Onboarding and Software Update Architecture for IoT Devices

by

Hemant Gupta

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the
requirements for the degree of

Master of Computer Science

in

Ottawa-Carleton Institute for Computer Science
School of Computer Science

Carleton University
Ottawa, Ontario

August 2018

Abstract

There has been a continuous growth in the usage of IoT devices. These devices are subject to an increasing number of attacks which exploit their software vulnerabilities. We need a secure architectural design for managing and using cryptographic keys involved in both initial configuration (onboarding) and secure automatic updates of IoT devices to perform authenticated key management and digital signature. Low-level IoT devices (8-bit) have low computational capabilities and a small memory size and are challenged to carry out desktop- and server-type public-key cryptographic operations, as needed for key establishment and authentication of software updates. We have designed and implemented a prototype to provide secure onboarding and update architecture and associated protocols for low-level IoT devices (8-bit). It uses elliptic curve cryptography (Curve25519), authenticated key establishment, and a known continuity-based key-locking mechanism that uses a public key embedded in a current software image to verify the signature on the software update. The design addresses the scenario of transfer of update authority, e.g., when a manufacturer or update provider ceases to provide ongoing software updates upon going out of business.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Paul van Oorschot of Carleton University for his support, helpful guidance, insight and feedback on my research, which acted as a true encouragement for me throughout my thesis journey. Without his guidance, this thesis would not have been possible.

I would also like to thank Dr. Anil Somayaji of Carleton University for his continuous help guidance and valuable comments. I would also like to thank members of the Carleton Computer Security Lab (CCSL), notably Christopher Bellman and Reza Samanfar, who helped a great deal by providing valuable feedback on my research.

I am also thankful to my committee members, Dr. Carlisle Adams and Dr. AbdelRahman Abdou, for providing insight and valuable comments.

Finally, I would like to thank my friends and family for their support and understanding throughout the process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growing network of connected devices called the Internet of Things (IoT) offers a vast array of security vulnerabilities. The security of IoT devices is vital, as current IoT devices, through interaction with the physical world, have the potential to cause physical harm not to mention putting at risk personal and medical data, which elevates the importance of means to provide secure software and firmware updates. Arguably, every time a company deploys a product as part of a global ecosystem, they have a responsibility for providing security updates for it. This is not so easily done in an IoT world of heterogeneous devices from a vast number of vendors.

The Gemalto Survey [32], conducted in November 2017 which involves countries like the US, UK, France, Germany, India, Japan, Australia, Brazil, South Africa and Middle East countries, found that many people are in favour of government regulation for IoT security. According to the survey, 61% of businesses want regulations to clearly define who is responsible for securing IoT devices and data at each stage. 55% of businesses also want to know the repercussions of non-compliance. In general, most organizations (96%) and consumers (90%) surveyed were in favor of government-enforced IoT

security regulation.

Many IoT devices continue to be protected only by a default password. Recently a law was passed in California requiring that IoT device [46] manufacturers provide a unique password for each device. While it is a step in the right direction, the assurance of secure software update for all low-level IoT devices remains an open problem, as is trust establishment between devices with no user input/output interface.

1.1 Problem Statement

Internet of Things (IoT) devices are deployed on many platforms like Amazon Web Service, Google Cloud Platform, etc. Many of them are low-level constrained devices [13]. A few IoT devices are capable enough to run a general purpose operating system. On the other end, we have devices which are battery operated and perform a single functionality, these are considered here as Class IV devices (later discussed in Chapter 2). Secure software update for Class IV level IoT devices is very difficult as they have no user input/output interface and no previously shared secret with another device. These IoT devices have low processing power and a small memory just sufficient enough for dedicated tasks. It is challenging to deploy public-key cryptography and to deliver software updates through the Internet for these constrained devices [13]. Secure software update requires an establishment of a communication channel between all the components involved. Also onboarding [33] is termed as a process of introducing a new node in a trusted environment such that it has suitable keying material to communicate with one or more trusted nodes.

In this thesis, we design and prototype an architecture for onboarding and secure software update of low-level IoT devices (8-bit).

1.2 Motivation

In 2016, Mirai botnet [2] exploited default access credentials to gain control of IoT devices and then used them for a DDoS (Distributed Denial of Service) attack. In the future, we expect that there will be many attacks that take place due to software vulnerability present in IoT devices like integer overflow [22], buffer overflow [21], and SQL injection [23]. In 2015 [106], similar attacks took place on hard disk firmware. Malicious code was inserted by reflashing with the help of a hacking tool. In 2013 [24], researchers demonstrated an attack on printer firmware by utilizing a design flaw in the remote update functionality.

As per our knowledge, most IoT devices are password protected, and we must change our default password immediately because of the attacks like Mirai botnet. Secure software update for low level IoT devices is still a big issue, another problem is trust establishment with an IoT device that has no user interface. Low-level IoT devices have low processing power and small memory which is just sufficient enough to perform the dedicated tasks. It is challenging (typically due to computational requirements) to use security protocols involving public-key cryptography and to deliver software updates via the Internet. Due to the increase in attacks on IoT devices, we need a method to provide software update to the IoT devices automatically whenever updates are available.

1.3 Solution Overview

In this thesis we have developed a method for onboarding IoT devices with secure key transfer and authenticated key establishment for Class IV IoT devices. This is a primary step in achieving secure software update where we use a digital signature to provide data origin authentication and the integrity of the update provided for IoT devices.

Figure 1.1 shows our basic architecture. It involves four components:

1. a Class IV IoT device (see Table 2.1, page 9)
2. a Software Update Provider (i.e., SUP which provides software update for devices)
3. a Class I level (see Table 2.1, page 9) Gateway device (to help IoT devices communicate with SUP)
4. a smartphone application (i.e., SmartApp, used for device registration with the SUP)

1.4 Contribution

We base our solution on a common architecture involving four components as listed immediately above: an IoT device, a gateway, a smartphone application, and a software update provider (SUP). Our design for secure software update uses a mechanism called "key-locking" from the method of Wurster et al. [105] [97] for self-signed binaries for smartphones. Using this, we offer the following contributions:

- We implement and test the performance of public-key cryptographic algorithms on 8-bit micro-controllers for providing secure software update.
- We apply the concept of key-locking [97] to the problem of software update on 8-bit IoT devices.
- We explain and prototype a method for onboarding IoT devices using authenticated key management involving a Gateway device and a smartphone.
- We address the issue of changing the authority of ownership of software update provider, in the case that a manufacturer or software update provider goes out of business.

Results of this thesis have appeared at peer-reviewed conference [36].

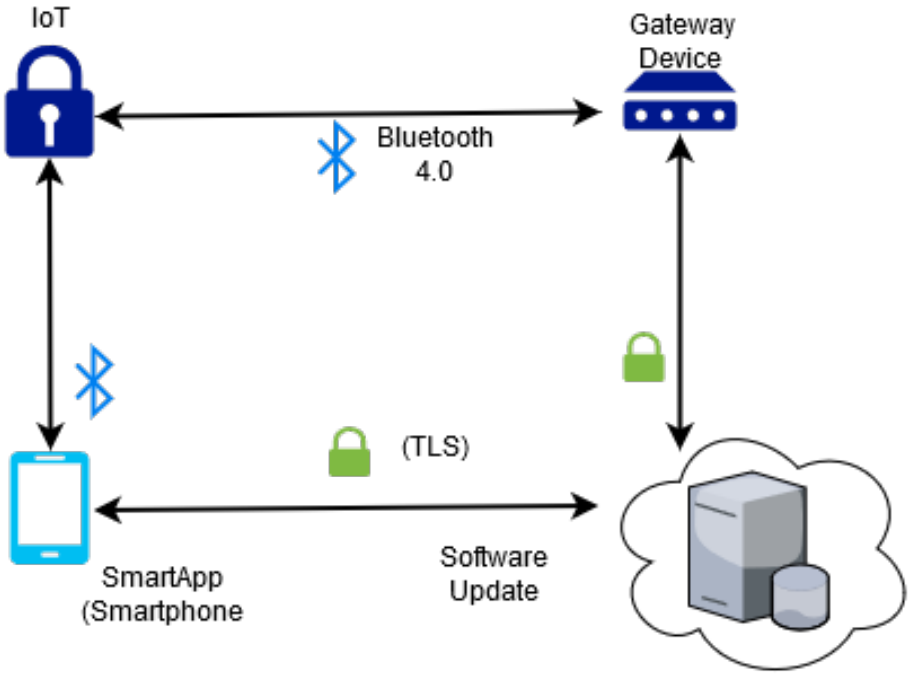


Figure 1.1: System Design for Onboarding and Software Update. Smart Lock [42] is an example IoT device

1.5 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 provides a brief background on IoT, software update, trust, authentication, cryptography, digital certificate and Bluetooth Low Energy (BLE). Chapter 3 summarizes related work in the area of software update and onboarding for IoT devices. Chapter 4 describes the architecture and overview of the system design. Chapter 5 presents the implementation details of a prototype. Chapter 6 provides evaluation, security analysis, limitations of the design, concluding remarks, and suggestions for future work.

Chapter 2

Background

In this chapter, we provide background on the algorithms and protocols used.

2.1 Internet of Things

The IoT is an extension of the Internet into the physical world for interaction with physical entities from their surroundings [107]. IoT is also known as the network of physical devices, vehicles, home appliances and other devices embedded with software, sensors, actuators, and connectivity, which enables these objects to connect and exchange data. It allows machines and real-world objects to connect, communicate, and interact with one another. An IoT device can communicate with other IoT devices through different means including Bluetooth Low Energy (BLE), ZigBee [77], or other wireless technologies. IoT services facilitate the integration of IoT entities into the service-oriented architecture (SOA) world [1]. IoT services are the transac-

tions between two parties: service providers and service consumers. IoT has many applications in the field of consumer and home devices, such as the development of smart infrastructures like the smart home and smart grid, and security and surveillance. Currently, the use of wearable and portable IoT devices is increasing in personal fitness rather than the healthcare sector, and in transportation, people are working on autonomous vehicles.

Borman [13] classified constrained IoT devices into three classes based on the size of memory (i.e., RAM and Flash), ranging from 10KB to 250 KB. We extend his classification in Table 2.1 based on our integration of research literature and white papers. In Table 2.1 on page 9, we classify the IoT device processors into five classes based on aspects of architecture [37] [95] [73] [79] including RAM (Random Access Memory) size (memory space available in the processor), bus size (number of bits that can be transmitted at a time), clock frequency (number of pulses per second defining machine cycles to complete an operation), and power consumption. Here, NA means that the feature does not apply to the current class of devices. Another feature in the table is "OS supported "[37]. There are operating systems designed for all kind of processors. Lightweight OS's for 8-bit and 16 bit IoT devices help them to function properly. Another feature is "Programming Language Supported"[37] which indicates that the operating system supported by the processor is written using these high-level languages, and these programming languages tools are supported by these IoT platforms. The last feature is "Asymmetric Cryptography Supported"[95] [73] [79], which shows a device is capable enough to perform encryption/decryption and digital signatures using public key cryptography. Class III and Class IV IoT devices can support a few of the NIST recommended elliptic curves for asymmetric cryptography according to our understanding. In our prototype we are using Class IV devices, as Class V consists of IoT devices which are programmable

only once.

Class/ Features	**Class I**	**Class II**	**Class III**	**Class IV**	**Class V**
Bus Size (in bits)	32 or 64	32	16	8	4 and 8
RAM Size	MBs to GB	KBs to MBs	10KB-1024KB	128B - KBs	128B to 8KB
Wi-Fi Supported	Yes	Yes	No	No	No
Clock Frequency	250 MHz - 400 MHz	80 MHz-180 MHz	4 MHz-80 MHz	128 Khz-16 MHz	32 kHz-8 MHz
Power Usage	10 mWatts to Watts	uWatts	$<$ 1 uWatt	$<$ 1 uWatt	unknown
OS Supported	Linux, Windows, noOS	Contiki, eCos, nuttX, mbedOS, embOS, noOS	Contiki, eCos, nuttX, TinyOS, embOS, noOS	Contiki, nanoRX, nuttX, TinyOS, embOS, no OS	NA
Asymmetric Crypto Supported	Yes	Yes	Yes (EC—-a few curves)	Yes (EC—a few curves)	NA
Programming Language Supported	C, C++, Python, JavaScript GO	C, GO, C++, Python, JavaScript	C, C++, Assembly	C preferred, C++, Assembly	Assembly
Example hardware (processor)	Raspberry Pi 3B+ (ARMv8), Beagle Bone	Arm Cortex M3, ARM7	AVR16, PIC24F	AVR8, AT89C51, tinyAVR	AMD Am2900, Atmel MARC4

Table 2.1: Extended Taxonomy of IoT Device Processors

2.2 Software Update

Another vital part of IoT security is software updates. Software updates fix features that are not performing as intended, add software enhancements, and most importantly, address security issues [94].

Software update mechanisms for the Windows operating system (e.g., Authenticode [54]), Linux, and Mac have common options:

1. Automatic update (download and install)
2. Automatic update (download, but ask the user permission before the installation)
3. Manual update (download and install)

End-users commonly opt for the first of these [99]. One problem with these scenarios is when the application vendor stops providing updates. For example like a company is out of business. In such a scenario, the outdated software is prone to attacks. In the case of manual downloads, downloading updates from a random website is also dangerous.

Software updates for smartphone operating systems like Android or iPhone are issued over the air (OTA) by the manufacturers and software update providers (SUPs). OTA is a way to keep the end user's software updated, including protection from malicious attacks [67][68]. Zhu [108] proposed three modes of OTA updates for IoT devices, similar to those noted by Barrera [8] for smartphone applications.

1. Client-initiated (i.e., pull model), where the client queries the server for updates periodically, when the new update is available, the client downloads the manifest data and image. When the device is in an IDLE state, the client installs the update. It is similar to a manual update as the client is the one initiating the request for download and installation.
2. Server-initiated (i.e., push model) where the server pushes the firmware

image to the device and updates it based on the status of the device. It is similar to automatic updates as there is no user involvement.

3. Negotiated, where the server notifies the client about the available new firmware image, and then the client decides whether they want to install the update or not. It is very similar to automatic updates where the system needs user permission before the installation.

Researchers from both academia and industry seek a standardized mechanism for software and firmware upgrades for constrained devices.

2.3 Onboarding

Onboarding [33] is a process of introducing a new node in a trusted environment such that it has suitable keying material to communicate with one or more trusted nodes. Onboarding can be performed between two devices and a device and an application. Traditional onboarding involves a pre-shared secret between devices or different methods of device pairing [55] and no public-key infrastructure for low-level IoT devices as traditional public key cryptographic algorithms (e.g., RSA) are computationally expensive. IoT onboarding depends on the end-user to physically connect the device to available infrastructure and the Internet, and to configure the device properly to the associated system. Technology is moving towards auto-onboarding, like Intel's secure device onboard service which uses a "zero touch model" [25], in which the user scans the QR-code and connects the device to the Internet, enabling the setup and activation of the product to be completed automatically.

2.4 Authentication

Authentication is the process of verifying the identity of a user or an entity [65]. There are three basic ways of performing authentication. It can be provided using something they know like passwords, something they have or possess like a hardware token holding a key, or something they are, like a biometric signature [65]. Passwords are a widely used form of authentication, but using only a password is a relatively weak form of security. NIST has provided many guidelines for a user-chosen password [35]. Many issues arise with password authentication, such as users not updating default device passwords. A recent example of malware that exploited weak authentication and access control is the Mirai Botnet [2]. Many companies and banking sites are utilizing two-factor authentication, which includes a password and a one-time authentication token generated by the service provider and shared with the end-user through a cell phone message or electronic mail. Tokens issued are only valid for a limited time and can only be used once. NIST also suggests using multi-factor authentication, which may include biometric impressions like a thumbprint and retina scan [34].

2.5 Hash Function

A hash function takes message data as input and gives a fixed length output known as a hash value.

Hash values are used to check the integrity of the data. Commonly used hash functions are SHA-1, SHA-256, SHA-3 [65] [39]. The message authentication code (MAC) is another mathematical function, which is used for data

origin authentication and can be calculated with the help of a message and the shared secret between sender and receiver. In MAC, we assume that the shared secret remains between the sender and receiver [65].

2.6 Cryptographic Algorithms

"The Art and Science of keeping messages secure is cryptography" [65]. It is done with the help of cryptographic algorithms. These mathematical functions are used for encryption, decryption, and digital signatures. These algorithms are used to maintain the confidentiality, authentication, integrity, and non-repudiation (assurance of someone performing an operation, which they cannot deny) of the plaintext message. Encryption and decryption are two related functions. Encryption (E) is the process of encoding the plaintext message (M) into ciphertext message using a secret. Decryption is the process of decoding the ciphertext message (C) into plaintext message (M) using the same or different secret. Authorized parties who have the shared secret (i.e., key) with them can participate in this process.

$$Encryption : C = E_K(M)$$

$$Decryption : M = D_K(C)$$

Here K refers to the key used for encryption and decryption.
We have two types of cryptographic algorithms: symmetric and asymmetric algorithms.

2.6.1 Symmetric Algorithm

In Symmetric algorithms, the key used for the encryption and decryption process is the same. Security of the algorithm depends on the strength of the key and the medium through which it is shared between the sender and the receiver. It might be through an out-of-band channel, or it can be calculated through a key agreement protocol like Diffie-Hellman. K above now refers to the symmetric key.

Symmetric algorithms are categorized in two different parts: the first one is the algorithm that works on one bit or byte at a time on the plaintext, these are called stream ciphers, like Rivest Cipher 4 (RC4). The other is an algorithm that is applied on a block of bytes; these are called block ciphers like Advanced Encryption Standard (AES) and Data Encryption Standard (DES). AES has a block size of 128 bits. It also supports a key size of 128, 192, and 256 bits. Longer key sizes make it harder for an adversary to succeed in a brute-force key-guessing attack. AES has different types based on the key size. AES-128 has a key size of 128 bits. AES-192 and AES-256 have a key size of 192 and 256 bits respectively. Therefore, for Class IV IoT devices ,which can only support symmetric algorithms, we use AES-256 and AES-128 bit for symmetric encryption and decryption.

