriving the design requirements for IoT development. The technical details regarding the developed IoT hardware platform, MkIoT, are described in Section 3. Security implementation followed by its analysis and discussions are presented in Section 4. Section 5 covers the development and evaluation of the life-cycle management. The paper ends with conclusions and references.

II. Requirements from a Maker's Perspective

Designing an IoT device for practical verification is complex, as there are a wide range of IoT applications and many heterogeneous IoT devices that complement them. To deal with such complexity, the type of potential IoT applications can be narrowed while preserving flexibility to test possible applications on the same platform. These design decisions should be addressed by considering the following requisites: i) ease of development, ii) data acquisition, processing and storage, iii) connectivity, iv) power, v) security and vi) cost.

1) Ease of Development: IoT development should be relatively manageable, as the increase in development complexity will directly affect the cost for development and more importantly, the time-to-market. The ease of development can be determined by the accessibility, availability and quality of the required software and hardware tools. At the same time, many hardware and software enterprises introduced various reference development kits for their ecosystems to enable makers to prototype quicker (e.g., Microsoft Azure Sphere Guardian dev kit and Azure Cloud). Technical support by both manufacturers and communities is also an important factor in evaluating ease of development. This is analogous to the Stack Overflow importance for software development.

978-1-6654-3156-9/21/$31.00 ©2021 IEEE

2) Data Acquisition, Processing and Storage Requirement: The sensor, actuator, data processing and storage requirements are dictated by the target application. Data acquisition requirement would determine both the type and number of sensors and actuators on an IoT device, along with the measurement data resolution, velocity and retainment policies. The raw measurement data obtained from the sensors often require local preprocessing before uploading to the endpoint. This is where the processing requirement comes into play. The storage requirement is usually related to the processing requirement for preprocessing and local cache.

3) Connectivity Requirement: Connectivity is a core function of IoT, which provides embedded device with the capability to communicate with other endpoints through local network and/or the Internet. Although connectivity can be achieved through both wired and/or wireless communication methods, this research will put more emphasis on wireless communication. Requirements for wireless communication include parameters such as the target operating range, distance of signals to be transmitted, network latency, anticipated volume and rate of data to be transmitted. Besides, fault tolerance, security and privacy of the communication methods are also essential connectivity requirements, especially for remote IoT devices that employ mesh network.

4) Power Requirement: Power requirement for an IoT device is highly dependent on the target application and context. For example, many environment sensing IoT devices for smart home are meant to be physically tiny, wireless for both power and communication, and are expected to have a long lifespan. Such top-level requirements imposes a lot of constraint for both hardware, software and even architecture of the entire IoT application. A non-exhaustive list of requirements that would affect the power requirement of a typical IoT device are the number and type of sensors and/or actuators required, sampling frequency of sensors and/or frequency of actuation, rate of network transmission, operating range for wireless connectivity, encryption of packets and operating lifetime for non-rechargeable and replaceable battery.

5) Security Requirement: Device and data security, as well as its authentication, authorization, confidentiality and integrity of the data are paramount for any IoT devices. Before implementing security, one has to establish the use cases of the IoT application and its context. This includes identifying system domains and concerns to deduce the target security issues. In such circumstances, a practical approach can be applied for the specific use cases rather than blanket application of best practices. Detailed security requirements are further explained in later sections.

6) Cost Requirement: IoT devices are tightly constrained by their cost, as the above requirements directly or indirectly contribute to the total cost. A maker may have a budget for development, which is highly dependent on the specific application and use cases. Costs for sensor and actuator, power supply, dedicated cryptography chip for encryption, maintenance and operational are some example costs which the maker has to balance to satisfy their application requirements.

