# Onboarding and Software Update Architecture for IoT Devices

Hemant Gupta and Paul C. van Oorschot
*School of Computer Science*
*Carleton University*
Ottawa, Canada
hemant@ccsl.carleton.ca

***Abstract*—The vast number of in-use Internet of Things (IoT) devices is by consensus, expected to continue rapid growth. These devices are subject to an expanding list of attacks that exploit both software vulnerabilities and design choices. This highlights the importance of architectural design of management for cryptographic keys involved in both initial configuration (onboarding) and secure, automatic update of device software and firmware. Low-level IoT devices with constrained processors and smaller registers and caches are computationally challenged to carry out desktop-type and server-type public-key cryptographic operations, e.g., as needed for key establishment and authentication of software updates. To this end, we design and prototype an architecture for onboarding and secure software update of 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 a software update. We also provide an informal security analysis. The design addresses the scenario of a transfer of update authority, e.g., when a manufacturer ceases to provide ongoing software updates upon going out of business.**

***Index Terms*—Internet of Things (IoT), Security, Onboarding, Software Update**

## I. Introduction

The growing network of connected devices, often called the Internet of Things (IoT), offers a vast array of security vulnerabilities for attackers to exploit. 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 sensitive personal data. This elevates the importance of means to provide secure software and firmware updates. Every time a company deploys a product, arguably, as part of a global ecosystem, they have a responsibility to provide ongoing security updates to it. This is not so easily done in an IoT world of heterogeneous devices from a vast number of vendors.

In 2016, the Mirai Botnet [1] exploited IoT devices. It largely used default access credentials to infect devices and then execute a DDoS (Distributed Denial of Service) attack using them. Similar attacks occured in 2015 [2] on hard disk firmware, where malicious code was inserted by reflashing with the help of a hacking tool. In 2013 [3], researchers demonstrated an attack on printer firmware by utilizing a design flaw in the remote update functionality.

Many IoT devices continue to be protected by only a default password. Recently, a law passed in California, requiring that IoT device [4] manufacturers provide a unique password for each device. While a small step, it is a step in the right direction. A secure software architecture update for low-level IoT devices remains an open problem as is device on-boarding for devices with no user input/output interface. Low-level IoT devices have low processing power and small memory—just sufficient for dedicated tasks. It is challenging to deploy public-key cryptography and to deliver software updates through the Internet for these constrained devices [5].

Secure software update for Class IV IoT devices (see Section II Table I) is challenging as they have no user input/output interface, no pre-shared secret for establishing trust between devices, and due to low computational capabilities, it is hard to implement public-key operations. Nonetheless for secure software update, we argue that onboarding using authenticated key management is the first step.

We define an architecture involving four components: an IoT device, a gateway, a smartphone application, and a software update provider (SUP). Our design for secure software update is derived from the method of Wurster [6] [7] for self-signed binaries for smartphones. Using this, we contribute the following:

- We explore the viability of secure software update using public-key cryptography on 8-bit micro-controllers.
- We use the concept of key-locking [7] for software update on 8-bit IoT devices.
- We explain and prototype a secure method for on-boarding IoT devices using authenticated key management involving a gateway device and smartphone.
- We address the issue of changing the authority of ownership of software update, in the case that a manufacturer goes out of business.

Section 2 provides background. In Section 3, we discuss the attacker goals and threat model related to software update. Section 4 describes the design and implementation of our prototype used for proof of concept of secure software updates for low-level IoT devices. Section 5 presents the evaluation and security analysis. Section 6 summarizes the related work

2019 IEEE. This is the authors copy for personal use, IEEE permission must be obtained for all others. To appear at PST in Fredericton, Canada,August 2019. Contact author:hemant@ccsl.carleton.ca.

in the area of software update and onboarding for IoT devices. Section 7 provides concluding remarks and future work.

## II. Background

Software update is a vital part of IoT security. Software updates fix features that are not performing as intended, add software enhancement, and address security issues [8]. Figure 1 shows our basic architecture involving four components: a Class IV IoT device (see Table I), a Software Update Provider (i.e., SUP which provides software update for devices), a Class I Gateway device (to help IoT devices communicate with SUP), and a smartphone application (i.e., SmartApp, used for device registration with the SUP). In the Internet of Computers (IoC), software update mechanisms for the Windows operating system (e.g., Authenticode [9]), Linux, and Mac have common options: automatic update, download and install; automatic update, download but ask the user permission before installation; and manual update, download and install. End-users commonly opt for the first of these [10]. One problem with these scenarios is if the application vendor stops providing updates. In such a scenario, if the user is not aware of the applications running on their system and how to update them manually, then outdated software is prone to attacks. In the case of manual software downloads, files from unofficial sources may be untrustworthy.