2.6.2 Asymmetric Algorithm

The second type of cryptographic algorithms are asymmetric algorithms, also known as public-key cryptography. In public-key cryptography, the key used for encryption (i.e., public key) is different from the key used for decryption (i.e., private key) but both are generated using a single process.

Both public and private keys are generated by a single entity, and the public key is distributed to all the other entities who want to communicate with the first entity (i.e., the owner of the private key) with the help of a trusted third-party or by different channels. Security of this algorithm depends on two things, the first being how secure the private key is kept by the user, and the second being the size of the key.

$$Asymmetric\ \ Encryption : E_{e_B}(M) = C$$

$$Asymmetric\ \ Decryption : D_{d_B}(C) = M$$

Here, e_B : Public encryption key

d_B: Private decryption key.

Another use of asymmetric cryptography is a digital signature which is used for data origin authentication, non-repudiation, and to verify the integrity of the message [65]. The owner of the private key signs the message with its private key and sends it to the receiving party along with an original message. The receiver passes the received signature and the original message received from the sender as inputs to the algorithm along with the public key of the sender. The output of the algorithm is either true or false. Digital signature validation failure (i.e., false output) means the message has either not originated from a trusted source, or is altered in transit, or an invalid key was used.

$$Digital\ \ Signature\ \ : S_{s_A}(M)$$

$$Digital\ \ Signature\ \ Verification : V_{v_A}(S_{s_A}(M)) == True\ \ or\ \ False$$

Here, s_A: Signature Private key

v_A: Verification Public key

Rivest-Shamir-Adleman (RSA) is commonly used asymmetric algorithms. RSA is widely used for digital signatures. RSA is based on finding two large prime numbers; the security of the algorithm depends on the factoring capability of the attacker. RSA is also classified as integer factorization cryptography (IFC). Diffie-Hellman is classified as finite field cryptography (FFC) [65]. We commonly use 2048 bits of RSA keys to be considered secure. This length of keys requires large computational power and transfer time, which is not feasible for class IV IoT devices. Therefore, we use elliptic curve cryptography.

Elliptic Curve Cryptography (ECC) is defined on the mathematical structure of the elliptic curve over finite fields [38]. Elliptic Curve Diffie Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) have a shorter key size than the RSA and is able to provide the same or higher security level with better performance as compared to RSA. A 256-bit public key of the elliptic curve provides the same security as 3072 bits of RSA public key [64] [38]. ECDH is used for key agreement for establishing a symmetric key and ECDSA is used for digital signatures. In RSA, the public-key operation is inexpensive compared to the private-key operation as we can use short exponents for the public key. On the other hand, in the case of ECC, public-key operations (like signature verification and encryption takes high computational time) are more expensive [91].

2.7 Bluetooth Low Energy (BLE)

BLE [104][50] is one of the most commonly used wireless standards for IoT devices. General Access Profile (GAP) is the layer of the BLE stack used to

find the network topology of the BLE system. In BLE, when two devices are connected, they start a pairing process during which they share some information to establish a secure encrypted connection. BLE devices come with 4.0, 4.1, and 4.2 versions of Bluetooth. BLE uses the AES-CCM [103] algorithm for encrypting data. The method used by BLE devices for sharing the symmetric key is known as pairing or association. Information received during the pairing process establishment is stored in the device so that the process does not have to repeat whenever devices next try to connect next [78] [28].

In BLE, the devices exchange a Temporary Key (TK), which is used to generate a Short-Term Key (STK) used to encrypt the connection. There are four well-known pairing Methods used in BLE for sharing a temporary key between devices [12] are the following:

1. Just Works: For BLE 4.0 and 4.1 devices, the temporary key (TK), which is a constant number (size 16 bit) across all BLE devices of a particular type, is used to generate a short-term key (STK). Therefore, it is easy for an attacker to use brute-force to get the STK. There is no method for validating devices participating in the process. For devices that support BLE 4.2, ECDH is used for key establishment and to perform the key confirmation using session key. This process is still vulnerable to a man in the middle (MITM) attack.

2. Out of Band (OOB) Pairing: The TK is shared between two devices using an out of band channel like wireless technology, such as NFC. The security of this method depends on the out of band channel.

3. Passkey: During this method, the TK is a random 6-digit number which is shared between the two devices by the user. Sharing of TK is per-

formed by reading the LCD screen on one device which displays the number and entering the same number on another device using a keypad. For BLE 4.2 devices, the passkey (6-digit number) is entered in each of the devices by the user. Both devices exchange public keys and use it with the passkey and a 128-bit nonce to validate the Bluetooth connection.

4. Numeric Comparison: This method is similar to Just Works but has an extra step. Both devices generate a six-digit random value using both nonces, generated using the AES-CMAC [44] function. Both devices have (i.e., shared during key establishment) and display it for the user to verify and give approval to the connection in case of match. This method is only supported on BLE 4.2 devices.

The OOB, passkey, and numeric comparison methods require extra resources like the screen, keypad, and NFC. Just Works is simple but susceptible to MITM attacks. BLE 4.2 is used with 32-bit processors. Class IV and V devices (e.g., August and Kevo smart lock) use Bluetooth 4.0 for communication.

2.8 Transport Layer Security

Transport Layer Security (TLS) [76] is designed to provide security over the communication network. TLS provides security for maintaining confidentiality and authenticity between two devices communicating over an insecure channel. TLS protocol has two layers: TLS handshake and TLS record. The latest version of TLS is 1.3 [75]. TLS 1.3 reduces the number of round

trips required for performing a handshake by sending the client key parameters during its first handshake message to the server. In contrast, in TLS 1.2 during handshake, the client only sends the hello message to the server and the server replies with its key parameters. After verifying those parameters, the client sends its parameters to the server. Another modification in TLS 1.3 is that support of all non-ephemeral key exchange algorithms like RSA has been removed from the cipher-suites (supported by TLS 1.2). In our prototype, we use TLS 1.2 (due to the unavailability of the TLS 1.3 support in software programming toolkit, i.e., React [27] [26]).

Chapter 3

Related Work

In this section, we discuss the work done on academic and industrial research exploring IoT security related to onboarding and software/firmware update for desktops, mobile devices, and IoT devices.

3.1 Software Update

For IoT devices, secure software and firmware updates are one of the challenges faced by the industry. Over-the-air (refer OTA in Chapter 2 on page 10) is used to provide software updates for IoT devices. In a report on a software update workshop for IoT devices, Tschofenig et al. [94] discussed the challenges faced for software and firmware updates, and proposed solutions for IoT devices. The main goal of this workshop was to explore methods for secure software updates for constrained devices [13]. Tschofenig et al. [67] [68] also introduced IETF (Internet Engineering Task Force) drafts for firmware update architecture for constrained devices [8]. All of these

drafts contained some common elements for providing secure software updates, including the use of public-key infrastructure for a digital signature to check integrity and data origin authentication. They also suggest keeping long-term cryptographic keys (i.e., private key, symmetric key) in secure boot memory and maintaining some minimum level of functionality in case of an update failure. Many researchers from academia and industry are trying to standardize one or more mechanisms for software and firmware upgrade for constrained devices.

Barrera et al. [5] discusses a method of installing applications on the Android operating system by maintaining update integrity and UID (UserID) assignment. Update integrity is achieved by digitally signing the application package with at least one developer signing key. UID assignment allows developers to grant (and possible inheritance) of permissions, which help in sharing UID's that allow developers to split functionality, yet share binary resources like fonts, images, and sound clips. Barrera's model is suggested for the smartphone, which has high computational capabilities—e.g., they use RSA, which is not possible for class IV IoT devices. Since there is typically only a single application package present in case of Class IV IoT device, the concept of UID assignment is not directly of interest to our work.

Another paper paper by Barrera et al. [8] on software installations and updates presented an overview and classification of software installation models used by four of the most popular smartphone operating systems. Authors also share their advantages and drawbacks. It suggested three general models for software installation:

1. Walled garden model: Vendor has the most control over devices (e.g., Apple IOS).

2. End-user model: Smartphone vendors have no control over the software once the phone is with the end-user (e.g., Android).

3. Guardian model: Security decisions are assigned to a well-informed third-party, which makes the fundamental security decision on behalf of the end-user. In this model, the third-party can adjust the level of security as desired (e.g., Blackberry OS).

These models represent modern IoT environment-based architecture. Our prototype lies between the end-user model and guardian model. In our prototype, the end-user decides the software update provider for their IoT devices and the software update provider provides a software update to prevent the device from attacks, based on software vulnerabilities present in device.

Wurster et al. [105] proposes a method for protecting smartphone binaries already installed on the system from malicious modification using digital signatures and generalized this concept, referring to it as key-locking [97]. Public keys are embedded in the binaries to verify the integrity of the updates. A "trust on first use"(TOFU) model is used for basic initial installation. The model has a few limitations: an attacker is still able to modify any files that are not digitally signed, and this can be used to infect the machine. Also, a malicious developer can easily install a new file onto the smartphone, as well as the device, as long as the device does not have any previous information regarding the file for validation. It can allow the malicious binary to perform changes in the application, which helps the attacker to corrupt the application. Moreover, each application has its own signature verification key pair. Since many applications are running on the smartphone, there are many verification public keys available. Class IV IoT devices have small memory size

and computational power compared to smartphones, and performance may be impacted.

Barrera et al. [6] propose Baton, a modification in smartphone app installation frameworks which help application developers transfer the signing authority of an application to a new developer in a secure manner without any user intervention. This addresses the problem of renewing a signing key by chaining them. A certificate chain is a sequence of delegation tokens. A delegation token is a signed collection of metadata, which acts as verifiable support in the transition from one set of certificates to a new set of certificates. Each delegation token includes a signed hash of the delegation token prior to itself in the chain, which allows verification of the complete certificate chain. Baton's main limitation is that the developers have to include a certificate chain in all versions after the certificate transition. Also, the certificate size is from 600 bytes to 2KBs due to the RSA key sizes. If we use EC algorithms the key-sizes and signature size will be small and would make the algorithm more space-efficient . Therefore, the application size is larger with the certificate chain.

BRSKI [74] [31] provides a solution for securing the devices with zero-touch methods (enabling the device to perform their functions with no installer action except physical placement, a network connection and power cables) [100] using an installed X.509 certificate in combination with the manufacturer's authorized online and offline services. This solution is generally designed for non-constrained devices [8] such as large router platforms in data centers. It may be used for very high-level IoT devices (Class I or II). The IoT devices working on battery or energy constrained devices used in deep space exploration, wireless sensor networks, and mobile ad-hoc networks are not suitable for BRSKI. It does not describe the mechanism of se-

lecting a suitable Wi-Fi Service Set Identifier (SSID) when multiple SSIDs are available. It also does not describe how BRSKI could potentially align with authentication mechanism.

Constrained Application Protocol (CoAP) [83] [63] is a web transfer protocol used for constrained devices and networks and is used for web applications. CoAP over UDP [83] is fast and efficient due to the low overhead of the Datagram Transport Layer Security (DTLS) protocol. The CoAP over UDP is preferred for a lightweight machine-to-machine communication. However, it lacks the reliability and service guarantees offered by TCP. It also has network overhead due to many keep-alive messages. CoAP over TCP [14] reduces network overhead by sending fewer keep-alive messages, has better congestion control, and better flow control than CoAP over UDP. The use of CoAP over TCP leads to a larger code size, high network traffic (due to message flow), increased RAM requirements, and larger packet sizes. Therefore, developers need to check the device capabilities before using CoAP over TCP for IoT devices. CoAP contains a complex run-time parser, which is an easy gateway for introducing vulnerabilities. These vulnerabilities cause remote crashes of a node or provide an adversary remote access for executing code. A CoAP response packet size might be larger than the request packet, which can be used by an attacker to convert a small attack packet into larger attack packet and can cause a denial of service attack. CoAP using UDP is also susceptible to IP address spoofing attacks due to the lack of a handshake. CoAP using web sockets is susceptible to SQL injection attacks. Due to these limitations, CoAP is not suitable for use in our design; devices as web applications are powerful and can support TLS 1.3.

CBOR web Token (CWT) [48] is a means of sharing claims between two parties. A claim is information related to a subject and contains a name-

value pair. It is built from the JSON Web Token (JWT). JWT is the standard security token format, uses JSON web signature and encryption. Concise Binary Object Representation (CBOR) is mostly used for IoT devices. It is a concise means of secure data transfer between two parties using encryption and signature. CWT is used with AES-128, AES-256, and ECDSA.

Weißbach et al. [102] suggested a solution for updating distributed IoT nodes providing the same services by introducing middleware. According to the authors, middleware handles the coordination of software update on several nodes with the help of three-layer architecture:

1. In the first layer, the update manager starts the process by issuing an update prescription (contains information about next software update) to a Local Update Manager (LUM).

2. In the second layer, the LUM checks the update prescription and forwards it to all local LUMs for maintaining a consistent update of the system. Either all LUMs download the update issued in the prescription successfully or none of them will.

3. The third layer is the application layer for user interaction and for controlling the functionality of local dynamic software updates.

According to the authors, this architecture helps in maintaining the consistency in the software update for a set of distributed nodes to prevent mismatching in communication and loss of data. The proposed solution is suitable for the non-constrained IoT devices due to the requirement of large resources in terms of memory and computational capabilities for implementing this architecture.

Shin [85] suggested a security framework using MQTT (Message Queuing Telemetry Transport) by incorporating the AugPAKE protocol [84]. MQTT [4] is a messaging protocol designed for lightweight machine-to-machine communication. It uses a centralized broker architecture, which is based on publishing and subscribing. It ensures message delivery and eases the challenge of IoT devices communicating across firewalls. It runs over TCP using the TLS. Clients publish their data to the broker and subscribe to the broker to download the commands. MQTT automatically pushes the data to all subscribed clients. MQTT is a higher-level protocol not suitable for our architecture. Limitations of MQTT are as follows: using TCP connections limits utility for low-power devices, TLS encryption is a poor match for constrained clients because of TCP code size, and; a large number of message flow. MQTT is also susceptible to the injection of spoofed control packets. It may also allow a denial of service attack. Communication using MQTT could be intercepted, altered, re-routed or disclosed. There is one attack [17] [60] that has taken place on IoT devices using MQTT protocol where an attacker can access the open ports on a server and can see and control the coordinates of airplanes, prisons with door control, electrical meters, medical equipment, and many other systems.

Choi [20] introduced a secure firmware validation and update scheme for consumer devices in a home networking system by utilizing an ID-based mutual authentication to distribute a firmware image securely. Many IoT devices like a smartwatch, smartphone, or laptop update their firmware while moving; therefore, the firmware is divided into a series of chunks called fragments. Each fragment is used to create a hash chain [96]. Hash chaining achieves the authenticity of the fragmented firmware image. Limitations of the proposed approach are that the public-private key pair of the interme-

diate device is generated by the server. Firmware submitted by different developers and publishers is not digitally signed for maintaining integrity. The proposed solution is not sufficient for firmware security as any remote attacker can use open telnet services with no root password configured to access the device, and the attacker can cause a malfunction and inject malicious code. Using third-party libraries in firmware lowers the security level and does not protect against firmware level rootkits and malware.

Lee [58] discussed a firmware update scheme which depends on blockchain technology and fundamental properties of asymmetric cryptography. A blockchain is an expanding list of records. Each block contains a cryptographic hash of the previous block and transaction data. In the proposed solution, a blockchain is used to check the firmware version and integrity. When the firmware is not up-to-date on the device, it can be downloaded from peer-to-peer firmware sharing network of nodes. Bit torrent trackers are used to keep track of firmware copies that reside on peers. With the solution proposed by Lee, we are unable to determine whether the provided update is legitimate, faulty or vulnerable. Nodes can be compromised by a physical attack, which may lead to a firmware modification attack. The limitation of this scheme is high network traffic, and nodes require high computational power, which is not suitable for low-level and mid-level IoT devices. Verification public-keys of the nodes are pre-shared. With an increase in the number of verification nodes, there are many public keys to maintain, and it may also increase the attack surface.

Uptane [105] is the first software update framework for automobiles to address automotive-specific vulnerabilities. Uptane is built on TUF. TUF [80] is an update framework which is used to design a security system for software repositories. It uses signed metadata to detect and prevent attacks

before installation. TUF aims to limit the impact of compromise after it happens. Uptane has few a additional features like additional storage for recovery in case of software infects the Electronic Control Unit (ECU), which maintains a vehicle manifest to keep track of versions installed on different ECU's, and a time server to delay the update for an indefinite time. Uptane offers original equipment manufacturers (OEMs) two different security levels based on the security sensitivity of the electronic control unit. First, it enhances the traditional design of TUF for securing software repositories by adding another role called director. The director role is at the server level and is used to blacklist faulty and corrupted software versions. Another modification is that the OEM may delegate the signing of images to the suppliers. During an update, the primary system downloads, verifies, and distribute the metadata and images to all secondary systems in the vehicle. The Uptane software update framework requires vast resources with regards to memory, computational power, and batteries. This framework is not suitable for Class IV IoT devices.

IDIoT [7] is a model proposed based on network-based isolation and filtering system to secure IoT devices from widespread network attacks by restricting the functionality of IoT devices to specifically their necessary network behaviour. IDIoT works as an intermediate device between IoT devices and networks and enforces policies based on the tasks performed by the devices. This design is useful when a device is unaware of the second party it is communicating with.

Khan [52] discussed a methodology for transferring the ownership of the IoT device automatically by maintaining device owner profiles and performing authentication. In case of new ownership, the device sends a challenge to the trusted device. If the response is correct, the device continues performing

ownership detection. In case of no response or response failure, the device protects the old data and asks the user to create a new profile. This design is beneficial in terms of security and maintaining privacy for the selling and buying of used IoT devices.