III. Reference IoT Hardware Platform

In order to be representative for generic IoT applications, the scope of possible applications for the MkIoT hardware platform should encompass common maker example use cases and applications. We follow the requirements presented in Section 2 for target use cases, such as simple smart home device, asset tracking, remote sensing in agriculture and many more applications that shares the same architecture. Thus, the MCU used for our MkIoT should be widely used in maker communities, supported by Arduino, have adequate processing capability and storage while being affordable. MkIoT wired connectivity should include a common bus to interface with off-the-shelf sensor and industrial modules, while it should also have WiFi and Low-power WAN (LPWAN) for wireless communication. In addition, MkIoT should be able to be powered through wired or battery source.

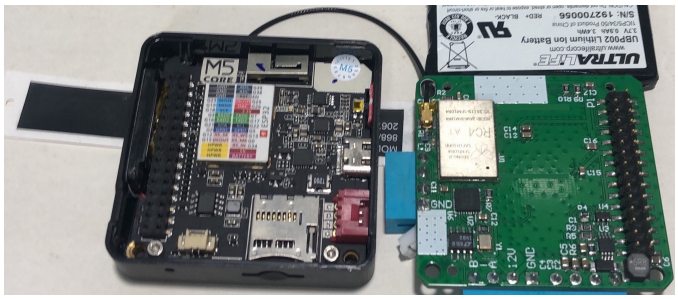

Fig. 1: Internal view of the MkIoT hardware platform.

MkIoT is composed of two main parts, as shown in Fig. 1: i) MCU module and ii) expansion daughterboard. The MCU is based on Espressif ESP32, a popular MCU in the maker communities and have a large varieties of tutorial and examples which satisfies the ease of development criteria. The MCU module is an off-the-shelf module, the M5Sstack ESP32 basic Core IoT development kit, which is a relatively affordable and accessible MCU module. It features a dual-core MCU, which runs up to 600 MIPS, 448KB ROM and 520KB SRAM. Besides, it also able to store large amount of data locally as it supports multiple external flash chips through its QSPI bus. The ESP32 module is equipped with both WiFi and Bluetooth Low Energy (BLE), which partly satisfy the connectivity requirement. The M5Stack module also supports both wired and battery operation, which satisfy the flexible power requirement.

The more interfaces the device has, the more application domains can explore. Therefore, for the design of MkIoT, we have placed emphasis on both wired and wireless connectivity.

1) Wired Connectivity: The ESP32 comes standard with I2C, I2S, SPI, UART and CAN bus. To allow MkIoT to communicate via industrial standard communication, RS485, we developed a daughterboard to provide RS485 comm though both NXP SC16IS752IBS I2C-to-dual UART interface and LTC1480 for ultra-low power RS485 transceiver. A DC step-up regulator was also added to the daughterboard to provide a 12V power rail to industrial device powering off MkIoT.

2) Wireless Connectivity: In order to support LPWAN, a Sigfox transceiver, Wisol SFM10, was included in the daughterboard design. There are two SFM10 transceiver that are tested in our work, which are RC3 for Japan and RC4 for Asia Pacific countries, such as Malaysia. The development and testing was done first in Ishikawa, Japan and later in Kuala Lumpur, Malaysia, and in both cases E2E communication from MkIoT to the Sigfox cloud was successful. One of the drawbacks observed was the absence of signal indoors, rendering it useless in such an environment. The u-blox SAM-M8Q GPS module was also included to support asset tracking use cases.

IV. Secure End-to-end

For our use cases, we implemented an end-to-end (E2E) application prototype in which the IoT device sends data to or receives data from the cloud. Hence, focus is placed on the device and E2E communication integrity and security. Below we explore the architectures and security practices that are relatively straightforward for a maker to implement, and analysed the pitfalls encountered throughout the process. This analysis is done under the assumption that the public cloud provider implements best security practices to safeguard their service and infrastructure [6].

A. Securing Communication via Sigfox and MQTT

In order to implement secure E2E communication, we need to ensure transaction-level security by measuring integrity of the entire system and transaction, E2E. For secure E2E communication, we implement it over LPWAN and IP network to the public cloud endpoint.