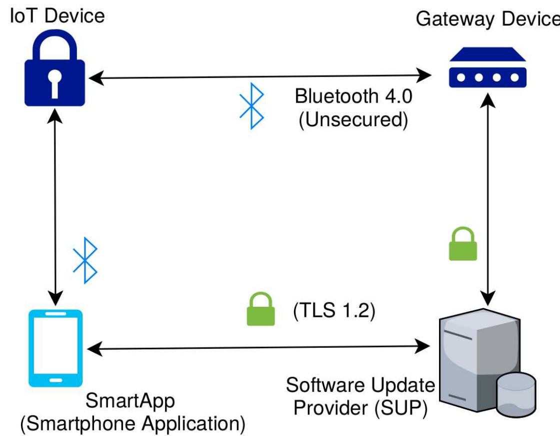



Fig. 1. System Design for Onboarding and Software Update. Smart Lock is used as an example IoT device

Software updates for a smartphone operating system like Android or iPhone are issued over the air (OTA) by the manufacturers and SUPs. OTA is a way to keep end user software updated, including protection from malicious attacks [11] [12]. Zhu [13] notes three modes of OTA updates for IoT devices, similar to those noted by Barrera [14] for smartphone applications. The first is client-initiated (i.e., pull model), in which the client queries the server for update periodically; when a new update is available, the client downloads the manifest data and image and when the device is in IDLE state, the client installs the update. The second is 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. The third is negotiated, where the server notifies the client about the available new software or firmware image and then the client decides whether they want to install the update or not. Researchers from both academia and industry seek a standardized mechanism for software and firmware upgrade for constrained devices.

Borman [5] 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 I based on our integration of research literature and white papers [15] [16] [17] [18]. In Table I, we classify IoT device processors into five classes based on aspects of architecture including RAM size, bus size, clock frequency, power consumption and others.

TABLE I
EXTENDED TAXONOMY OF IOT DEVICE PROCESSORS.
NA refers to not applicable to the class.

| Class/ Features | Class I | Class II | Class III | Class IV | Class V |
|---|---|---|---|---|---|
| Bus Size | 64 bit | 32 bit | 16 bit | 8 bit | 4 and 8 bit |
| 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 - GHz | 80 MHz-180 MHz | 4 MHz-80 MHz | 128 kHz-16 MHz | 32 kHz-8 MHz |
| Power Usage | 10 mWatts to Watts | uWatts | <1uWatt | <1uWatt | unknown |
| OS Supported | Linux, Windows | Contiki, eCos, nuttX, mbedOS, embOS | Contiki, eCos, nuttX, TinyOS, embOS, no OS | Contiki, nanoRX, nuttX, TinyOS, embOS, no OS | NA |
| Asymmetric Crypto Supported | Yes | Yes | Yes (EC-Few Curves) | Yes (EC-Few Curves) | NA |
| Program Language Supported | C, C++, Python, GO | C,GO, JavaScript | C, C++ | C, C++, Asm. | Asm. |
| Examples | AM3358 ARM Cortex-A8, ARMv8 | Arm Cortex M3, ARM7 | AVR16, PIC24F | AVR8, AT89C51, tinyAVR | AMD Am2900, Atmel MARC4 |

## III. Security Model and Threat Model

In this section, we provide a security and threat model for software updates of IoT devices. We identify attacker goals, capabilities, and threats. Figure 2 shows the threat model of a software update for low-level IoT devices.

**Attacker's Goal and Capabilities**: One goal of an attacker in a software update scenario is to provide a malicious image to the device, such that it is successfully installed. We take into account two potential attackers: an attacker who wants to install malicious updates, and a rogue former SUP. Both attackers may have various capabilities in our model. First,

the attacker has the basic information about the device like the company and model number, what the update manifest data looks like, and information about the image. The attacker may try to act as a man in the middle, and when the gateway device sends a query for the software update to the SUP over the internet, the attacker provides the malicious manifest data and update for the device. An attacker may also try to send the malicious manifest data or image directly to the IoT device over a low-power wireless channel, e.g., Bluetooth. Second, an attacker can be a former SUP that was providing services for the devices, but has now gone rogue. Such a former SUP has all the information related to the user's devices and can use that to deploy a malicious software image.