In our proposal, we are motivated by the idea proposed by Wurster [105] of self-signed binaries. The issue in the case of constrained IoT devices is that they do not have enough computational power. Therefore, we modified the solution and made it deployable by using the single digital signature for the complete application. Another problem with IoT devices is that their lifespan is 10-15 years, and a single public-private key pair is insufficient for stopping the attacker, depending on the system available to the attacker in question. In order to solve this problem, we generate a new public-private key pair with each update. So that after 15 years, we are not still using the same signature-verification key-pair (which might mean it has become attackable after 15 years).

3.2 Onboarding

In this section, we discuss the work done in the field of device pairing methods and authentication.

3.2.1 Device Pairing

Device pairing is an important concept for trust establishment of wireless devices. Most of the IoT devices communicate over Bluetooth and use an out

of band (OOB) channel for sharing secret. But these devices require extra resources like speaker, microphone, light emitter and detector to support these OOB channels. Our design provides an solution for the IoT devices which do not have these resources to share secret. We use the conventional internet as an OOB channel to share secret between devices using a Gateway device and a smartphone.

The Resurrecting Duckling model [88] is based on the idea of trusting and pairing with the first device that makes contact with a newly 'awaken' device. If many entities are present at the time of device initialization, the first device to sent a key to the awoken device becomes the owner of the device. Trust between these two devices can only be broken in two cases: either the slave device is reset, or it dies due to battery depletion.

Another method is "Seeing is Believing" [62], which uses a visual image as an OOB channel. In this design, one device requires a camera and another device requires a display screen. The requirement of a camera or display screen is not generally suitable for Class IV level IoT devices. Another method suggested for pairing, "Blinking LEDs" [81], also uses the visual channel as an OOB communication. In this method, one device requires a light detector or camera, and another device requires at least one LED. The device with the LED sends the message by blinking the LED, and another device records the data and extracts information based on gaps between the blinking. The receiver device informs the user of success or failure, and the user informs the other device to accept or abort.

In HAPADEP [87], both devices require a speaker. For bidirectional verification, both devices send encoded cryptographic protocol messages using a slow more pleasant-sounding codec to each other using the wireless channel,

which is then verified by the user. Both devices play the audio created using the slow codec, and the user then verifies whether the two sequences match or not.

BEDA [86] is a secure pairing method that requires significantly less infrastructure and human interaction. It is based on the pairing of devices by pressing buttons. It has many variations such as "LED-Button", "Beep-Button", "Vibration-Button", and "Button-Button". In the first three variations, one of the devices will either blink, beep or vibrate, and the user must press a button on receiver device. In the case of the button-button method, the user must press the button on both devices at the same time. This protocol uses PAKE [15] and sends 3-bit block messages whenever the LED blinks from one device to another, establishing the common secret.

A few other device pairing methods involve user interaction [51]:

1. "Compare and Confirm" where a user compares the number displayed by both the devices and gives the confirmation.
2. "Select and Confirm" in which one device displays a number, and another device displays a list of numbers, the user needs to select the number displayed by the first device from the list of numbers and confirm.
3. "Copy and Confirm" where one device displays a random string, and another device asks the user to enter the same string.
4. "Choose and Enter" in which a user enters the same random number or string in both the devices.

3.2.2 Authentication

Wazid [101] designed a new secure, lightweight remote user authenticated key management protocol for IoT-based smart home architecture. This proposed solution uses a gateway node for communication between clusters of IoT devices and users with the help of three-factor authentication, i.e., the smart card, password, and personal biometrics for node and user registration to the gateway node. It uses these three factors for performing authentication and key agreement between the IoT node, gateway node, and user. The IoT node may get compromised or due to constraints, exhaust due to low battery power. This may cause transmission delay, and an adversary might be able to deploy new nodes in the network. Another limitation of this model is that user biometrics are stored on the server to allow remote access verification. In the case of server compromise, users biometric templates may be compromised.

Mahto [61] proposed a model for security improvement of a one-time password (OTP) scheme for bank transactions, using elliptic curve cryptography (ECC) with iris biometric. The iris biometric and the key of ECC mitigate the weakness in the current OTP system. With the help of user's iris, the system generates a public-private key pair, and the public key securely transferred to the OTP server of the service provider which is then used to encrypt a plaintext OTP message. At the user's end, the private key is used to decrypt the OTP, which is used for the transaction. Mahto tested this solution on the Class I level system.

Kogan [53] presented a time-based offline one-time password scheme, T/Key which is based on S/Key [56] and Time-based One-time password (T-

OTP) [98] using a secure hash chain. T/Key compensates for the limitations of S/Key as a password is chosen but is not valid for a long time and like S/Key, T/Key does not utilize the same hash function at every iteration of the hash chain. T/Key also compensates for the short-comings of T-OTP by not storing secrets on the server side, precluding access by an attacker in case of server compromise. T/Key is tested using an Android application and a laptop, which are powerful devices as compared to Class IV IoT devices.

The NIST report on lightweight cryptography [95] for 2017 mainly suggested the use of symmetric cryptography for constrained IoT devices. It placed public key cryptography in scope but suggested that it must be robust against quantum attacks and use a combination of general public key cryptographic schemes with lightweight primitives. NIST also organized a lightweight cryptography workshop in 2015, in which performance of different cryptography algorithms was analyzed on ARM-based microprocessors [35].

Tschofenig [93] proposed a model of user authentication for the IoT with the help of four logical server-side functions. Authentication and user identity management include FIDO alliance support. The server generates an OAuth token for authentication, access permission, resource directory (a discovery component for resources offered by IoT devices), and for remote access of the IoT device. Device management allows companies to remotely manage their devices when configuration changes are necessary, or software updates are available.

Ozmen [73] proposed a low-cost asymmetric algorithm for wireless IoT devices using an ECC library and the NIST-recommended secp192 curve. In his implementation, he tested the proposed solution with an 8-bit microcon-

troller using the Arazi-Qi (AQ) self-certified ephemeral scheme [3] and the Boyko Peinado Venkatesan (BPV) technique [16]. Ozmen is able to boost the performance of key-exchange, integrated encryption, and hybrid construction. Further, they make use of the concept of self-certification. The author is able to improve battery life over standard cryptographic algorithms.

Salman [79] proposed a pairing based cryptography method for lightweight hardware and software based on Barreto-Naehrig (BN) curves [9] used for a non-interactive key agreement protocol where the session key is established without prior communication. In their experiment, they were able to achieve improvements regarding latency, power, and energy over commonly used algorithms for a key agreement like ECDH.

Efficient Augmented Password-only Authentication and Key Exchange (AugPAKE) [84] is a password-authenticated protocol for Internet Key Exchange Protocol version 2(IKEv2). The AugPAKE protocol is secure against active attacks, passive attacks, and offline dictionary attacks, and provides resistance to server compromise. This protocol consists of two phases. In the first phase, the user computes the password and transfers it securely to the server for verification. In the second phase, the authenticated key establishment takes place between the user and server by hashing the binary string, which is the password shared in the first phase.

J-PAKE [40] is a password-authenticated key exchange protocol using juggling. It means juggling between two people — if we assume a public key as a "ball". In round one, each person sends two public keys to each other. In round 2, each person combines the available public keys and the password to generate a new public key and sends the new "ball" to each other. After round 2, the two parties can securely compute a common session key. It

offers a security proof which some other key exchange protocols lacks, e.g., EKE and SPEKE, and is built on an earlier mechanism of Schnorr [41]. It is also compatible with elliptic curves. It protects against online and offline dictionary attacks and maintains forward secrecy.

In our design, we use the trusted network for transferring the secret key and Bluetooth MAC-address of one device to another. With the help of Bluetooth MAC-address, the device can directly communicate with another device, which saves device time, which was usually used in searching for another device. The secret key is used for entity authentication, as a class IV device does not have any input/output interface and avoids user interaction. ECDH with Curve25519 is used for key establishment, and the secret key of the device is used for key confirmation.

Chapter 4

System Design: Architecture and Overview

A primary goal of our design is to prevent unauthorized updates of software (i.e., rogue updates that are not from a legitimate source), and to facilitate a means to ensure that updates are properly signed. Verifying the signature on a software image provides assurance that it is from a legitimate source. These software updates help prevent the exploitation of software vulnerabilities of IoT devices. In this chapter, we discuss the threats related to onboarding and software update, and an approach to robust security against identified threats.

4.1 Threat Model and Assumptions

In this section, we focus on the threats related to onboarding and software updates of the IoT devices by identifying the attacker's goals, attack sur-

face, and attacker's capabilities over the system. We discuss the different attacks possible during authenticated key management and software update for low-level IoT devices.

Attacker's Goal: We identify the goals of an attacker in an onboarding and software update scenario to cause different threats explained later:

G1: The attacker may attempt to remotely control the device.

G2: The attacker may attempt to cause denial of service attack.

G3: The attacker may attempt to access the user's private data.

As discussed previously, IoT devices contain user privacy sensitive data and authentication keys of the user's wireless network. Therefore, the IoT device is one of the primary targets of a potential attacker.

Attack Surface: In the complete system involved in onboarding and software update, IoT devices commonly store data in the local storage and at the SUP database, i.e., the cloud. The user's authentication-related information is stored in the local storage of the IoT device, Gateway device and SUP which includes user identity and device information. Device information is sent to the Gateway device through SUP. There are several attack surfaces in this case:

S1: IoT device

S2: Gateway device

S3: SUP

S4: SmartApp

S5: Communication channel between SmartApp and SUP

S6: Communication channel between SUP and Gateway device

S7: Communication channel between IoT device and Gateway device

S8: Communication channel between IoT device and SmartApp

Attacker Capabilities: We take into account two potential attackers in a software update scenario with different capabilities:

C1: An attacker who has all basic information about the device such as the IoT device model number, manufacturer, update manifest data template, and information about the software/firmware image.

C2: A former SUP who has gone rogue, has all the information related to the user's device and manufacturer's legitimate private signature key, and can use that information for a malicious software update.

Figure 4.1 shows the threat model for onboarding and software update for low-level IoT devices. We consider two possible onboarding and software update scenarios of introducing threats based on attacker's goal. The "T" items in scenarios represent the number of the identified threat.

Scenario 1: Onboarding and software update for a Class IV IoT device.

Description: This scenario includes a legitimate user establishing trust between all components involved in onboarding and sending a query to the SUP for new updates. The SUP then replies with new software update manifest data (i.e., a combination of metadata that describes the software image which includes device model number, device manufacturer company, software version, and sequence number) and software image.

T1: **Replay Attack/Old Firmware** An attacker may attempt to send an old version of the valid software image to the device.

T2: **Offline Update** An attacker targets a device that has been offline for a long time and has missed a few updates. An attacker sends a software image which has a version higher than the one installed on the device but not the latest, aiming to exploit vulnerabilities in that image.

T3: **Device Mismatch** An attacker sends a valid firmware image (i.e., all information in software image used for validation is correct except the IoT device model number and type; the same signature key is used for different device types) for a different type of device.

T4: **Update without Authentication** An attacker attempts to impersonate a SUP, sending a malicious software update that is accompanied by neither a digital signature nor a MAC. This may succeed if no method is

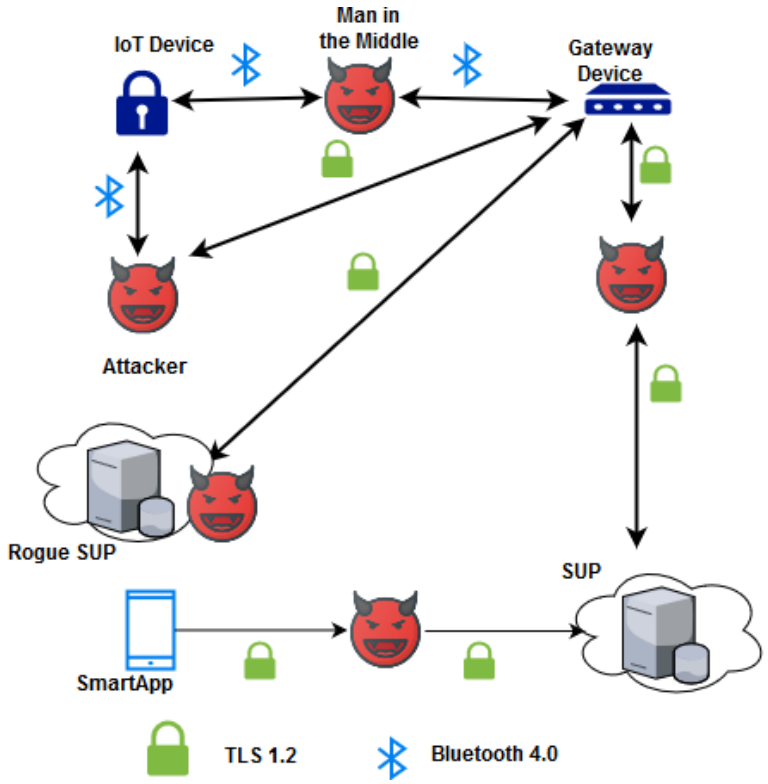


Figure 4.1: Threat Model for Onboarding and Software update

in place for verifying the data origin authentication of updates.

T5: **Man in the Middle** An attacker may attempt to spoof the Gateway device during the key establishment between IoT device and Gateway device (Figure 4.1 (1)). The attacker passively monitors the software update or modifies the authentication and access control data sent from the Gateway to IoT device.

T6: **Attacking Non-Ephemeral Keys** This is an attack aiming to exploit the use of long-term private keys, given the lifespan of IoT devices (5-15 years). An attacker aims to deduce the private key of SUP's public-private key pair using available computational capabilities, and if successful, forges a signature on a rogue update.

Scenario 2: Software Update Provider Change for a Class IV IoT device.

Description: In our model, legitimate software updates are provided by SUPs. If a manufacturer or software update provider who is providing updates to the devices goes out of business, IoT devices may continue to run on obsolete software, leaving them prone to attacks if vulnerabilities are found in the existing software image. The change in software update provider introduces a change of software update provider from the former SUP to the new one. This change of ownership can introduce threats for a device owner in the case where the former SUP goes rogue.

T7: **Rogue SUP** A former SUP, with access to the details of the user's devices and the signature key of the base image of the device, has gone rogue (i.e., is no longer trustworthy). A rogue former SUP may exploit these details to send an old software image or a malicious software update with a valid signature of the base image.

T8: **Password Guessing** If all IoT devices of the same company or model have the same default password or user-chosen weak password, an attacker may aim to gain unauthorized access to the device by guessing the password.

We make the following assumptions for our proposed design solution and prototype (notation used here is explained in Table 4.2):

A1: Each IoT device has a unique password (w_D), Serial Number (N_D), and Bluetooth MAC-address (B_D). These are in a sealed booklet or sticker (in a QR code format) which comes with the device. The password is programmed into ROM of the device during manufacturing. The unique password (w_D) is important to avoid attacks like Mirai Botnet. There is also a law passed in California, requiring that IoT device manufacturers provide a unique password for each device [46].

A2: Each IoT device generates an EC public-private key pair $(e1_D, d1_D)$ during first initialization and on any reset, which is stored in EEPROM (Electronically Erasable Programmable Read Only Memory).

A3: Each Gateway device generates an EC public-private key pair $(e1_G, d1_G)$ and RSA public-private key pair (e_G, d_G) during first initialization or reset, which is stored in EEPROM.

A4: During Gateway device configuration, the data encryption public key (e_G) is the only Gateway device parameter (in QR-code format) visible to the user and is scanned by SmartApp.

A5: SUP's all signature private keys for different IoT devices must not compromised. These are used to sign manifest data and software images.

A6: The verification public key (v_P) of the manufacturer is stored in ROM of device D during the manufacturing process. D itself can modify the verification public keys after a successful software update for the next software image verification (as will be explained in Section 4.3.2 on page 54). The base image is the same for IoT devices of the same model/type.

A7: SUP has at least one version available of the software image ($I_{T(D)}$) higher than the base image version of the supported IoT device.

A8: The manufacturer shares its software image signing key (s_P) for D's particular device, T(D), securely with SUPs.

A9: The manufacturer provides a list of trusted SUPs to the user via registered email or its website or some other trusted means which is outside of our scope. The user uses this list to check for a new SUP in case of first initialization or SUP change. The user also uses this list for downloading SmartApp of the SUP on their smartphone.

A10: The same verification public key is embedded into all instances of the device image for all IoT devices of same type/model (for a given software update) by the manufacturer.

A11: Each SUP is vetted by the manufacturer, by a process out of scope for this thesis.

A12: The process used by SUP to obtain software updates from third parties is out of scope for this research.

A13: Attacks involving the SmartApp and the SUP are out of scope for this thesis.

A14: All physical devices (i.e., IoT device and Gateway device) can only be reset manually (refer to Section 4.3.1 on page 49 for an overall effect of resetting a device).

A15: Attacks involving physical access to end-user IoT devices and Gateway devices are out of scope (targeted attacks).

4.1.1 Evaluation Criteria

The primary goal of this thesis is to develop an architecture and prototype system for an onboarding and software update. The proposed solution should be able to provide a software update automatically. Thus, we evaluate our prototype based on following criteria:

E1 **Resource-Constraints:** Ability to perform signature verification on devices that are resource-constrained (i.e., Class IV IoT devices) within a reasonable time (< 60 sec).

Challenges: Smart home IoT devices are built with limited resources. However, some security solutions require significant memory and computational capabilities. The proposed solution should be able to perform signature verification on IoT devices with limited resources within a reasonable time. For practical evaluation, we implement our prototype on a typical Class IV IoT boards available in the market, namely, Arduino Mega2560.

E2 **Robustness:** Security against all possible threats mentioned in Section 4.1 in case of software update and SUP change scenarios.

Challenges: Section 4.1 on page 36 identified several possible threats in case of software updates. The intended solution should provide robust

security against all these identified threats.

4.2 Architecture

4.2.1 Components

We now discuss the types and functionalities of different software and hardware components used in the system design based on Figure 1.1 on page 5.