1) Sigfox Security: Sigfox is a lightweight protocol, suitable for very-low power remote sensor IoT application. Some use cases for Sigfox are remote sensing and asset tracking. The lightweightness of Sigfox also constrains the application to 12-bytes payload for uplink and 8-bytes payload for downlink. Besides, there is also a limit of up to 140 uplink and 4 downlink messages per day, which is still sensible for its target use case market.

For our implementation using LPWAN, two architectures were considered, where their endpoints are: i) Sigfox cloud and ii) Google Cloud Platform (GCP) through Sigfox backend. For the first architecture, the implementation was fairly straightforward as the SFM10 Sigfox transceiver can receive AT commands over UART as long as the transceiver identity is registered at the Sigfox cloud and is authorized to communicate. Further infomation on Sigfox UID and encryption key can be found in [7]. The second architecture is actually an extension of the first, where the uplink payload is received at Sigfox cloud and a set of callbacks using Sigfox backend API is setup to forward the payload to GCP HTTPS endpoint. The transaction between Sigfox and GCP is authenticated by basic HTTP username and password over HTTPS, for which the credentials are stored in plain text on Sigfox backend. Hence, at this point, we have to assume that the services provided by public cloud providers are secure.

Sigfox frames are sent in plain text by default, although there is encryption support since 2017, which is provided by the Sigfox end [8]. However, there are no details about enabling payload encryption in the SFM10 datasheet. This restricts the makers to either utilize another transceiver or rollout their own payload encryption, which is an issue by itself in [7]. This issue is also covered in [9], where Sigfox is used only for the network security itself, and delegates the payload security issue to the makers.

2) MQTT Security: For our implementation using IP, the IoT device communicates over the IP network using the MQTT protocol with GCP. Before we dive into the implementation details, let us look into a few key points on MQTT security. Firstly, the original MQTT standard does not require authentication in MQTT as mandatory [10]. While it is possible to host an authentication-less MQTT broker in a secured network, this is not widely practised in most of the implementation tutorials and examples. Secondly, basic authentication in MQTT is not as secure as mostly thought. While authentication is supported as username and password fields in the CONNECT message, these are sent in plain text in TCP over the network, which is an eavesdropping risk. Many popular MQTT setup tutorials, especially for smart homes, are using this method, which is a risk if the maker expose the service to public network.

In order to solve this issue, MQTT can be used over TLS to encrypt the whole MQTT communication. While introducing TLS on the MQTT broker has insignificant performance penalty, that cannot be said very constrained IoT devices, where the increase in overhead processing will reduce its battery operational time. The maker has to balance the requirements in order to accommodate the overhead or settle for a less secure implementation. For public cloud provider such as GCP,TLS connection to their MQTT broker is mandatory.

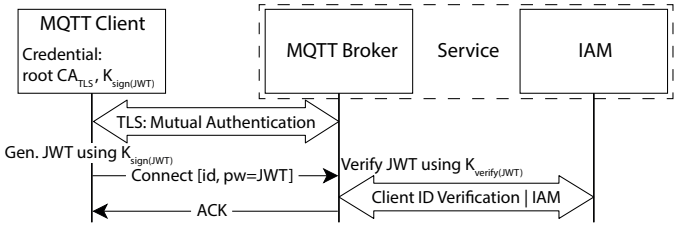

Fig. 2: MQTT communication using TLS and JWT.

In addition to TLS, other security mechanisms can be used to increase security, for instance, JSON Web Token (JWT), which is also used in addition to TLS on GCP MQTT broker, as shown in Fig. 2. JWT enables per-device authentication, which limits the attack surface as compromised key would only affect a single device rather than the whole group. Besides, each JWT is only valid to up to the user setting or maximum 24 hours, which ensures that the compromised key will expire. Google maintains a sample Arduino library to make implementation simple for makers. One issue that affected many makers using the Arduino library to connect to GCP is that there is no built-in mechanism to refresh the

JWT. This requires the maker to periodically disconnect and reconnect back to GCP with a new JWT causing session interruption, which may not be ideal for certain IoT applications as unwanted latency is introduced. Another issue that we faced during implementation is the connection rejection if the JWT timestamp does not tally with GCP side. JWT is also time dependent; hence, there is a risk of DoS as the IoT device is now dependent on NTP servers to resolve its system time.