T1-T8 below are the threats that we have considered to derive the security requirements for the onboarding and secure software update for Class IV IoT devices [8] [11] [12]. Here, *valid software image* means non-malicious, signed from the legitimate SUP, and intended for the particular device.

T1: **Replay Attack/Old Firmware** An attacker can 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 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 for a different type of device.

T4: **Unauthenticated Update** An attacker attempts to spoof a SUP, sending a malicious software update. This may succeed if there is no method for verifying integrity and data origin authentication.

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

T6: **Non-Ephemeral Keys** At some point over the expected lifespan of IoT devices (5-15 years), an attacker aims to deduce the private key of a device's public-private key pair with available computational capabilities and forge a signature on a rogue update.

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 provisioned during manufacturing, has gone rogue. A rogue SUP uses this to send an old software image or a malicious software update.

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

We make the following assumptions for the prototype:

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.

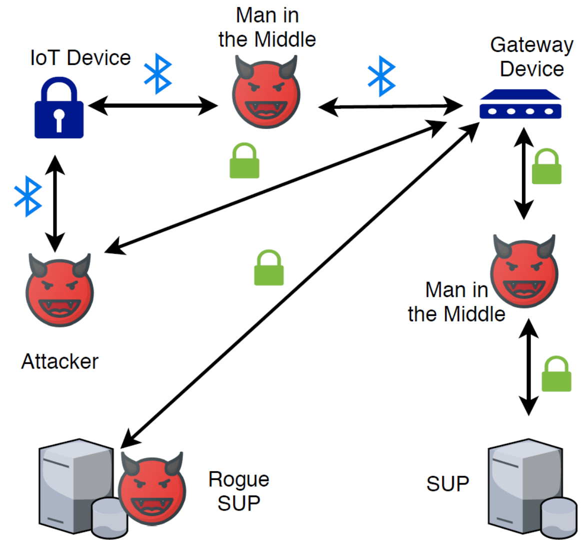



Fig. 2. Threat Model for Onboarding and Software Update

A2: Each IoT device generates an elliptic curve (EC) public-private key pair $(e1_D, d1_D)$ during initial 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 initial initialization or reset, which are 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 keeps all signature private keys for signing the software of different IoT devices offline. These are used to sign manifest data and software images.

A6: The verification public key $(v_P)$ of the manufacturer is embedded in base image stored in ROM of device D during the manufacturing process. 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 $(I_{T(D)})$ 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 outside of our scope. The user uses this list to check for a new SUP in case of initial 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 base image for all IoT devices of same type/model (for a given software update) by the manufacturer.

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

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 paper.

A14: All physical devices (i.e., IoT device and Gateway device) can only be reset manually.

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

## IV. Design and Implementation

A primary goal of our prototype is to prevent exploitation of software vulnerabilities of IoT devices by facilitating a secure automatic software update. The solution provides robust security against identified threats in Section III. Software update for Class IV IoT devices is a two-step process. First, on-boarding is achieved with the help of the key management scheme. For this, we use four algorithms: Elliptic Curve Diffie-Hellman (ECDH, 256 bit key, for session key establishment), AES-256 (for session key), SHA-256 (for key confirmation) and RSA(2048 bit key, for encryption/decryption). Second is the software update. This involves the transfer of software image from SUP to an IoT device. We use the elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA) [19], i.e., Ed25519 using SHA-512 and Curve25519 for signing and verification of manifest data and software image. We use the Arduino Cryptography Library (updated on November, 2018 with version 0.2.0) for the above algorithms.