- *IoT Device:* A Class IV device with no OS installed, no user input/output interface, and no visible port for configuration. We assume an 8-byte random string (a unique per-device authentication key) is available in a sealed booklet or sticker that comes with a new device, in a QR-code format. The IoT device is only able to keep one software image in memory in addition to the base image in ROM due to constrained memory size. The device is denoted by subscript 'D' in our notation. Each IoT device type is provisioned with the same base image during manufacturing (refer to assumption A6 on page 42).
- *Smartphone Application (SmartApp):* This is used on a user's smartphone to register the devices over the Internet to the SUP. The SmartApp is denoted by subscript 'A' in our notation.
- *Gateway Device:* A Class I device with a custom OS, including capabilities to handle multiple smart home devices. It serves as a communication portal for the IoT devices, which do not have capabilities to communicate over the Internet. The Gateway device communicates with IoT devices over BLE and with the SUP over the Internet through a router. There are no user input/output interfaces except one physical

port which is used to configure it with the help of a laptop or desktop by the user. The Gateway device is denoted by subscript 'G' in our notation.

- *Software Update Provider (SUP):* The SUP is responsible for acquiring and checking the authenticity of IoT software updates before sending it to the end-user. The SUP verifies the signature of the developers sending an update and runs the test suites on the software update image before delivering it to the device. SUPs store the information (Bluetooth MAC-address, password) for each target IoT device in a SUP database. The SUP generates the public-private key pair used for signing the software image updates and firmware. The Software Update Provider is denoted by subscript 'P' in our notation.

Our design involves a secure onboarding protocol with authenticated key management to establish trust between all components (SmartApp, SUP, Gateway device and IoT device). Our two devices, D and G, do not have any physical input/output interfaces and no pre-shared secret. We use the Internet for sharing the secret between the devices, as explained in Section 4.3 on page 47.

4.2.2 Context

In this section, we discuss the data structures used for the prototype implementation. We define five data structures.

- *userProfile:* This refers to the information of the user during the registration process to the SUP. It includes username, emailId and user-to-SUP-password. The username is a unique parameter in the database, and all

IoT devices are registered with respect to userProfile. The emailId is also a unique parameter, used by SUP for sending any notification to the user. The user-to-SUP-password is used for user authentication by the SmartApp and the Gateway device and is stored in a salted, hashed format in the SUP database.

- $IoTData_D$: This refers to the information about a Class IV IoT device which contains the Bluetooth MAC-address, and a unique password (w_D) with respect to the manufacturer. During the process of IoT device registration, the user registers the IoT device to the SUP using the SmartApp. $IoTData_D$ is used by the Gateway device for communicating with the IoT device and key establishment.

- $knownData_D$: This refers to the IoT device. The user sends it as part of a software update request to the SUP. It includes the IoT device model number, manufacturer information, current software version (*curVersion*), and required software version (*reqVersion*). *reqVersion* is used only in case of update failure.

- $manifestData_D$: This refers to a manifest, which means information about a specific software image intended for a particular IoT device. It includes IoT device model number, manufacturer information, software image version, and sequence number of next update (populated before sending to the Gateway device).

- $softwareImage_D$: This refers to both the metadata and binary of the software image for the IoT device. It includes IoT device model number, manufacturer information, software image version, sequence number of next update (populated before sending to the Gateway device), the next public verification key ($verNextKey_D$, used to verify the signature of next $softwareImage_D$), and the binary itself.

Table 4.1 defines the data structures, and Table 4.2 defines the notation used in the design and implementation.

Name	**Variables or Symbol Meaning**
userProfile	username (32 bytes), user-to-SUP-password (32 bytes), email (32 bytes)
$IoTData_D$	Bluetooth MAC-address (16 bytes), Manufacturer provisioned unique password (8 bytes)
$knownData_D$	Model number (16 bytes) of D, Manufacturer (16 bytes) of D, curVersion (16 bytes), reqVersion (16 bytes used upon update failure)
$manifestData_D$	Model number (16 bytes) of D, Manufacturer (32 bytes) of D, Software version (16-bytes), Sequence Number (8 bytes, populated when signed)
$softwareImage_D$	Model number (16 bytes) of D, Manufacturer (32 bytes) of D, Sequence Number (8 bytes, populated when signed), verNextKey, Software version (16-bytes), Binary

Table 4.1: Data Structures used in Design and Implementation. D refers to the IoT device

4.3 Solution Overview

In the previous section 4.1 on page 36, we discussed relevant threats to on-boarding and software update for IoT devices. We also discussed some requirements and challenges in the proposal for a secure software update of IoT devices. There are mainly two stages to consider for a secure software update:

1. Onboarding, including authenticated key establishment and sharing the unique password of the IoT device with the Gateway device.

Notation	Meaning
D	IoT device
G	Gateway device
P	SUP (Software Update Provider)
A	SmartApp
$curVersion$	Current software version running on IoT device
$reqVersion$	Required software version for IoT device (Used in update failure)
$user_{id}$	Unique identification Generated by SUP database for storing $userProfile$
T(D)	Device Type of D
$M_{T(D)}$	Manifest data image for given device model
$I_{T(D)}$	Software image of the particular IoT device model
$verNextKey_D$	Verification key of next software image of IoT device
$verCurrKey_D$	Verification key of current (i.e., running) software image of IoT device
w_D	8-byte IoT device unique password
W	Symmetric key (AES-256) derived from w_D
k	Symmetric session key (AES-256) between G and D
$K_G = K_D = K$	Long term shared secret (AES-256) generated using ECDH keys
$e1_D, d1_D$	Public-private (256 bits) ECDH keys of D from A2
$e1_G, d1_G$	Public-private (256 bits) ECDH keys of G from A3
e_G, n	RSA encryption public key of G, key size 3072 bits from A3
d_G	RSA Decryption private key of G, key size 3072 bits from A3
v_P	Elliptic curve verification public key (Ed25519 [57]) of SUP with key size of 256 bits
s_P	Elliptic curve signature private key (Ed25519) of SUP with key size of 256 bits
$H(x)$	Hash of x using SHA-256
$a\|\|b$	Concatenation of a and b
N_D	Serial number of IoT device
B_D	MAC-address of D's Bluetooth hardware
$E_K(x), D_K(x)$	Generic encryption and decryption notation, either symmetric or asymmetric according to the type of key used as subscript
V_K(x)	Signature verification of x using key K
S_K(x)	Signature of x using key K
Req(x)	Request for data x
NULL	Variable is empty
f(x)	Symmetric 256 bit key from x

Table 4.2: Notation used in Design and Implementation

2. Secure software update

We propose an architectural solution for a software update that is new, to our knowledge, with a primary focus on Class IV devices.

4.3.1 Onboarding Key Management (Chain of Custody)

Chain of custody is a concept that enables authenticated key exchange and helps in onboarding the IoT device and the Gateway device. It is similar to the concept of chain of custody of physical evidence, which is properly logged while being submitted or transferred to an authority. Here, the evidence is the information about the device's unique password, and who owns and is responsible for maintaining the integrity of the evidence before handing over or taking custody of the evidence.

By design assumption, the IoT device does not have any physical input/output interface (other than wireless) for user interaction, and the Gateway device has only one physical port for the user to configure the Gateway through a laptop or desktop. A Class IV device supports the Bluetooth 4.0 protocol; since this protocol is not secure, we need a method of onboarding that establishes a secure communication channel between the device and the Gateway. In our prototype, communication between the SmartApp, the SUP and the Gateway is over HTTPS using TLS 1.2. While TLS 1.3 is the current standard, we use TLS 1.2 due to availability of supporting toolkits, and henceforth say "TLS".

We now discuss the method of onboarding between the device and the Gateway. We use RSA [69] with a key-size of 3072 bits for sending encrypted $IoTData_D$ (w_D and B_D) from the SmartApp to the Gateway device through

the SUP (refer to Figure 4.2). There is no direct communication channel between the SmartApp and the Gateway device. Therefore, a two-party key agreement (e.g., Diffie-Hellman) protocol is not feasible. During the configuration of the Gateway with SUP information, the public key of the Gateway device is available to the user in QR-code format for scanning with the SmartApp. The SmartApp stores it in the local storage for future usage.

We are using RSA here because of the availability of the *react-native-rsa* library [72] (version 1.0.24, July 8, 2018). In practice, the Elgamal-EC [59] [43] [30] is preferred in an IoT environment for encryption and decryption in place of RSA, as it uses a smaller key-size.

We use ECDH with Curve25519 [57] for establishing the session-key (k) between the device and the Gateway to generate a shared secret (K) derived during key-establishment. We use SHA-256 for key-confirmation with k and W. We use a long-term shared secret (i.e., AES-256 (K)) for encryption/decryption while transferring a new session key from the IoT device to the Gateway device. The long term shared secret the computational burden on the Class IV device for performing public-key operations conducted multiple times a day.

Note: In our current design, we are sending the serial number (N_D) of the IoT device during registration. N_D is a redundant parameter in the current design as we are storing it in the SUP database but not using it for any validation in onboarding and software update. It is useful for future work when we are going to extend our design for storing the data of the IoT device in the SUP database with regards to the serial number of each IoT device.

We now discuss the authenticated key management using Figure 4.2 on page 59.

1. User downloads and installs the SmartApp of the SUP from a trusted source.

2. User creates their account with the SUP through SmartApp by sending $userProfile$ (which includes username, user-to-SUP-password and email address) and then logs in after successful account creation.

3. After a successful login to the SUP, the user starts the Gateway device (G) by pressing the on/off button and G starts its initialization process. During initialization, G generates an EC public-private key pair $(e1_G, d1_G)$ and an RSA public-private key pair (e_G, d_G). The user configures G using a laptop with the SUP's URL, user's username and user-to-SUP-password of the SUP account.

4. The user scans the RSA public key (e_G) of G with the SmartApp (QR code) from the laptop screen during configuration and saves the key on the local storage of the smartphone.

5. The user starts the IoT device by pressing the on/off button and D starts its initialization process. During initialization, D generates an ECDH key pair $(e1_D, d1_D)$. The user scans the 1D bar code of the serial number (N_D) and $IoTData_D$ which includes the unique password (w_D) and the Bluetooth MAC-address (B_D) of D through the SmartApp during the IoT registration from D's sealed booklet.

6. The SmartApp encrypts the $IoTData_D$ of the IoT device with the public key of the Gateway (e_G), and sends it to the SUP over TLS along with the serial number (N_D) of device. The SUP stores N_D and encrypted $IoTData_D$ in the user's queue of the IoT devices in SUP's database.
   Z = $E_{e_G}(IoTData_D)$ where $IoTData_D = w_D, B_D$
   A $\longrightarrow$ P: Z, N_D;

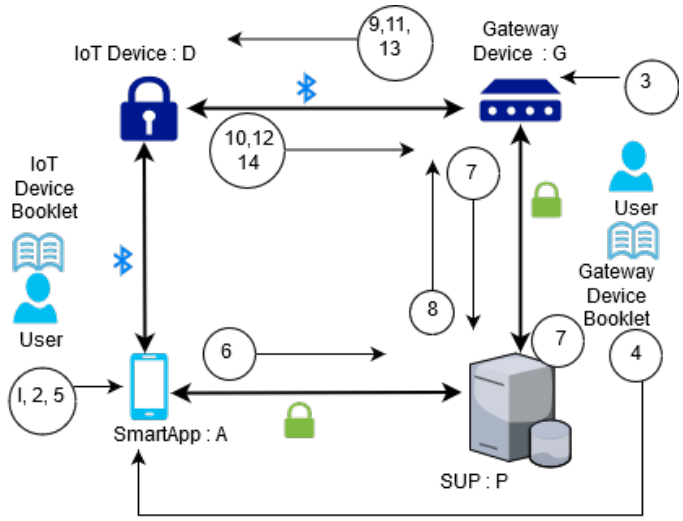
7. The Gateway device requests the next $IoTData_D$ from the SUP over TLS. The SUP checks the next available encrypted $IoTData_D$ in the queue and removes it from the user's queue of IoT devices, and stores it locally on the SUP database. The Gateway device continuously send request for next $IoTData_D$ until the complete queue is empty. The process repeats itself arbitrarily after every 24 hours to check for new IoT device information.
   G$\longrightarrow$ P: Next Z;

8. The SUP sends the stored encrypted $IoTData_D$ of the next D to G over TLS. G decrypts the $IoTData_D$ (w_D and B_D) using the RSA private key (d_G) and stores them locally on itself.
   P$\longrightarrow$ G: Z;

9. G sends its EC public-key ($e1_G$) to D over a plaintext channel using the Bluetooth MAC-address (B_D). D receives ($e1_G$) and generates a shared secret key (K_D, AES-256) using the EC public-key ($e1_G$) of G and the EC private-key ($d1_D$) of itself. It also generates a random symmetric session key (k, AES-256) and symmetrically encrypts k using K_D.
   G$\longrightarrow$ D: $e1_G$;

10. D sends its public-key ($e1_D$) and encrypted session key to G. G receives ($e1_D$) and generates a shared secret key (K_G, AES-256) using the EC public-key ($e1_D$) of D and the EC private-key ($d1_G$) of itself such that $K_G = K_D = K$. It decrypts session key (k).
    D$\longrightarrow$ G: $e1_D, E_{K_D}(k)$;

11. G sends double hash of session key (k) concatenated with W (W = f(w_D)) to D. D generates W = f(w_D) and verifies the hash and aborts in case of mismatch. This convinces D that the G knows W.
    G$\longrightarrow$ D: $H(H(k||W))$;



$Z = E_{e_G}(IoTData_D)$ where $IoTData_D = w_D, B_D$

6. A⟶ P: Z, N_D; P adds Z and N_D in the user's IoT devices queue

7 G⟶ P: Next Z; P checks the next available Z in the queue maintaining user IoT devices list and if available, dequeues it and stores it locally.

8. P⟶ G: Z; G recovers w_D and B_D using d_G

   Steps 9-10 are ECDH key establishment and Steps 11-12 are key confirmation using weak secret and session key over Bluetooth 4.0. Steps 13-14 populate $knownData_D$.

9. G⟶ D: $e1_G$; D receives $e1_G$ and stores it in EEPROM and generates the shared secret K_D using ECDH, $e1_G$ and $d1_D$.

10. D generates fresh random session key k
    D⟶ G: $e1_D$, $E_{K_D}(k)$; G receives $e1_D$ and stores it in EEPROM and generates shared secret K_G ($K_G = K_D = K$) using $e1_D$ and $d1_G$. Uses K to recover session key k. G also generates W = f(w_D).

11. G⟶ D: $H(H(k||W))$; D generates W = f(w_D) and verifies the hash, aborts if mismatch. This convinces D that the sender (G) knows W.

12. D⟶ G: $H(k||W)$; G verifies the hash, if yes, accepts K as session key, else protocol fails. This convinces G that the sender (D) knows W.

13. G ⟶ D: Req($knownData_D$); D populates $knownData_D$ with device model number, manufacturer, $curVersion_D$ and $reqVersion_D$.

14. D ⟶ G: $E_k(knownData_D)$; G recovers $knownData_D$ and stores it alongside $IoTData_D$

Figure 4.2: Authenticated Key Establishment. Steps 1 to 5 include the configuration steps of the IoT and the Gateway device. Steps 6 to 8 show message transfer over TLS. All notation is explained in Table 4.2.

12. D sends the single hash of session key (k) concatenated with W to the Gateway. G validates the hash. If it matches, it accepts k as the session key; otherwise, the protocol fails. This convinces G that D knows W.
    D$\longrightarrow$ G: $H(k||W)$;

13. G sends the request for $knownData_D$ to D.

14. D populates $knownData_D$ with its device model number, manufacturer, *curVersion* and *reqVersion* (NULL by-default) and encrypts it using the session key. D encrypts $knownData_D$ it with k and sends the encrypted $knownData_D$ to G. G decrypts $knownData_D$ and stores in its database with respect to the $IoTData_D$.
    D $\longrightarrow$ G: $E_k(knownData_D)$;

15. When the Gateway wants to communicate with the IoT device or vice versa, it generates a new session key (i.e., k, a symmetric key) for a new session and repeats step 12 and 15.

In case of a manual reset of any D, D and G must both repeat steps 9 through 15 for each D. In the case of G being manually reset, all configuration and data stored after initialization is deleted, and steps 1 through 15 needs to be repeated for all IoT devices. We are using this chain of custody of evidence (i.e., w_D) for the establishment of trust between the IoT device and the Gateway device and key establishment by securely transferring it from the SmartApp to the Gateway through the SUP over the Internet.

4.3.2 Software Update

There are two options for providing software update: the push model and the pull model discussed in Section 2.2 on page 9. The pull model requires less

infrastructure than the push model in terms of complex databases. The pull model is end-device centric, and the push model is SUP centric. There are disadvantages for both models when updates occur automatically, without user involvement. In the case of the pull model, the end-device periodically checks to see if an update is available, and when it is, the SUP sends it down to the device; it consumes resources in polling. In the case of the push model, the SUP might require the maintenance of large and complex databases for maintaining the information of each device. In our design, we use the pull model for the software update.

IoT devices commonly require the use of a smartphone for the initial configuration of a device that does not have any non-wireless user input or output interface for the interaction. After successful onboarding of the IoT devices and the Gateway devices, they can start their intended functionality. The integrity of the software update image is maintained by the digital signature using ECDSA with Ed25519 [57] with key-size 256 bits. Software updates for IoT devices might occur monthly or annually. As shown in Table 6.1 on page 83, the time taken for signature verification with curve Ed25519 is approximately 10 seconds (refer to Table 6.1 on page 83). Even though this is a long time taken by the public-key algorithm on Class IV devices, this is feasible once a month for a software update. In our implementation, the Gateway device queries the SUP for new software updates every 24 hours, but this is configurable.

We use the concept of key-locking [97], where each software image includes a next-update verification key for verifying the integrity and authenticity of the future software images. Associated with each software image version is a public-private key pair for signature and verification. Each image verification public key is available to the IoT device by having been em-

bedded in the previous image. Therefore, the lifespan of each signature-verification key pair is equal to the availability of a new update image. A manufacturer provisions the first image with the verification key in the ROM of D during the manufacturing process. All IoT devices (D) with the same model number can (and should) use the same software image version. Any SUP who wants to provide an update for IoT devices must obtain, from the manufacturer of the devices, the corresponding signature key for providing first update. It is the responsibility of the manufacturer to approve the SUP. When the SUP receives approval, the manufacturer authorizes it by sharing the signature key of the particular IoT device base software image. The manufacturer also publishes the list of trusted SUPs for the user, so that for future software update of IoT devices, the user can configure the gateway device with details of any one of the SUPs.