B. Securing the ESP32 based IoT Device

The ESP32 MCU has some built-in security features, such as secure boot, flash encryption, 1024-bits OTP, cryptographic hardware acceleration for AES, hash (SHA-2), RSA, ECC and random number generator (RNG). In this section, we will explore and discuss on how to use the MCU hardware security features to secure the device integrity.

1) Arduino Core: The Arduino software platform plays a very important role in the maker culture. It enables laymen to tinker with programming and electronics with little effort due to its extensive abstraction of the hardware layer into simple functions to control the MCU's I/O and perform computations. Arduino platform is also famous for its extensive software libraries, which are contributed by the open source communities. Arduino also provides an IDE to develop applications, direct firmware build and upload to the ESP32 device from the user friendly IDE. While these may be positive from a ease of development standpoint, it may also quickly turn into a security nightmare. There is a number of vulnerabilities that make Arduino easily exploitable as presented in [11]. Reference [5] also includes a case study on the Arduino vulnerability impact on IoT devices.

2) Locking Down Arduino on ESP32: The ESP32 MCU is often paired with an external flash chip that stores the user applications and data. Hence, any maker with physical access to the ESP32 based device could read the flash chip content via serial or desoldering and reading the physical flash chip directly through SPI. Firmware reverse engineering and modification could be done if the device maker does not secure the device sufficiently. Fortunately, ESP32 features hardware-based flash encryption and secure boot to prevent unwanted flash accesses. There's a caveat, however, as it requires the Espressif integrated development framework (ESP-IDF), which is mainly low-level embedded C, to enable the security features. Since the prototype implementations were developed using Arduino, we decided to keep the Arduino code, as it is difficult for a non-technical maker to jump from Arduino into embedded C. One workaround is to use Arduino core as a "component" in ESP-IDF as shown in [12].

We ported the Arduino code using the workarounds in [12] to ESP-IDF, but many issues had to be ironed out, such as external SPI RAM failing to initialise, broken links to BLE, missing TLD setting and many more. Besides, the Arduino core had to be compiled to specific version of ESP-IDF, which were outdated (v3.x) compared to the latest stable (v4.x), missing out certain security features and bug fixes. Enabling the security features in ESP-IDF is trivial as shown in Fig. 3.

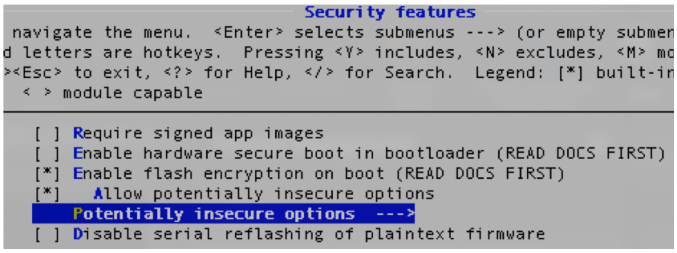

Fig. 3: ESP-IDF security features.

The ESP32 bootloader, ported Arduino application code and custom partition table from Arduino minimal SPIFFS partition scheme were prepared. Then, an encryption key was generated and burned into the ESP32 device eFuse as shown in Fig. 4.

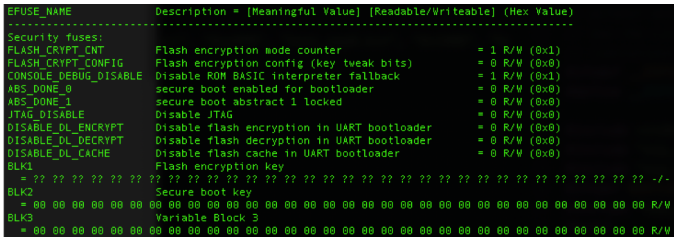

Fig. 4: eFuse after enabling flash encryption and burning the encryption key into the eFuse BLK1 register.