Figure 4.3 shows a simplified flow diagram of the software update after authenticated key management. According to assumption A7 on page 42, the SUP always has at least one version of the software image available and is responsible for acquiring and checking the authenticity of the software update from the third-party. Here, third party refers to an arbitrary (legitimate) software source that provides an image to the SUP. The SUP tests the software update image before delivering it to the device. Each new software image contains the verification key for the upcoming software image. In our model, the IoT device maintains only two verification keys in its EEPROM: one for the current working software image ($verCurrKey_D$) and another for the upcoming software image ($verNextKey_D$). With every successful software update, the device updates the verification keys.

After onboarding, the Gateway device contains the $knownData_D$ of the IoT device. When the system starts, G sends a request for manifest data

($M_{T(D)}$) to the SUP with $knownData_D$. The SUP first checks whether the request is for an upgrade or not, by checking the *reqVersion* field in $knownData_D$. If the *reqVersion* field is NULL, then it is an upgrade request. The SUP checks its database for an available latest software version that is higher than the *curVersion* sent by G for D, based on the device and manufacturer information present in $knownData_D$. If the SUP finds a latest software version greater than *curVersion*, it appends a sequence number with $M_{T(D)}$. It then checks if the *curVersion* is the base image version (first update of D) for that IoT device model. If yes, then it uses the manufacturer's shared signature private key; otherwise, it uses the signature private key of the last generated key pair for the particular IoT device model. The SUP digitally signs the manifest data ($M_{T(D)}$) of the found software image. The SUP sends M1 = ($S_{s_P}(M_{T(D)})$, $M_{T(D)}$) to G over TLS. G establishes a secure session key (k) with D. G forwards the message M1 to D after encrypting it with the session key as E_k(M1). D *verifies* the signature within M1. After successful verification, D *validates* the device model number, manufacturer information, software version and sequence number. *Sequence Number* is used in the manifest data and software image to avoid a replay attack using an old software version. D communicates success to G on successful validation. G sends a request for the software image ($I_{T(D)}$) based on $M_{T(D)}$ to the SUP. The SUP then finds $I_{T(D)}$ in its database. $I_{T(D)}$ contains a new public verification key embedded in it for the next software update image, and also a sequence number which is filled before signature. The SUP signs the $I_{T(D)}$, with the same signature private key which is used to sign $M_{T(D)}$. The SUP sends M2 = (($S_{s_P}(I_{T(D)})$), $I_{T(D)}$) to G. G forwards the digitally signed image to D after encrypting it with session key as E_k(M2). The IoT device decrypts ($D_k(E_k$(M2))) and verifies the digital signature and validates the information about the device (e.g., checks D's model number, manufacturer, sequence number, and software version

should be greater than *curVersion*). After successful verification and validation, D starts the installation of the new image (update).

After successful installation, D saves the verification key of the running software image in $verCurrKey_D$, and a new verification key of the upcoming image from the SUP in $verNextKey_D$. It stores both keys in the EEPROM of D. The device then sends an acknowledgment of the successful software update with new $knownData_D$ by modifying the *curVersion* field to G. G stores the $knownData_D$.

D only keeps the last two verification public keys for signature verification. Therefore, in case of a software installation failure in the middle during the update process, our device cannot work either with a current image or new image as a few files have modified, and new installation is aborted in the middle. D can only store one software image at a time (refer to Section 4.2.1 on page 44). In the case of an update installation failure, the IoT device sends a failure notice to G along with $knownData_D$ after modifying the *reqVersion* with *curVersion*. G then sends the update query to the SUP, which is then treated as an update installation failure of the software as *reqVersion* is not NULL.

As an IoT device has an expected lifespan of 5-15 years, one public-private pair for signature/verification is not secure. With the computational capabilities at the disposal of today's attacker, the keys which are stronger today, might be weak after few years. This makes the key-locking mechanism a good fit for IoT devices. As a drawback, on manual reset, the IoT device returns to its base image provisioned during manufacturing. However, on the software update, the IoT device is provided with the latest software version image by the SUP by signing it with the manufacturer's shared signature private key of the base image of D.

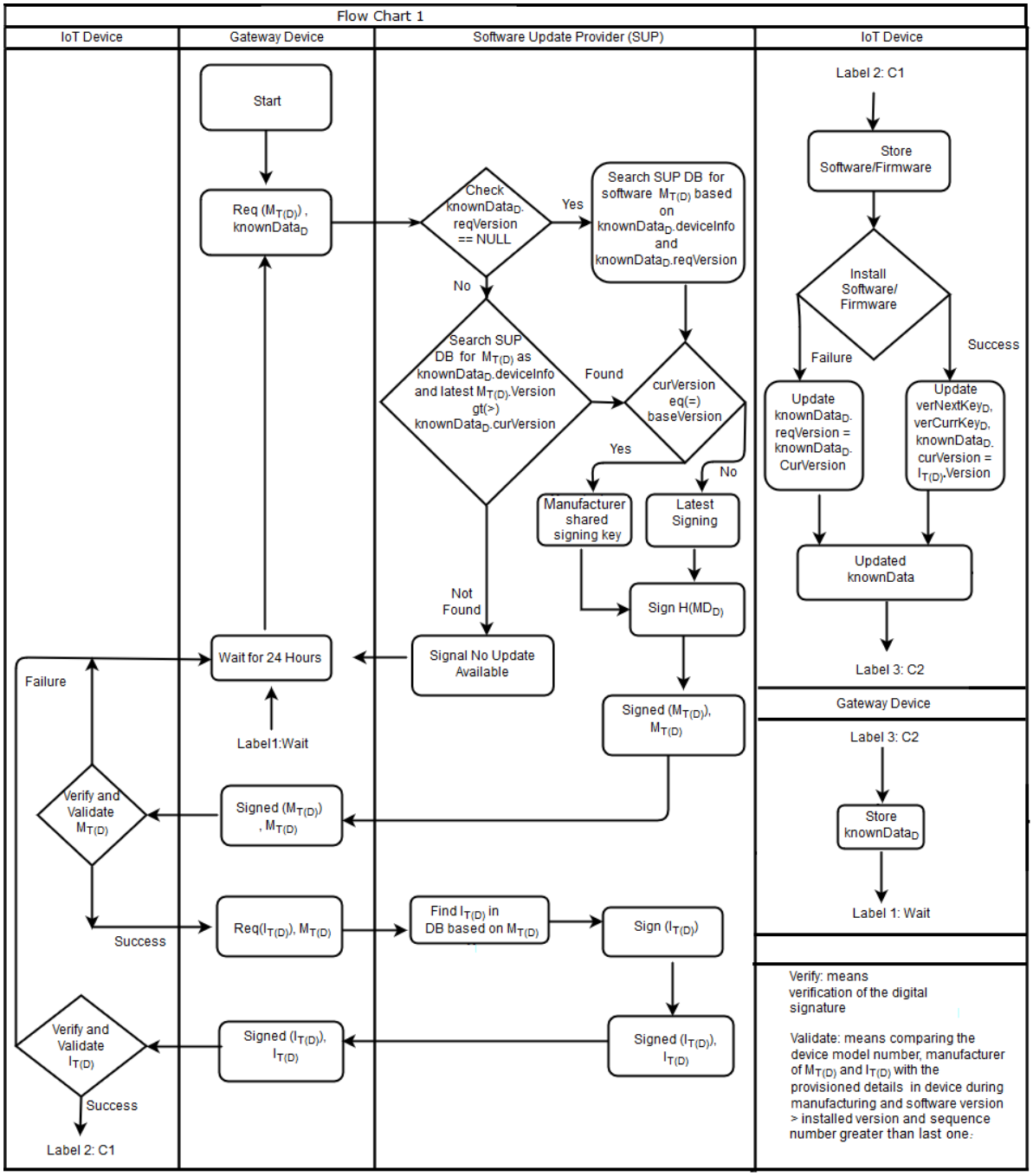


Figure 4.3: Software Update Logic Flow Chart

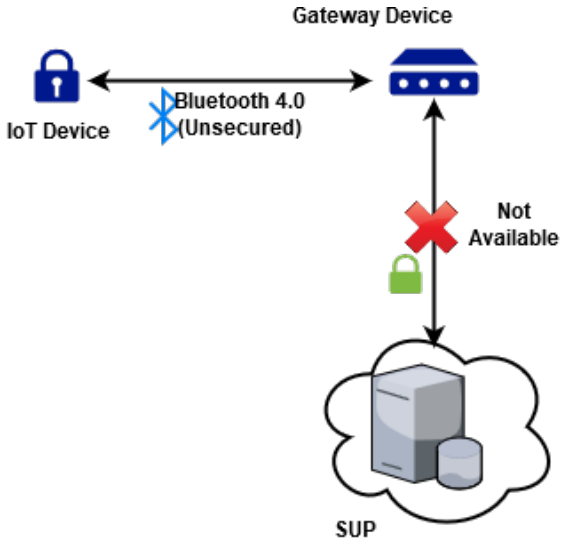


Figure 4.4: Software Update Provider Change Scenario

4.3.3 Software Update Provider Change

In figure 4.4, the SUP is not available due to a merger or bankruptcy or some other reason. The IoT device needs a new SUP who provides software updates regularly. The manufacturer shares the list of the SUPs to the user through its webpage or registered email address. After re-configuring the Gateway device with the new SUP's details, the user needs to repeat onboarding process explained in section 4.3.1 on page 49. The ability to accommodate changing the SUP is helpful in scenarios where the manufacturer or current SUP goes out of business, or the user is not satisfied with the current SUP. Change in SUP requires the user to reset D and G, which clears their EEPROM that contains the relevant information and sends them back to the base image. The user has to register all of the IoT devices again with the new SUP and manually configure the Gateway device using the laptop or desktop with the new SUP information. The IoT device and Gateway need to repeat the process of onboarding discussed in Section 4.3.1 on page 49. After successful onboarding, G requests a software update from the new SUP. The SUP provides the D a new software update through G, and this allows the

user to change the software update provider.

Chapter 5

Design and Implementation Details

In this chapter, we provide the design details of onboarding and a secure software update.

5.1 Implementation

In this section, we will briefly discuss the choices that we have implemented based on components and design requirements. Here, booklet refers to a physical booklet that comes with the device.

Hardware and Software Components: In the section, we discussed the capabilities of all components. Here, we discuss the implementation of these components mentioned in Section 1.3 (refer to page 5).

- IoT Device: It is implemented on an Arduino Mega2560 board, which has an 8-bit microcontroller (ATmega2560) with 16 MHz clock frequency,

commonly programmed using C++ libraries. The Bluetooth module (i.e., HC-05) is in listening mode after initialization.

- Gateway Device: A Raspberry Pi 3B+ module is programmed in Nodejs [71]. It is a 32-bit board with inbuilt Bluetooth and WiFi module. The Gateway is classified as a Class I device per Table 2.1 on page 9.

- SUP: It is a software module and programmed in our prototype in Node.js and uses the MongoDB [66] database. In our prototype, the SUP runs on a laptop with the following specifications: i7-8th Gen processor, 16 GB RAM, 512GB hard disk.

- SmartApp: An Android application, implemented in our prototype using the React platform [27] [26] running over the Samsung Galaxy Note 2 with 2GB RAM and 32 GB memory.

5.1.1 Registration

User Registration

User registration is the first step in onboarding. A username and user-to-SUP-password are used for login into the SUP account, and an email address is used by the SUP to send notifications to the user about changes in his SUP account. The user downloads the SmartApp provided by the SUP from a trusted website (assumption A9 on page 42). Next, the user creates an account with the SUP using the SmartApp. The SmartApp sends a $userProfile$ in a JSON format [18] which includes a username (unique), email address (unique), and user-to-SUP-password over TLS to the SUP. The SUP checks if the received username and email address are previously registered; if not, the SUP generates the salted hash of the user-to-SUP-password using the bcrypt

library [92]. After successful validation, the SUP generates the user's unique identification ($user_{id}$) with regards to the profile and stores the username, email, and salted hashed user-to-SUP-password in a database, and sends a success notification back to the user. The database generates a user's unique identification (i.e., $user_{id}$) for each user profile for internal search and to maintain the sequence of profiles. Figure 5.1(a) explains the signup procedure.

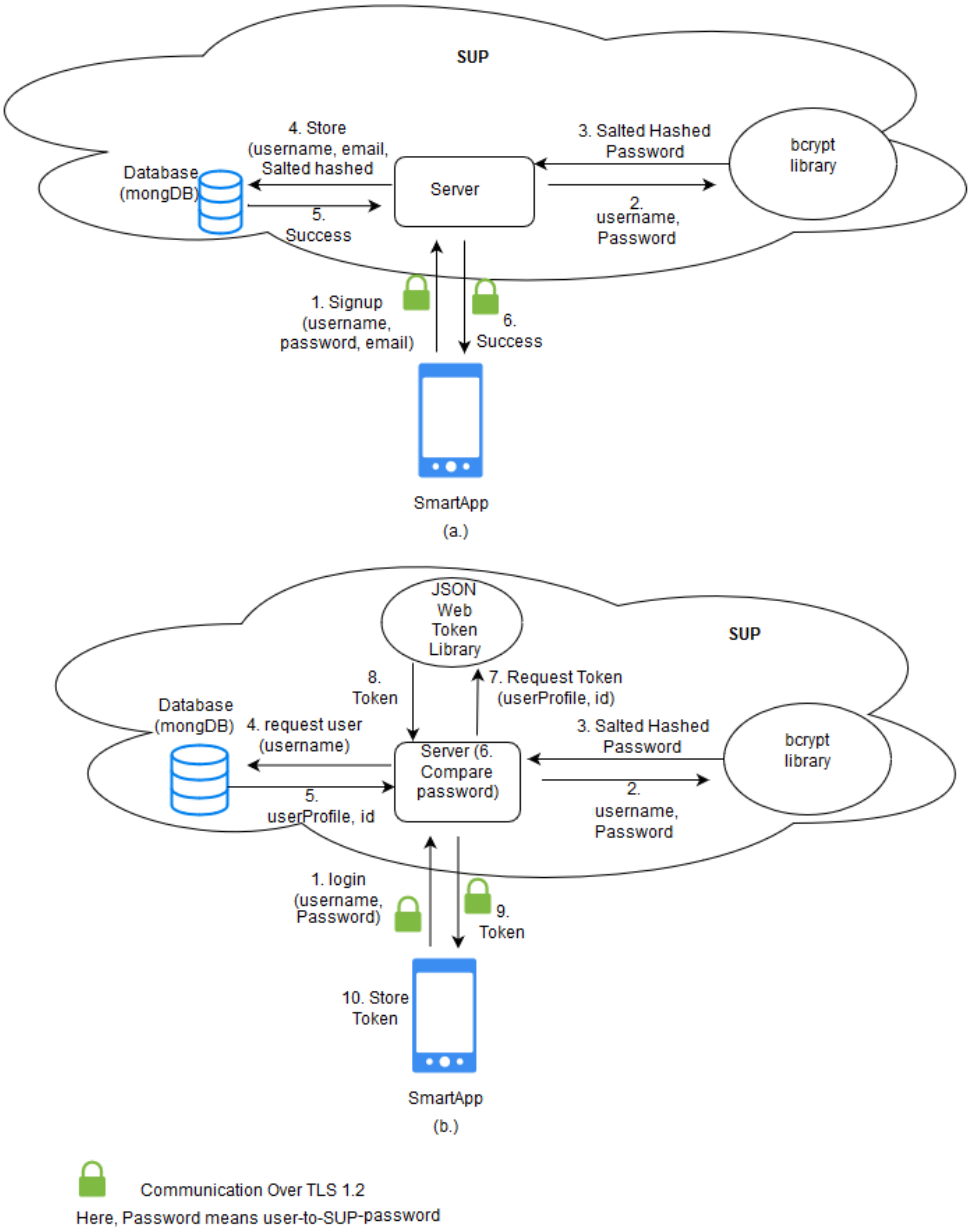
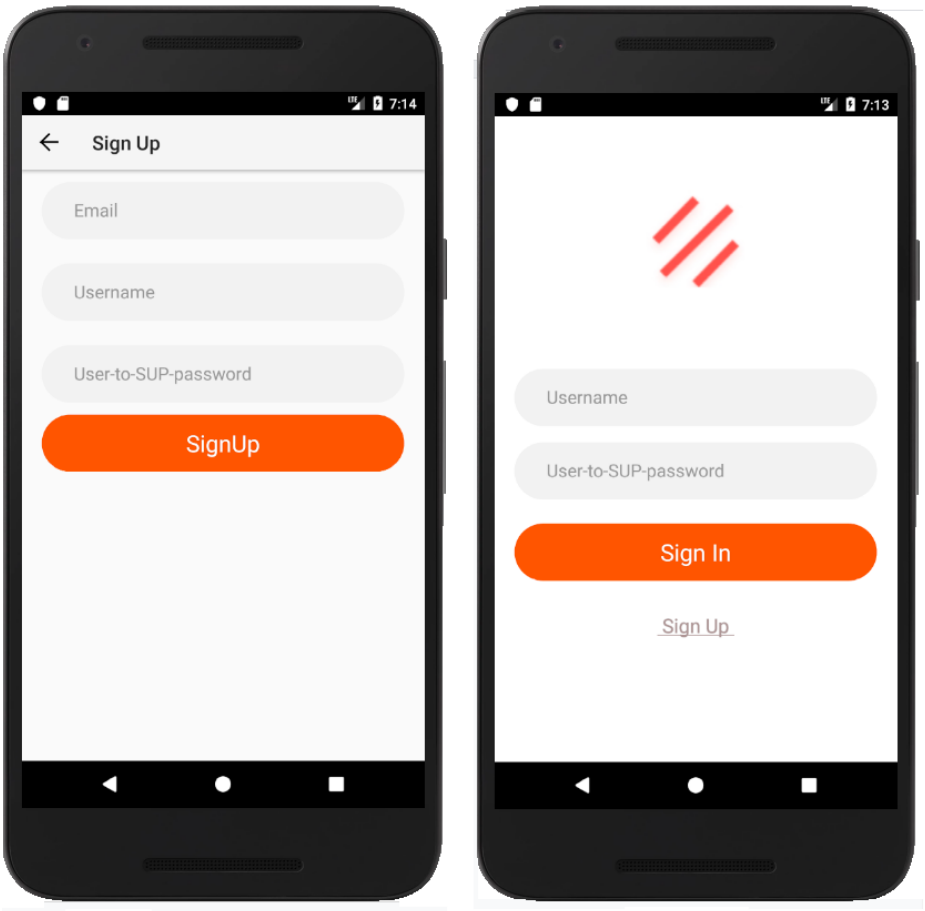


Figure 5.1: User Registration (a) Sign-Up process (b) Login process



a.) SignUp b.) Login

Figure 5.2: User registration using SmartApp

After successful registration, the user logs in to their account over the SmartApp using the username and user-to-SUP-password, and the SmartApp sends the information to the SUP over TLS. The SUP sends a request for user information to the database based on the username. The SUP also generates a salted hash of the user-to-SUP-password using the bcrypt library and compares the hashes. Upon success, the SUP sends all the user information (which includes $user_{id}$, username, user-to-SUP-password, email address) to the JWT (JSON web token) [47] library to generate an authentication token used for future communication until logout. The JWT library returns a token (64-byte random string), and the SUP sends it to the SmartApp which stores the token in local storage. The JWT library also stores the token with regards

to the user's unique identification ($user_{id}$). After successful login authentication, the authentication token is sent to the SUP as part of a header with each request. The SUP sends the token to the JWT library. The JWT library verifies the authentication token and returns the payload (i.e., user's unique identification ($user_{id}$) generated by the database). In case of an authentication token mismatch, the JWT library returns a failure. The user stores the token in local storage, and uses this authentication token for future communication with the SUP. When the user logs out of the account, the SmartApp destroys the authentication token from the local storage. Figure 5.1(a) explains the signup procedure.

Gateway Device Configuration

During the first initialization or after a reset, the Gateway device generates an RSA public-private key pair (used for sending the IoT device secret data from the SmartApp to the Gateway device) and an EC public-private key pair (used for key establishment between D and G). Figure 5.3 shows the Gateway device registration using SmartApp.

After the user registration and successful login to the SUP, the user performs configuration of the Gateway device. The user configures the Gateway device to communicate with the SUP by connecting the physical port of the Gateway device to a laptop. The user enters his login credentials (username and user-to-SUP-password) for the SUP and the URL of the SUP (available at IoT device manufacturer's website; refer to assumption A9 on page 42) in the Gateway device configuration file. The RSA public key (e_G) is visible to the user on the laptop screen during configuration in QR code format, and the user then scans the QR-Code using the SmartApp and stores it in the

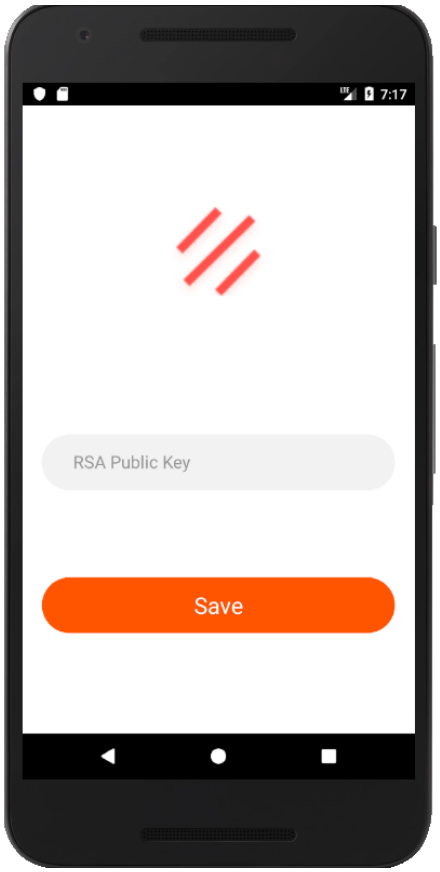


Figure 5.3: Gateway Device Configuration using SmartApp

smartphone's local storage. This completes the configuration of the Gateway device. In case of the SUP changes, the user needs to reconfigure the Gateway device with the new SUP's information.

IoT Device Registration:

During the first-time initialization or after a reset, the IoT device generates an EC public-private key pair (timing result is in Table 6.1 on page 83). We have designed our solution with the intent that one Gateway device can communicate with multiple IoT devices (approximately 10).

The IoT device also has a sealed booklet or sticker which contains the QR-codes of the serial number (N_D), an 8-byte unique password (w_D) with

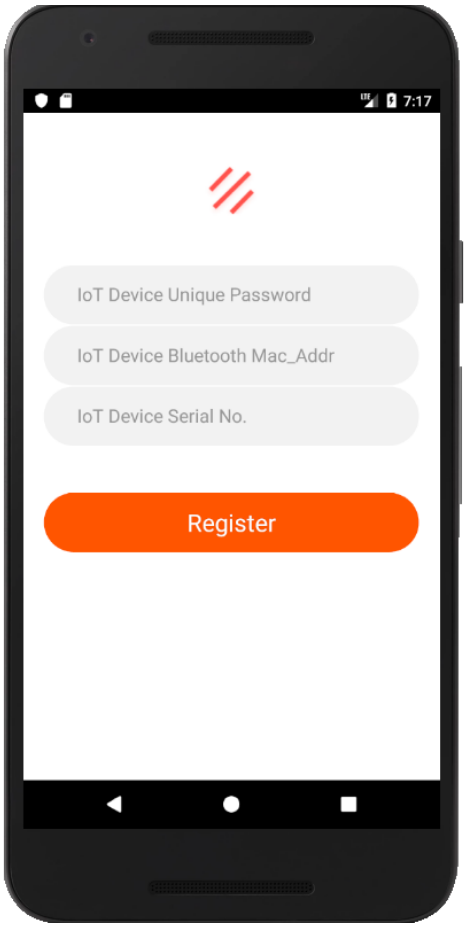


Figure 5.4: IoT Device Registration using SmartApp

regards to the manufacturer, and a Bluetooth MAC-address (B_D). B_D and w_D are used for establishing a secure communication channel between the IoT device and the Gateway device. The user determines the integrity of the booklet or sticker by checking for signs of tampering on the sealed booklet during the first time initialization.

The user opens the booklet and scans the serial number (N_D), Bluetooth MAC-address (B_D), and the password (w_D) from the booklet during the process of the IoT device registration. The SmartApp encrypts the (B_D) and (w_D) using the RSA public-key (e_G) of the Gateway device and sends this encrypted data along with the serial number (N_D) and authentication token (as part of a message header which is stored in local storage of the smart-

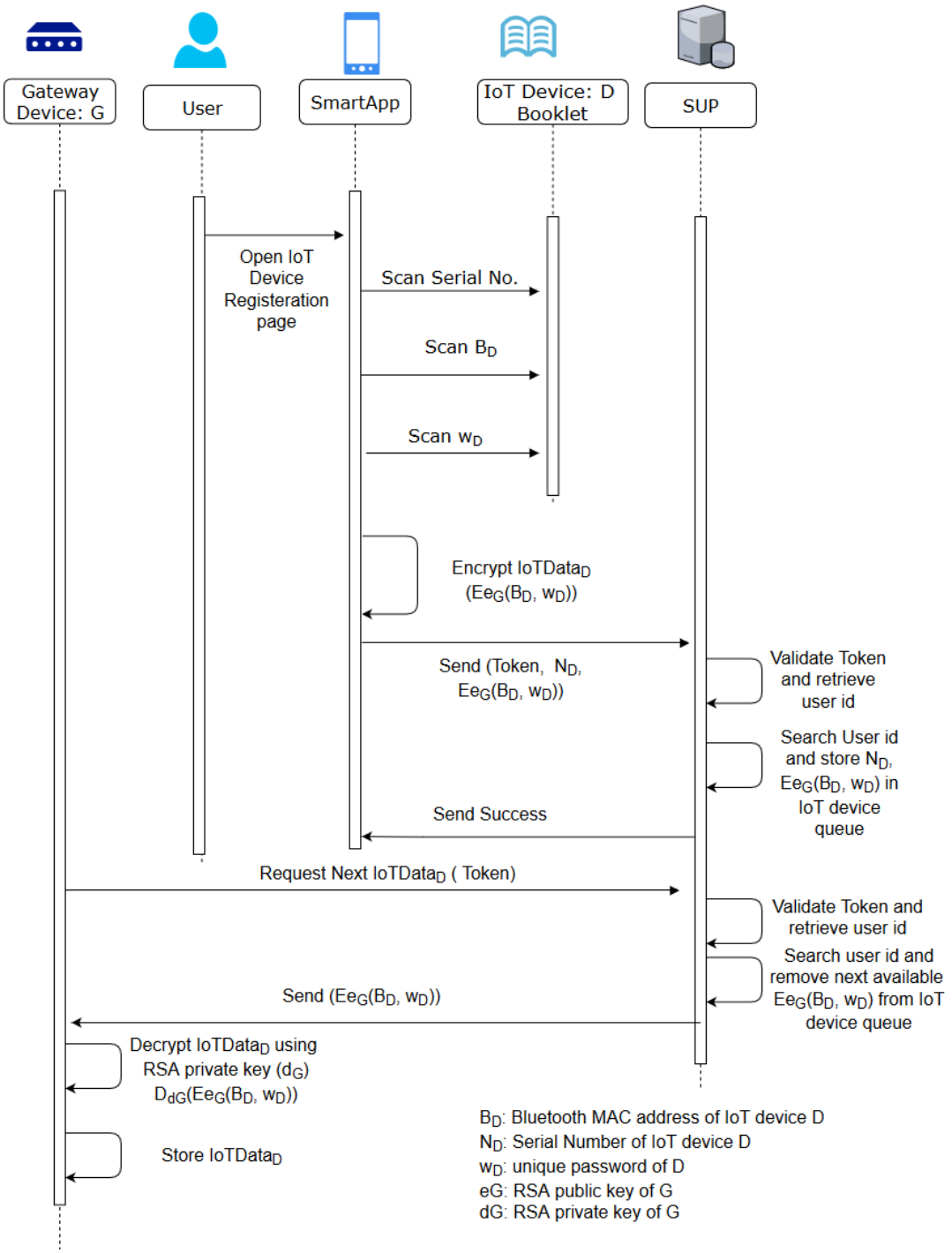


Figure 5.5: IoT Device Registration Process

phone after successful login of the user to the SUP via SmartApp) to the SUP. The SUP validates the authentication token using the JWT library. Upon successful authentication token validation, JWT returns the $user_{id}$. Then, the

SUP stores the encrypted $IoTData_D$ and N_D in the user's IoT device queue in SUP database with regards to the $user_{id}$. Figure 5.5 shows the sequence diagram of the IoT device registration.

Upon successful registration of the IoT device, the Gateway device requests the SUP to send the next $IoTData_D$. The SUP validates the authentication token using the JWT library. Upon successful authentication token validation, JWT returns the $user_{id}$. The SUP checks the IoT device queue with regards to the $user_{id}$, and removes the next available encrypted $IoTData_D$ and sends it to the Gateway device. The Gateway device decrypts the $IoTData_D$ using its RSA private key (d_G) and stores the $IoTData_D$ in its local storage. In the case of IoT device reset, the user does not need to perform the IoT device registration process. However, in the case of Gateway device reset, we need to repeat the complete process of the IoT device registration.

Note: We are using the serial number (N_D) of the IoT device for registration. N_D is a redundant parameter in the current design. However, N_D is usable if we want to extend our design for storing the data collected by the IoT device in the SUP database for each IoT device.

5.1.2 Data Encryption within System

Data encryption is an important part of the design. Communication between the SmartApp, SUP, and the Gateway device utilizes HTTPS, which is HTTP over TLS 1.2, (which provides authenticated encryption). We use TLS 1.2 instead of TLS 1.3 because of the availability of the software tool. However, communication between the IoT device and the Gateway device is over Bluetooth 4.0, which is not secure as it uses a plaintext protocol. Hence, to make the communication channel between the IoT device and the Gateway device

secure, we aim to securely transfer the $IoTData_D$ to the Gateway device as part of the IoT device registration and generate session key (k) using authenticated key establishment.

We use three algorithms between the IoT device and the Gateway device: Elliptic Curve Diffie-Hellman (ECDH-256 bit key, for session key establishment), AES-256 (for session key), and SHA-256 (for key confirmation)). ECDH with curve25519 provides 128-bit equivalent security with recommended prime $2^{255} - 19$ for performance on a wide range of applications. Using the curve25519 [10] library, 32-byte (256 bit) public-private key pairs are generated for two devices. Both devices use their private key and the other's public-key to generate a 32-byte shared secret using the ECDH key establishment. This shared secret is used to authenticate and encrypt data between the two devices.

We use the Advanced Encryption Standard (AES) block cipher as the symmetric key algorithm for the encryption and decryption of block size of 128 bits. For AES, NIST approved three different key lengths: 128, 192 and 256 bits. AES can be performed on 8-bit microprocessors [95]. AES-256 performs 14 rounds to convert plaintext to ciphertext. Each round consists of several operations, one of which depends on the encryption key itself. All of these rounds are used in reverse to convert ciphertext to plaintext using the same encryption key.

SHA-256 [39] is a part of the SHA-2 family of cryptographic hash functions. It gives a hash output of size 256 bits and is defined in the NIST (National Institute of Standards and Technology) standard.

During the first time initialization or after a reset, the IoT device generates an EC public-private key pair, and the Gateway device generates an RSA

public-private key pair and an EC public-private key pair. These two devices use ECDH for generating a shared secret (K), and for encrypting a randomly generated symmetric session key. The IoT device and the Gateway device share their public key with each other in plaintext. An attacker can tamper with these public keys. Key confirmation uses the IoT device unique password (w_D) and the session key (k) to test that the key is known by the IoT device and the Gateway device. Key confirmation is important from operability perspective. The Gateway device sends its EC public key ($e1_G$) to the IoT device using the Bluetooth MAC-address (B_D). The IoT device, after receiving the Gateway device public key, generates a AES-256 shared secret (K_D) using the EC public-key ($e1_G$) of the Gateway device and the EC private-key ($d1_D$) of itself. The IoT device then generates a random symmetric session key (k, AES-256) and symmetrically encrypts k using K_D. The IoT device sends its EC public-key ($e1_D$) and encrypted session key to the Gateway device ($e1_D$, E_{K_D}(k)). The Gateway device receives the IoT device EC public key ($e1_D$) and generates shared secret (K_G, AES-256) using the EC public-key ($e1_D$) of the IoT device and the EC private-key ($d1_G$) of itself such that $K_G = K_D = K$. The Gateway device decrypts the session key, k ($D_{K_G}(E_{K_D}(\text{k}))$).

We use the key confirmation method used in the SPEKE protocol [45]. The Gateway device derives the symmetric key (W, AES-256) from the IoT device unique password, w_D, such as W = f(w_D). The Gateway device sends the double hash of the session key k concatenated with W (i.e., H(H(k||W)) to the IoT device. The IoT device also generates W from the stored w_D in its ROM and then validates the double hash received from the Gateway device. After successful validation, the IoT device has evidence that the Gateway device knows W. After this, the IoT device generates the single hash of the W and k (i.e., H(k||W)) and sends it to the Gateway device. The Gateway device validates the hash. After successful validation, the Gateway device

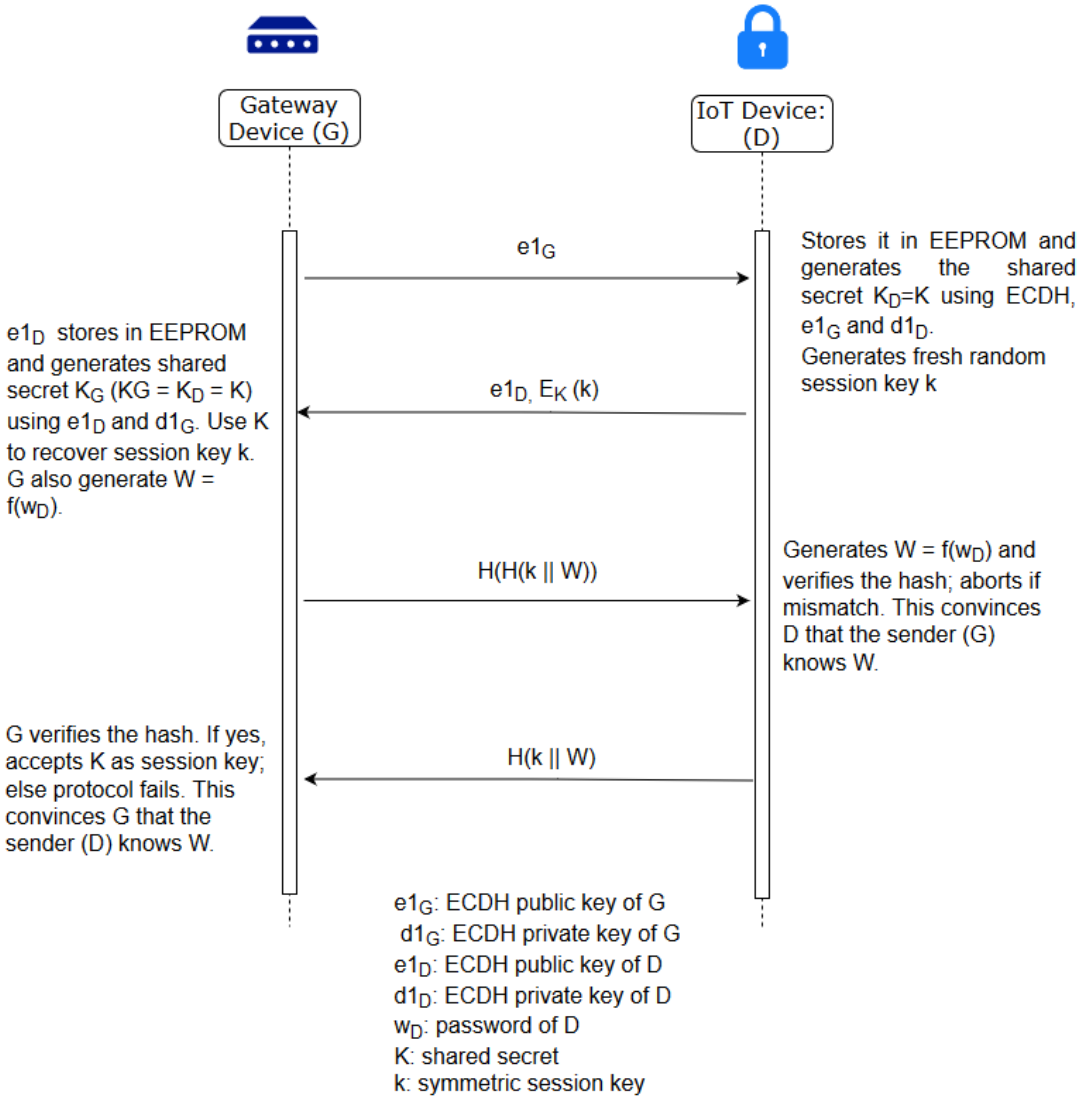


Figure 5.6: Key Establishment for Encryption/Decryption

has evidence that the IoT device knows W. After successful key establishment and confirmation that both devices have the session key (k), to be used for encryption and decryption of data throughout the session. After the session terminates, when the Gateway device next wants to communicate with the IoT device, it generates a new random session key (k) and sends it to the IoT device after encrypting it with a long term shared secret (K), which is followed by the key confirmation process. Steps 9 to 12 in Section 4.3.1 on page 52 shows the key establishment.