The unencrypted bootloader, partition table and application are then written to the ESP32, which will auto reset and encrypt the bootloader, application partitions and any partition that is flagged as encrypted (see Fig. 5).

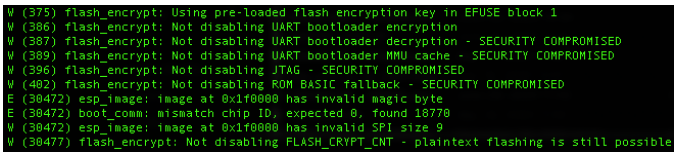

Fig. 5: ESP32 device security issues log on first boot.

While Fig. 5 shows some security warnings regarding insecure configurations, we did not fully lock down the ESP32 device as it would limit our ability to further analyse the flash after encryption. Moreover, if there are any errors during flash encryption procedures, the ESP32 may be bricked, rendering the device useless.

3) Flash Chip Content: Next, we will analyse the ESP32 flash chip for both unencrypted and encrypted scenarios. For ESP32 series, all except ESP32 pico have the firmware stored on an external memory chip. We read the flash chip content through ESP32 serial for both scenarios, where a sample of the bootloader before and after encryption is shown in Fig. 6.

However, through analysis, we found that not every partition is encrypted. By default, only the bootloader, partition table, OTA data and App related partitions are encrypted [12]. The prototype code that performs secure E2E communication utilized the Arduino EEPROM library to store its private keys

Fig. 6: Extracted binary (bootloader section) before and after flash encryption.

in the flash. The earlier version of the EEPROM library reads and writes contents into a fixed partition block where the recent version emulates the read write functions and store the data in the non-volatile storage (NVS) partition. NVS partition is not directly compatible with flash encryption while fixed EEPROM partition block is unencrypted by default. Hence, the application private keys can still be read as plain text, as shown in Fig. 7.

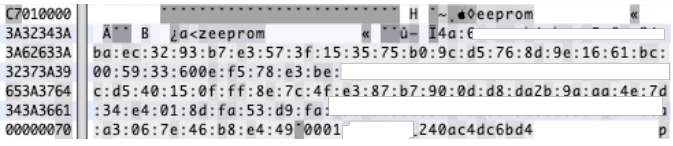

Fig. 7: Plain-text private keys in EEPROM after encryption.

Fig. 8 shows the encrypted, unencrypted and unencryptable partitions to the Arduino minimal SPIFFS partition scheme. While it is possible to encrypt the EEPROM partition by setting the encryption flag, the Arduino EEPROM library does not support such operations as its uses esp_partition_write() to write data into the partition instead of the required esp_rom_spiflash_write_encrypted(). For NVS based EEPROM case, one method to encrypt the private keys is to encrypt them in software before storing them into the NVS.

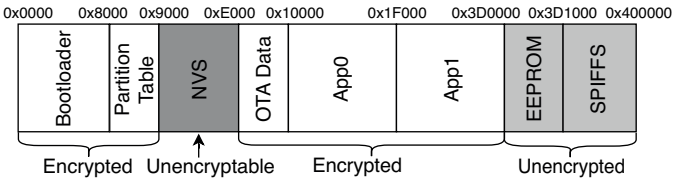

Fig. 8: Encrypted and unencrypted partitions in ESP32 device.

Securing the device and communication is just one part of the puzzle. In the next section, the process of securing the cloud for IoT life-cycle management is discussed.

V. IoT Life-cycle Management

Implementing security measures according to current best practises does not guarantee the IoT device will be secure for the rest of its operational lifetime. Manufacturers must have the capabilities to rollout bug fix and security patches efficiently as new security issues are discovered. Even for incidents such as unpatchable exploits (e.g., ESP32 fault injection vulnerability [13]), the capability to decommission the IoT device so as to minimize the damage is essential. Next, we discuss about an IoT life-cycle management implementation with Google Cloud Platform (GCP) for the provisioning, service, maintenance and decommissioning phases.