5.1.3 Digital Signature System

The digital signature is an important part of the software update. It allows one to verify the integrity and data origin authentication of the software image. We use the elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA) [49], i.e., Ed25519 using SHA-512 and Curve25519 for the signing and verification of the manifest data and software image.

EdDSA provides attack resistance equal to 128 bits of symmetric ciphers. It is a variant of the Schnorr signature system with Edwards curves. It uses public key of size 32-bytes (i.e., 256 bits) and signatures of 64 bytes for Ed25519. We use the Arduino Cryptography Library (updated on November 2018 with version 0.2.0) for the previously mentioned algorithms. Now the steps for the generation of a digital signature in the case of a software update will be discussed. For notation, please refer Table 4.2 on page 48.

1. The manufacturer of the IoT device provisions the first verification public key (v_P), that will be used to verify the signature on the next image in the ROM of the device along with the base image during the time of manufacturing.

2. The SUP requests authorization for providing a software update for the IoT device by asking for the signature private key (s_P) from the manufacturer.

3. The manufacturer vets the SUP, using a process that is beyond the scope of this thesis, which is mentioned as part of assumption A11 on page 42.

4. Upon successful vetting, the manufacturer sends the signature private key (s_P) of the base image of the IoT device to the SUP. Each SUP has

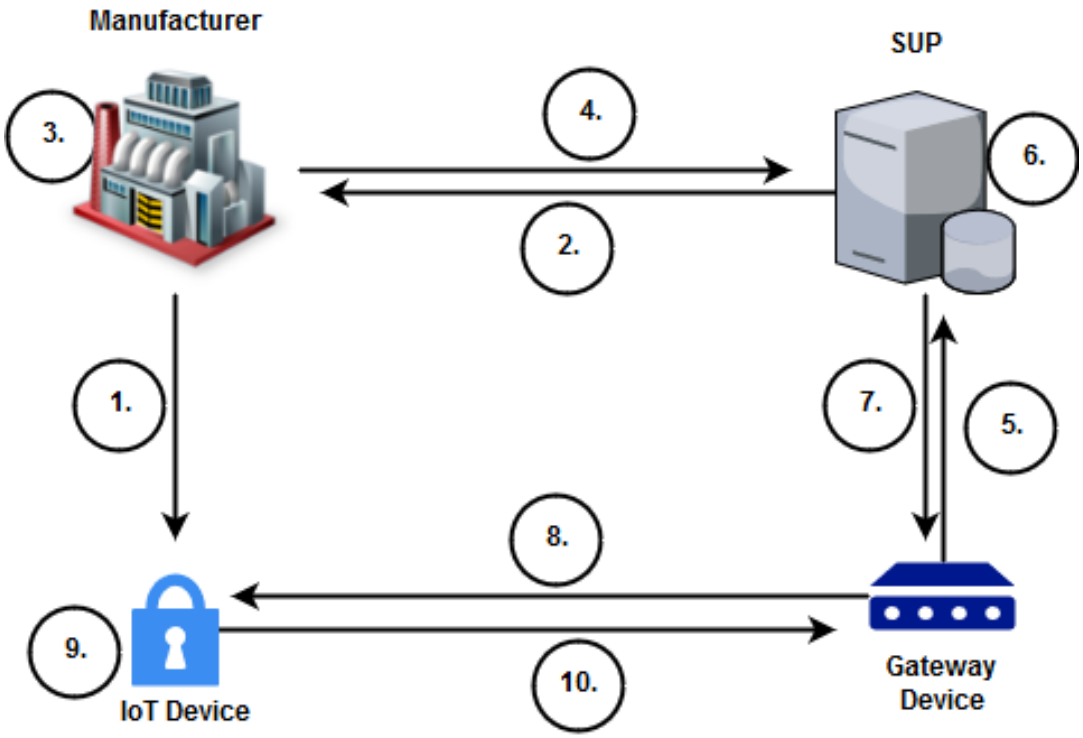


Figure 5.7: SUP Registration Process

a software version higher than the base image version of the IoT device according to assumption A7 on page 42. The SUP stores the s_P. We acknowledge that this is a risky design choice, i.e., the security of the entire system relies on every one of the vetted SUP's being able to protect this private key from compromise successfully. Each IoT device requests a software update within 24 hours and at different time which is hard for an attacker to know.

5. The Gateway device requests a software update (for the IoT device) from the SUP by sending the *curVersion* of the software image of the relevant IoT device.

6. The SUP finds the latest version of the image available and checks if its version is higher than the *curVersion* (i.e., base image version) provided by the Gateway device according to assumption A7 on page 42.

If yes, then the SUP generates a new EC public-private key pair for signature-verification and embeds the verification public key for the next software image in the current software image with the sequence number of the software update. After this, the image is signed with the signature private key (s_P) received from the manufacturer.

7. The SUP makes the signed image available to the Gateway device. After the first update, the SUP will generate a new EC key pair for each subsequent software update.
   Note: This naturally allows for the evolution of stronger signing keys over the lifetime of the IoT device.

8. The Gateway device forwards the signed image to the IoT device.

9. The IoT device verifies the signature of the image with the verification public key (v_P) programmed during manufacturing. Upon successful verification, the IoT device extracts the new verification key from the image and stores it in *verNextKey* variable in EEPROM and updates the *curVersion* field in *knownData*.

10. The IoT device sends the *knownData* to the Gateway device to be stored in the Gateway device's local storage with regards to the IoT device info.

With these steps, the SUP provides the first software update for the IoT device. The method for vetting the SUP by the manufacturer and how the SUP obtains software updates from third parties is out of scope for this research, which is mentioned as part of assumptions A12 on page 42.

Chapter 6

Prototype Evaluation and Security Analysis

This chapter provides an informal security analysis and discusses the evaluation. It also provides the limitations and the conclusion of the prototype.

6.1 Security Analysis

In this section, we have very briefly and informally sketched how our design addresses the threats outlined in Section 4.1 on page 36.

SA1: [Threats T1, T2] The IoT device only receives update information from the registered Gateway device. $M_{T(D)}$ and $I_{T(D)}$ contain the information about the IoT device software version and manifest sequence number (which is filled in during the signature of $M_{T(D)}$), and $I_{T(D)}$ for the software update. $M_{T(D)}$ and $I_{T(D)}$ are digitally signed by the SUP with s_P. Each software version has its own corresponding public-private

key pair. T1 and T2 are mitigated by the verification of the signature and validation of the software version and sequence number present in $M_{T(D)}$ and $I_{T(D)}$.

SA2: [Threat T3] $M_{T(D)}$ and $I_{T(D)}$ contains the IoT device model number and manufacturer information. The IoT device validates the model number, and the manufacturer information in the received manifest data and software image to mitigate threat T3.

SA3: [Threat T4] Verification of the digital signature on $M_{T(D)}$ and $I_{T(D)}$ during a software update mitigates this threat. The manufacturer authorizes the SUPs for providing software updates by sharing the signature private key of the base image of the specific IoT device model.

SA4: [Threat T5] The extended ECDH key exchange protocol (i.e., including a confirmation step) addresses the man in the middle. Knowledge of W is tested as part of onboarding (section 4.3 on page 47) in steps 11-12. The communication between SmartApp, SUP, and the Gateway device is over TLS 1.2, which provides encryption, integrity, and data origin authentication.

SA5: [Threat T6] T6 is mitigated by the key-locking mechanism. The RSA key pair is used in encryption/decryption of the $IoTData_D$ for sending it from SmartApp to the Gateway device. The EC key pairs in the Gateway device and the IoT device used for key establishment are generated every time during first initialization or reset. Therefore, the vulnerability window is minimized.

SA6: [Threat T7] Our model allows the user to change their SUP, e.g., if the manufacturer or current SUP goes out of business, or if the user is not satisfied with the current SUP. However, this adds a new attack surface

of a former SUP with all IoT and Gateway device details. T7 is reduced somewhat by keeping the IoT device information encrypted at the SUP to avoid exploitation of the IoT device's secret (w_D) used in the key confirmation between the IoT device and the Gateway device by a former rogue SUP.

SA7: [Threat T8] T8 is mitigated by generating new RSA and EC keys for the Gateway and IoT device upon any reset. A15, on page 43, rules out a physical device attack.

Gateway Device Analysis

The Gateway device is a Class I device based on Table 2.1, on page 9. The Gateway device can only be configured using a physical connection between the laptop/desktop and the port of the Gateway device. According to assumption A14 on page 42, the Gateway device and the IoT device can only be reset manually. Therefore, no remote attacker can reset the device. We used the pull model for the software update. The end-user configures the Gateway device with the URL of the SUP from the IoT device manufacturer's website (refer to assumption A9 on page 42). Therefore, the Gateway device is the one requesting for software update on behalf of the IoT device. The communication between the Gateway device and the SUP is over TLS.

In case of IoT device reset, D goes to the base image, which is provisioned during the manufacturing (refer to assumption A14 on page 43). There is a small window for a rogue former SUP to send a malicious software update to the Gateway device: the former SUP has both the end-user IoT device information and the signature private key (s_P) of the base image of the IoT device, but, to provide the malicious update, the rogue SUP needs to be aware of IoT device reset time, which is very hard.

Note: Security analysis of the SmartApp and the SUP is out of scope of this thesis (refer to assumption A13 on page 42).

6.2 Evaluation

Onboarding and Software Update

We confirmed the onboarding and software update functionality of our design using the implementation of a prototype. We were able to share the IoT device password to the Gateway device through the network. This secret is used for key confirmation, after key establishment. Our prototype used ECDH (Curve25519) for key establishment, as noted in Section 5.1.2 on page 70. We built a test application for the IoT device and proceeded through multiple software releases with an incremental version code for the software update. In our prototype, we successfully upgraded the version to the IoT device in sequential order ($V1 \longrightarrow V2 \longrightarrow V3$). During testing, we did not upgrade the IoT device software image itself, but tested the logic of receiving and verifying signed images and manifest data, testing out the key management aspects. To test an update installation failure, we tried to upgrade to V4 and modified it to fail in the middle. We are able to install the previous working version (V3) on the IoT device as the two verification keys were kept in the EEPROM of the device (Section 4.3.2 on page 54). The IoT device automatically performed the update and recovery from update installation failure.

The architectural design able to fulfill all the evaluation criteria is discussed in Section 4.1 on page 36.

1. **Resource Constraints:** We tested our prototype on a Class IV IoT de-

vice, namely, Arduino Mega2560. We also performed the timing evaluations (given below) and noted the time taken by Arduino Mega2560 to perform the public-key operations, which are somewhat high as compared to symmetric operations. We also found that it takes approximately 10 sec for signature verification; however, it is usable in case of the relatively infrequent software update scenario.

2. **Robustness:** We sketched the informal security analysis in Section 6.1 of all the threats discussed in Section 4.1 on page 36.

Timing Evaluation

We have timed our prototype implementation on the Arduino Mega2560 board with an 8-bit micro-controller using the Arduino Cryptography Library (last updated on April 2018 with version 0.2.0). Please refer to Table 6.1 on page 83 for checking the time taken by the 8-bit Arduino Mega2560for different cryptographic operations.

The time used for public key operations is non-trivial but affordable as key generation operations are performed once during the initialization or after reset. Similarly, the digital signature verification of software images is relatively infrequent, as it is expected that the release of software updates for Class IV IoT devices will be once or twice a year, and at most, on a monthly basis. Bernstein [11] calculated the number of cycles used for a signing procedure. For a short message, it takes 87548 cycles, and the verification procedure takes under 134000 cycles per signature using a key-size of 256 bits. Bernstein performed his test on a 64-bit 2.4 GHz Intel Westmere (Xeon E5620) CPU. Although the IoT device is only performing the signature verification, we have also tested the time taken in performing the digital signature. Bernstein explained the time difference in performing signature and verification using curve Ed25519. Bernstein's analysis is for a 64-bit processor, rather

than an 8-bit processor, but is nonetheless informative to our work.
We use ECDH with Curve25519 for key establishment. ECDH [19] takes two scalar multiplications. Bernstein [10] showed that curve25519 is faster than other elliptic curves.
AES counter mode (AES-GCM) is used with a key-size of 256 bits. It is used as a session key for encryption and decryption. There is no time difference in the encryption and decryption process when using AES-GCM [89].

Table 6.1 summarizes the time taken by symmetric and asymmetric algorithms to process a single byte or operation, as illustrated in Table 6.1 on page 83. We performed encryption and decryption using AES-GCM with a key size of 256 bits. The test was done ten times over 64 bytes of data. The average result was taken (over the ten trials), then we calculated the time taken to encrypt/decrypt 1 byte of data. Similarly, we performed digital signing and verification ten times over 64 bytes of data and received the average result. In the case of ECDH 256 bit public-private key generation and key agreement, we have performed these operations ten times and took the average result. In the case of large data sizes, the overall time to do AES-GCM encryption and decryption on complete data will be increased as AES is performed on 16 bytes block at one time. But we would expect per-byte encryption/decryption time to remain the same. ECDH 256-bit key generation and key agreement are going to take the same time, as these are independent of the data size. EdDSA signature and verification show timing of performing operations, but on increasing data size from 100 bytes to 1KB-3KB, there is a subsequent increase in timings in milliseconds (approx. 50 to 150 msec).

Timing is measured using the serial monitor tool of Arduino IDE software calculated over Arduino ATMega2560 microcontroller with 16MHz clock frequency.

Operation	Time Taken
AES-GCM-256 encryption	123.52 μsec per byte
AES-GCM-256 decryption	123.06 μsec per byte
Digital Signing (Ed25519, 256 bit ECC)	6.003532s per signature
Signature verification (Ed25519, 256 bit ECC)	9.778552s per signature verification
ECDH 256 bits public-private key generation (Curve 25519)	3.330908s per pair generation
ECDH (key agreement, Curve 25519)	3.332504s per key agreement

Table 6.1: Timing evaluation by 8-bit ATMega2560 with 16MHz clock

6.3 Limitations

In this section, we discuss the limitations of our design and prototype for onboarding and software update.

L1: Our design is unable to handle software version downgrade. But a SUP can provide an older version of software image legitimately by re-assigning it with a new latest version number and signed with last generated signature private key.

L2: The SUP must maintain all signature private keys for every IoT device's type/model. A SUP may be dealing with many types of IoT devices and many instances of each.

L3: We use the pull model for a software update. The end-device periodically checks to see if an update is available by polling.

L4: We mainly focus on maintaining the integrity and data origin authentication of the SUP. We do not handle the passive man-in-the-middle attack. An attacker is able to read the plaintext image file and later exploit software vulnerabilities in the software image which are remotely exploitable by arbitrary parties without any special privileges.

L5: Key establishment using ECDH is not properly ephemeral, i.e., by our design, the ECDH new key pair is only generated during initialization or at the time of device reset. IoT device and Gateway device uses the long-term shared secret (K) until one of the devices is reset.

L6: A single RSA key-pair of a Gateway device is also not properly ephemeral. A new RSA key pair is generated during reset and initial initialization. One RSA key pair for the whole lifetime of a Gateway device increases the vulnerability that the particular key pair is susceptible to discovery. A static RSA private key may, over time, fall within the computational reach of an attacker.

L7: In the case of the IoT device booklet loss, we are unable to recover the unique password used in the ECDH key confirmation in Section 4.3.1 on page 49.

L8: We did not perform the power consumption analysis for Class IV device with and without our solution.

6.4 Future Work and Conclusion

In general, studies [82] [90] [29] [94] have found large numbers of software vulnerabilities in IoT devices, which illustrates the need for a mechanism for an automatic secure software update. We have demonstrated, with the help of our model and a prototype implementation, the viability of using public-key algorithms for one Class IV IoT device (Arduino ATMega2560 8-bit microcontroller with a 16MHz clock) by providing a practical instantiation, which has been tested for onboarding and software updates. Our design uses the concept of key-locking for software updates of the IoT devices.

So that after 15 years, we are not still using the same signature-verification key-pair (which might mean it has become attackable after 15 years). Our design is particularly a good fit for automatic IoT software update, as it also addresses the issue of a manufacturer going out of business. Our model demonstrates that EC-based public key algorithms with curve25519 (security equivalent to RSA-3072) can be used with Class IV IoT devices. Our onboarding system design allows secure transportation of the unique password (w_D) from the IoT device to the Gateway device through the conventional Internet and maintains the integrity and data origin authenticity of the software updates. IoT devices are typically left unattended, which means they are rarely, if ever, patched and often rely on default credentials.

Due to the unique password (proposed as part of our design) of IoT device and all configuration of IoT is controlled by Gateway device present locally, attacks like Mirai and Brickerbot [70] who exploit default username and password or weak password of IoT device can be prevented. This model can be enhanced to use the push model for software updates, the encryption of the image, and the usage of a serial number for the data storage of an IoT device at the SUP database.

Bibliography

[1] Abomhara, M. and Køien, G. (2015). Cyber Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks. *Journal of Cyber Security*, 4:65–88.

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

[3] Arazi, O. and Qi, H. (2005). Self-certified group key generation for ad hoc clusters in wireless sensor networks. In *14th International Conference on Computer Communications and Networks, ICCCN*.

[4] Banks, A. and Gupta, R. (2014). MQTT Version 3.1.1. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[5] Barrera, D., Clark, J., McCarney, D., and van Oorschot, P. C. (2012). Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 81–92.

[6] Barrera, D., McCarney, D., Clark, J., and van Oorschot, P. C. (2014). Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In *ACM WiSec*.

[7] Barrera, D., Molloy, I., and Huang, H. (2017). IDIoT: Securing the Internet of Things like it's 1994. *CoRR*, abs/1712.03623. A subset of this paper appeared as 'Standardizing IoT Network Security Policy Enforcement', Workshop on Decentralized IoT Security and Standards (DISS 2018), 6 pages.

[8] Barrera, D. and van Oorschot, P. (2011). Secure Software Installation on Smartphones. *IEEE Security and Privacy*, 9(3):42–48.

[9] Barreto, P. S. L. M. and Naehrig, M. (2006). Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, pages 319–331. Springer Berlin Heidelberg.

[10] Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman Speed Records. In *PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, April 24-26, 2006, Proceedings*, pages 207–228.

[11] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B. (2012). High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89.

[12] Bon, M. (2016). A Basic Introduction to BLE Security. *digikey.com*.

[13] Bormann, C., Ersue, M., and Keränen, A. (2014). Terminology for Constrained-Node Networks. RFC 7228.

[14] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and Raymor, B. (2018). CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. *RFC 8323*.

[15] Boyko, V., MacKenzie, P., and Patel, S. (2000). Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. In *EUROCRYPT*.

[16] Boyko, V., Peinado, M., and Venkatesan, R. (1998). Speeding up discrete log and factoring based schemes via precomputations. In *Advances in Cryptology — EUROCRYPT'98*, pages 221–235. Springer Berlin Heidelberg.

[17] Brad, M. (2017). Over-The-World-Through-MQTT-Aftermath. `https://www.blackhat.com/docs/us-17/thursday/us-17-Lundgren-Taking-Over-The-World-Through-Mqtt-Aftermath.pdf`.

[18] Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159.

[19] Brown, D. R. L. (2009). Standards for Efficient Cryptography. SEC 1: Elliptic Curve Cryptography. `http://secg.org/sec1-v2.pdf`. SECG-SEC-v2.

[20] Choi, B., Lee, S., Na, J., and Lee, J. (2016). Secure firmware validation and update for consumer devices in home networking. *IEEE Transactions on Consumer Electronics*, 62(1):39–44.

[21] Common Weakness Enumeration (2019a). CWE - CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). `https://cwe.mitre.org/data/definitions/120.html`. (Accessed on 08/01/2019).

[22] Common Weakness Enumeration (2019b). CWE - CWE-190: Integer Overflow or Wraparound. `https://cwe.mitre.org/data/definitions/190.html`. (Accessed on 08/01/2019).

[23] Common Weakness Enumeration (2019c). CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html. (Accessed on 08/01/2019).

[24] Cui, A., Costello, M., and Stolfo, S. (2013). When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *NDSS*.

[25] Echevarria, R. (2017). Intel secure device onboard: Onboarding billions of devices just got simpler. https://software.intel.com/en-us/blogs/2017/10/03/intel-secure-device-onboard. (Accessed on 06/22/2019).

[26] Facebook (2019). "React Native 0.59 · A framework for building native apps using React". https://facebook.github.io/react-native/. (Accessed on 06/12/2019).

[27] Facebook (2019). "React v16.8.6 – A JavaScript library for building user interfaces". https://reactjs.org/. (Accessed on 06/12/2019).

[28] Fawaz, K., Kim, K.-H., and Shin, K. G. (2016). Protecting Privacy of BLE Device Users. In *USENIX Security Symposium*.

[29] Federal Trade Commission (2015). FTC Report on Internet of Things Urges Companies to Adopt Best Practices to Address Consumer Privacy and Security Risks.
https://www.ftc.gov/system/files/documents/reports/federal-trade-commission-staff-report-november-2013-workshop-entitled-internet-things-privacy/150127iotrpt.pdf .

[30] Finney, H., Donnerhacke, L., Callas, J., Thayer, R. L., and Shaw, D. (2007). OpenPGP Message Format. RFC 4880.

[31] Friel, O., Lear, E., Pritikin, M., and Richardson, M. (2018). BRSKI over IEEE 802.11. Internet-Draft draft-friel-brski-over-802dot11-01, Internet Engineering Task Force.

[32] Gemalto (2017). "Gemalto survey confirms that consumers lack confidence in IoT device security". `https://www.gemalto.com/press/pages/gemalto-survey-confirms-that-consumers-lack-confidence-in-iot-device-security-.aspx`.

[33] Gilburg, J. (2017). Zero Touch Device Onboarding for IoTControl Platforms. *RSA Conference*.

[34] Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., Richer, J. P., Lefkovitz, N. B., Danker, J. M., Choong, Y.-Y., and et al. (2017a). NIST: Digital Identity Guidelines :Authentication and Lifecycle Management. *NIST Special Publication 800-63B*.

[35] Grassi, P. A., Garcia, M. E., and Fenton, J. L. (2017b). NIST: Digital Identity Guidelines. *NIST Special Publication 800-63*.

[36] Gupta, H. and van Oorschot, P. C. (2019). Onboarding and Software Update Architecture for IoT Devices. In *Privacy, Security, and Trust*.

[37] Hahm, O., Baccelli, E., Petersen, H., and Tsiftes, N. (2016). Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(5):720–734.

[38] Hankerson, D., Menezes, A. J., and Vanstone, S. (2003). *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg.

[39] Hansen, T. and E. Eastlake 3rd, D. (2006). US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634.

[40] Hao, F. (2017a). J-PAKE: Password-Authenticated Key Exchange by Juggling. RFC 8236.

[41] Hao, F. (2017b). Schnorr Non-interactive Zero-Knowledge Proof. RFC 8235.

[42] Ho, G., Leung, D., Mishra, P., Hosseini, A., Song, D., and Wagner, D. (2016). Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 461–472. ACM.

[43] Igoe, K., McGrew, D., and Salter, M. (2011). Fundamental Elliptic Curve Cryptography Algorithms. RFC 6090.

[44] Iwata, T., Song, J., Lee, J., and Poovendran, R. (2006). The AES-CMAC Algorithm. RFC 4493.

[45] Jablon, D. P. (1996). Strong password-only authenticated key exchange. *ACM SIGCOMM Comput. Commun. Rev.*, 26(5):5–26.

[46] Jackson (2018). SB-327 Information privacy: connected devices. *California Legislative Information*.

[47] Jones, M., Bradley, J., and Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519.

[48] Jones, M., Wahlstroem, E., Erdtman, S., and Tschofenig, H. (2018). CBOR Web Token (CWT). RFC 8392.

[49] Josefsson, S. and Liusvaara, I. (2017). Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032.

[50] Julien, C., Liu, C., Murphy, A. L., and Picco, G. P. (2017). BLEnd: Practical Continuous Neighbor Discovery for Bluetooth Low Energy. In

ACM/IEEE International Conference on Information Processing in Sensor Networks.

[51] Kainda, R., Flechais, I., and Roscoe, A. W. (2009). Usability and Security of Out-of-band Channels in Secure Device Pairing Protocols. In *5th Symposium on Usable Privacy and Security*. ACM.

[52] Khan, M. (2017-08-28). Enhancing Privacy in IoT Devices through Automated Handling of Ownership Change. `http://urn.fi/URN:NBN:fi:aalto-201709046805`.

[53] Kogan, D., Manohar, N., and Boneh, D. (2017). T/Key: Second-Factor Authentication From Secure Hash Chains. In *ACM CCS*.

[54] Kotzias, P., Matic, S., Rivera, R., and Caballero, J. (2015). Certified PUP: Abuse in Authenticode Code Signing. In *ACM CCS*.

[55] Kumar, A., Saxena, N., Tsudik, G., and Uzun, E. (2009). A comparative study of secure device pairing methods. *Pervasive and Mobile Computing*, 5(6):734–749.

[56] Lamport, L. (1981). Password Authentication with Insecure Communication. *Commun. ACM*, 24(11):770–772.

[57] Langley, A., Hamburg, M., and Turner, S. (2016). Elliptic Curves for Security. RFC 7748.

[58] Lee, B. and Lee, J.-H. (2017). Blockchain-based Secure Firmware Update for Embedded Devices in an Internet of Things Environment. *J. Supercomput.*, 73(3):1152–1167.

[59] Li, L., Abd El-Latif, A., and Niu, X. (2012). Elliptic curve EIGamal based homomorphic image encryption scheme for sharing secret images . *Signal Processing*, 92:1069–1078.

[60] Lundgren, L. (2017). Taking Over the World Through MQTT - Aftermath. https://www.blackhat.com/us-17/briefings.html.

[61] Mahto, D. and Yadav, D. K. (2017). One-time password communication security improvement using elliptic curve cryptography with iris biometric. *International Journal of Applied Engineering Research*, 12:7105–7114.

[62] McCune, J. M., Perrig, A., and Reiter, M. K. (2005). Seeing-is-believing: using camera phones for human-verifiable authentication. In *2005 IEEE Symposium on Security and Privacy*, pages 110–124.

[63] Melnikov, A. and Fette, I. (2011). The WebSocket Protocol. RFC 6455.

[64] Menezes, A. J. (1994). *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers.

[65] Menezes, A. J., Vanstone, S. A., and van Oorschot, P. C. (1996). *Handbook of Applied Cryptography*. CRC Press, Inc.

[66] MongoDB (2017). "The most popular database for modern apps | MongoDB, Version 3.6". https://www.mongodb.com/. (Accessed on 06/19/2019).

[67] Moran, B., Meriac, M., and Tschofenig, H. (2019a). A Firmware Update Architecture for Internet of Things Devices. Internet-Draft draft-moran-suit-architecture-02, IETF.

[68] Moran, B., Tschofenig, H., and Birkholz, H. (2019b). Firmware Updates for Internet of Things Devices - An Information Model for Manifests. Internet-Draft draft-ietf-suit-information-model-02, IETF.

[69] Moriarty, K., Kaliski, B., Jonsson, J., and Rusch, A. (2016). PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017.

[70] Mustapha, H. and Alghamdi, A. M. (2018). DDoS Attacks on the Internet of Things and Their Prevention Methods. In *Proceedings of the 2Nd International Conference on Future Networks and Distributed Systems*, ICFNDS. ACM.

[71] Node.js Foundation (2019). Node.js 12.4.0. `https://nodejs.org/en/`. (Accessed on 06/12/2019).

[72] NPM (2018). react-native-rsa-native - npm, 1.0.24. `https://www.npmjs.com/package/react-native-rsa-native`. (Accessed on 06/12/2019).

[73] Ozmen, M. O. and Yavuz, A. A. (2017). Low-Cost Standard Public Key Cryptography Services for Wireless IoT Systems. In *Workshop on Internet of Things Security and Privacy*. ACM.

[74] Pritikin, M., Richardson, M., Behringer, M. H., Bjarnason, S., and Watsen, K. (2019). Bootstrapping Remote Secure Key Infrastructures (BRSKI). Internet-Draft draft-ietf-anima-bootstrapping-keyinfra-18, Internet Engineering Task Force.

[75] Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446.

[76] Rescorla, E. and Dierks, T. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246.

[77] Ronen, E., Shamir, A., Weingarten, A., and O'Flynn, C. (2018). IoT Goes Nuclear: Creating a Zigbee Chain Reaction. *IEEE Security Privacy*, 16(1):54–62.

[78] Ryan, M. (2013). Bluetooth: With Low Energy Comes Low Security. In *Workshop on Offensive Technologies*. USENIX.

[79] Salman, A., Diehl, W., and Kaps, J. (2017). A light-weight hardware software co-design for pairing-based cryptography with low power and energy consumption. In *International Conference on Field Programmable Technology (ICFPT)*.

[80] Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. (2010). Survivable Key Compromise in Software Update Systems. In *ACM CCS*.

[81] Saxena, N., Ekberg, J. ., Kostiainen, K., and Asokan, N. (2006). Secure device pairing based on a visual channel. In *2006 IEEE Symposium on Security and Privacy*.

[82] Schneier, B. (2014). Essays: The Internet of Things Is Wildly Insecure—And Often Unpatchable. `https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html`.

[83] Shelby, Z., Hartke, K., and Bormann, C. (2014). The Constrained Application Protocol (CoAP). RFC 7252.

[84] Shin, S. and Kobara, K. (2012). Efficient Augmented Password-Only Authentication and Key Exchange for IKEv2. RFC 6628.

[85] Shin, S., Kobara, K., and and (2016). A security framework for MQTT. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 432–436.

[86] Soriente, C., Tsudik, G., and Uzun, E. (2007). BEDA: Button-Enabled Device Pairing. *IACR Cryptology ePrint Archive*, 2007:246.

[87] Soriente, C., Tsudik, G., and Uzun, E. (2008). HAPADEP: Human-Assisted Pure Audio Device Pairing. In *Information Security*, pages 385–400, Berlin, Heidelberg. Springer Berlin Heidelberg.

[88] Stajano, F. and Anderson, R. J. (1999). The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Security Protocols, 7th International Workshop, April*.

[89] STMicroelectronics (2013). UM0586: User manual STM32 Cryptographic Library. https://www.st.com/content/ccc/resource/technical/document/user_manual/34/1a/20/75/7f/84/45/cb/CD00208802.pdf/files/CD00208802.pdf/jcr:content/translations/en.CD00208802.pdf#page=119. (Accessed on 06/23/2019).

[90] Tal, S. and Oppenheim, L. (2014). Too Many Cooks: Exploiting the Internet of TR-069 Things. http://mis.fortunecook.ie/too-many-cooks-exploiting-tr069_tal-oppenheim_31c3.pdf.

[91] Thiranant, N., Lee, Y. S., and Lee, H. (2015). Performance Comparison Between RSA and Elliptic Curve Cryptography-Based QR Code Authentication. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*.

[92] Tidelift (2019). Bcrypt 3.0.6 on npm - libraries.io. https://libraries.io/npm/bcrypt. (Accessed on 06/19/2019).

[93] Tschofenig, H. (2016). Fixing User Authentication for the Internet of Things (IoT). *Datenschutz und Datensicherheit*, 40(4):222–224.

[94] Tschofenig, H. and Farrell, S. (2017). Report from the Internet of Things Software Update (IoTSU) Workshop 2016. RFC 8240.

[95] Tschofenig, H. and Pegourie-Gonnard, M. (2015). Performance of State-of-the-Art Cryptography on ARM-based Microprocessors. https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf.

[96] van Oorschot, P. C. (2019). *Computer Security and the Internet Security: Tools and Jewels*. Springer Nature.

[97] van Oorschot, P. C. and Wurster, G. (2012). Reducing Unauthorized Modification of Digital Objects. *IEEE Trans. Software Eng.*, 38(1):191–204.

[98] View, M., Rydell, J., Pei, M., and Machani, S. (2011). TOTP: Time-Based One-Time Password Algorithm. RFC 6238.

[99] Wash, R., Rader, E., Vaniea, K., and Rizor, M. (2014). Out of the Loop: How Automated Software Updates Cause Unintended Security Consequences. In *10th Symposium On Usable Privacy and Security*.

[100] Watsen, K., Abrahamsson, M., and Farrer, I. (2019). Secure Zero Touch Provisioning (SZTP). RFC 8572.

[101] Wazid, M., Das, A. K., Odelu, V., Kumar, N., Conti, M., and Jo, M. (2018). Design of Secure User Authenticated Key Management Protocol for Generic IoT Networks. *IEEE Internet of Things Journal*, 5(1):269–282.

[102] Weißbach, M., Taing, N., Wutzler, M., Springer, T., Schill, A., and Clarke, S. (2016). Decentralized coordination of dynamic software updates in the Internet of Things. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 171–176.

[103] Whiting, D., Housley, R., and Ferguson, N. (2003). Counter with CBC-MAC (CCM). RFC 3610.

[104] Willingham, T., Henderson, C., Kiel, B., Haque, M. S., and Atkison, T. (2018). Testing Vulnerabilities in Bluetooth Low Energy. In *ACMSE*.

[105] Wurster, G. and van Oorschot, P. C. (2007). Self-Signed Executables: Restricting Replacement of Program Binaries by Malware. In *USENIX HotSec*.

[106] Zetter, K. (2015). How the NSA's Firmware Hacking Works and Why It's So Unsettling. https://www.wired.com/2015/02/nsa-firmware-hacking/.

[107] Zhou, W., Jia, Y., Peng, A., Zhang, Y., and Liu, P. (2019). The Effect of IoT New Features on Security and Privacy: New Threats, Existing Solutions, and Challenges Yet to Be Solved. *IEEE Internet of Things Journal*, pages 1–1.

[108] Zhu, J. (2018). A Secure and Automatic Firmware Update Architecture for IoT Devices. Internet-Draft draft-zhu-suit-automatic-fu-arch-00, IETF.