1) Provisioning: Since the introduction of [14], devices that are able to connect to the Internet must not have default passwords or even if there is a default password, it must be able to generate a new means of authentication before full access is granted to the device. While this legislation does not limit device manufacturers to implement a fixed set of measures, it increases the complexity of general device provisioning. This may require secure facility for key generation and provisioning procedures [15].

We developed and implemented a streamlined IoT device provisioning to GCP. The device provisioning tool can be deployed on any platform that supports Python, ESP-IDF and Google API python client. A Raspberry Pi 3 was setup to perform firmware programming, key generation and GCP device registration, while the GCP Cloud IoT Core is setup as the MQTT gateway and device manager. First, the "blank" ESP32 device is connected to the Raspberry Pi 3 through serial communication, where initial check will be done to verify that the connected device is an ESP32 device before uploading the pre-encrypted firmware into the device. Three 256-bits ECDSA PKI key pairs are generated before the private keys are send through serial to the ESP32 device. Although ECDSA is relatively new when compared to the RSA encryption standard, the resulting keys are smaller and cryptographic processing is faster with ECDSA, which is suited for constrained IoT devices. Verification of the private keys are also done to verify the integrity of the keys as low voltage serial communication may be affected by electrical noise from the surrounding during hi-speed transmission. A magic packet will then be sent to trigger private key install mode on the device and the tool will provision the public keys and device ID on GCP.

2) Service: The service phase includes the capabilities to manage the device, monitor and remote command and control from the cloud. Most cloud platforms provide the basic functionality to manage individual device or a group of devices with integrated event logging. In our implementation, Google IoT Core logs the event according to registry wide default policy or per device into its platform monitoring suite. Events such as MQTT heartbeat, errors can be selectively directed into user-created log sinks such as Pub/Sub, which can in turn trigger a Cloud Function to handle the event. We implemented a monitoring function that parses log file for the MQTT heartbeat to determine when a device was last seen, and alert the user if it is "missing" after a period of time; the alert is sent to the user's Slack messaging app via the messaging bot webhook API.

3) Maintenance: Firmware over-the-air (FOTA) update is part of the maintenance phase. The Arduino Core provides FOTA update over HTTP, which enables ESP32 to download

firmware from a target server into its App0/App1 partition and switch to the updated firmware. The FOTA update can be implemented in two ways: i) the device periodically polls the target server to check whether a newer firmware is available, where it automatically updates itself or require the user to approve the update, ii) the device is notified of new firmware through the cloud as a command and updates itself. We utilize the remote command function to push the update command to the IoT device as it allows flexible implementation of user notification about updates and permissions. The GCP Cloud Storage was used to store the firmware binaries. Moreover, a GCP Cloud Function was set up to trigger on file upload into the bucket, where proper identity and access management (IAM) parameters is set and the Cloud IoT Core device registry is updated with file URL. The URL is then sent to device as a command through Cloud IoT Core. This allows the device to securely receive the target URL through a secured channel instead of polling a predefined address, which is less flexible.

After the device receives the FOTA command, it must verify that the URL endpoint supports HTTP and type (application/octet-stream). Content length is verified to ensure that the application partition is able to store the new firmware. The OTA data partition is updated for the next reboot to run the new firmware after it is downloaded and verified. If the new firmware crashes after a FOTA update, rollback can be enabled to ensure the device working state. However, from a security perspective, anti-rollback is often implemented to prevent exploit by downgrading to vulnerable firmwares. Another issue we faced while securing FOTA was the firmware IAM permission in GCP Cloud Storage. Our implementation allows public read access to the firmware, which is a security risk as the firmware binaries are unencrypted. Hence anyone with the binary URL could download the firmware from the Cloud Storage for reverse engineering. GCP Cloud Storage only provides OAuth method for authenticated access, which is impractical for embedded devices. Even methods such as using API keys do not serve as an authentication method, as the key is only used for accounting purposes, such as bandwidth usage for that key. Thus, such limitation should be considered when implementing IoT life-cycle management.

4) Decommissioning: Decommissioning an IoT device from the cloud can be done in several ways, including but not limited to decommissioning via public key expiry for each device on the cloud or via disabling access to the cloud gateway. Most cloud providers provide options for setting public key expiry, disabling and deleting the device from its registry. Mass decommissioning or a temporary disable can also be swiftly rolled out via API in case of a massive cyber-attack.

VI. Conclusion

In this paper, we presented the challenges of implementing a secure IoT application from a maker perspective. The development process of an IoT application, and the functional and non-functional design requirements were meticulously described. An IoT hardware platform, MkIoT, was designed and developed to facilitate secure IoT development for makers. We demonstrated the efforts necessary to secure IoT implementation, including communication, device and the cloud. For E2E communication, we covered both LPWAN Sigfox and MQTT to public cloud implementations. Issues with Sigfox payload encryption and securing MQTT session were investigated and discussed. Furthermore, for the device, maker-friendly Arduino code had to be ported into ESP-IDF in order to implement flash encryption. We have also shown that a flash encrypted ESP32 device may not be fully encrypted and discussed the pitfalls in ensuring the device is fully secure. Finally, we implemented a life-cycle management for the IoT device using public cloud. In each phases, implementation details and security implications were discussed in detail.

References

[1] C.-Y. Yeh, Y.-M. Cheng, and S.-J. Lou, "An internet of things (iot) maker curriculum for primary school students: Develop and evaluate," *International Journal of Information and Education Technology*, vol. 10, no. 12, 2020.

[2] D. Sarpong, G. Ofosu, D. Botchie, and F. Clear, "Do-it-yourself (diy) science: The proliferation, relevance and concerns," *Technological Forecasting and Social Change*, vol. 158, p. 120127, 2020.

[3] D. De Roeck, K. Slegers, J. Criel, M. Godon, L. Claeys, K. Kilpi, and A. Jacobs, "I would diyse for it! a manifesto for do-it-yourself internet-of-things creation," in *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*, 2012, pp. 170–179.

[4] A. Morris and N. Lessio, "Deriving privacy and security considerations for core: An indoor iot adaptive context environment," in *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*, 2018, pp. 2–11.

[5] A. A. Gendreau, "Internet of things: Arduino vulnerability analysis," *A Primer for Security*, p. 32, 2016.

[6] W. M. Stout and V. E. Urias, "Challenges to securing the internet of things," in *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2016, pp. 1–8.

[7] R. Fujdiak, P. Blazek, K. Mikhaylov, L. Malina, P. Mlynek, J. Misurec, and V. Blazek, "On track of sigfox confidentiality with end-to-end encryption," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 1–6.

[8] Sigfox, "Sigfox technical overview," 2017.

[9] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-range communications in unlicensed bands: The rising stars in the iot and smart city scenarios," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, 2016.

[10] M. Calabretta, R. Pecori, M. Vecchio, and L. Veltri, "Mqtt-auth: A token-based solution to endow mqtt with authentication and authorization capabilities," *Journal of Communications Software and Systems*, vol. 14, no. 4, pp. 320–331, 2018.

[11] C. Alberca, S. Pastrana, G. Suarez-Tangil, and P. Palmieri, "Security analysis and exploitation of arduino devices in the internet of things," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 437–442.

[12] Espressif. To use as a component of ESP-IDF. [Online]. Available: https://github.com/espressif/arduino-esp32/blob/master/docs/esp-idf_component.md

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

[14] California Legislative Information. Sb-327 information privacy: connected devices. [Online]. Available: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180SB327

[15] Amazon Web Services. [Online]. Available: https://www.slideshare.net/AmazonWebServices/gpstec318iot-security-from-manufacturing-to-maintenance