TABLE II
SYMBOLS AND TERMINOLOGY USED IN DESIGN AND IMPLEMENTATION

| **Notations** | **Variables or Symbol Meaning** |
|---|---|
| $IoTData_D$ | IoT device Bluetooth MAC-address (16 byte), IoT device unique secret password (8 byte) |
| $knownData_D$ | IoT device model number (16 byte), IoT device manufacturer (16 byte), curVersion (16 byte), reqVersion (16 byte used for update failure) |
| $curVersion$ | Current software version running on IoT device |
| $reqVersion$ | Required software version for IoT device (Used in update failure) |
| Manifest Data | IoT device model mumber (16 byte), IoT device manufacturer (32 byte), Software version (16 byte), Timestamp (16 byte, populate during signature) |
| $M_{T(D)}$ | Manifest data of software image for the particular IoT device model |
| Software Image | IoT device model number (16 byte), IoT device manufacturer (32 byte), Timestamp (16 byte, populate during signature), verNextKey, Software version (16-byte), Binary |
| $I_{T(D)}$ | 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 secret password |
| $W$ | Symmetric key (AES-256) derived from IoT device unique secret password $w_D$ |
| $k$ | Symmetric session key (AES-256) shared by Gateway device and IoT device |
| $K_G = K_D = K$ | Long term shared secret (AES-256) generate using EC public private keys |
| $e1_D$, $d1_D$ | IoT device public-private (256 bits) keys for ECDH |
| $e1_G$, $d1_G$ | Gateway device public-private (256 bits) keys for ECDH |
| $e_G$ | RSA encryption public key of Gateway device with key size of 2048 bits |
| $d_G$ | RSA Decryption private key of Gateway device with key size of 2048 bits |
| $v_P$ | Elliptic curve verification public key (Ed25519 [20]) 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 |
| $B_D$ | Bluetooth MAC-address of IoT device |
| $E_K(x), D_K(x)$ | E, D denote generic encryption and decryption either symmetric or asymmetric according to the type of key used as subscript |
| $V_K$(x) | Signature verification on x using key K |
| $S_K$(x) | Signature on x using key K |
| $Req$(x) | Send a request for data x |
| $NULL$ | Variable is empty |
| $T$(D) | Device type of D |
| $f$(x) | f generates a symmetric 256 bit key from 8 byte password x |

### A. Components

We now discuss the types and capabilities of different software and hardware components used in the system design based on Figure 1.

*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 unique authentication key available in a sealed booklet of the device, in a QR-code format. Our implementation uses an Arduino board, which is an 8-bit microcontroller (ATMega2560) with 16 MHz clock frequency programmed using C++ libraries. The Bluetooth module (i.e., HC-05) is in listening mode. 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 in Section III).

*Smartphone Application (SmartApp):* This is used to register the devices to the SUP. It is programmed using the React platform running over Samsung Galaxy Note 2 with 2GB RAM and 32 GB memory.

*Gateway Device:* A Class I IoT device with a custom RTOS (real-time OS) including capabilities to handle multiple smart home devices. It serves as a communication portal for the IoT devices, which do not themselves have capabilities to communicate through the Internet. The Gateway communicates with IoT devices over BLE and with the SUP using Wi-Fi through a router. The Gateway can address many IoT devices in a sequential order. It has only one physical port used to configure it with the help of a laptop or desktop manually. It is implemented using the Raspberry Pi 3B+ module (Broadcom BCM2837B0 quad-core A53 (ARMv8)), a 32-bit board with Bluetooth and Wi-Fi module, programmed with Node.js. 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 the software up-

date 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. SUPs store the device information (serial number, weak password, etc.) containing device-specific details in a SUP database. SUP generates the public-private key pair used for signing the software and firmware. It is a software module programmed in Node.js and uses MongoDB as a database. In our prototype, SUP runs on a laptop with specifications: i7-8th Gen processor, 16 GB RAM and 512GB hard disk. The SUP is denoted by subscript 'P' in our notation.

To establish trust between all components, our design involves a secure onboarding protocol with authenticated key management. Our IoT device (D), do not have any physical input/output interface. D and G do not have any pre-shared secret. We use the internet for sharing the secret between the devices.

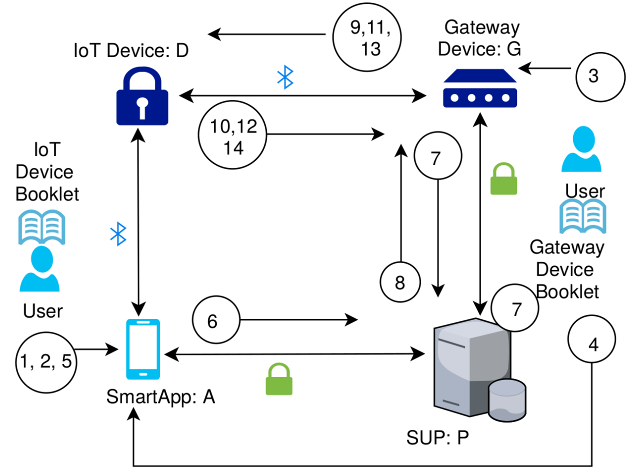
### B. Onboarding using Key Management (Chain-of-Custody)

Chain-of-custody is a concept that enables authenticated key exchange and helps in onboarding the IoT device and the Gateway device. It is based on 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 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.

As mentioned in Section IV-C, the IoT device does not have any physical input/output interface for user interaction, and the Gateway device has only one physical port for manual configuration through a laptop or desktop. A Class IV device supports Bluetooth 4.0 protocol which is not secure, therefore, we need a method of onboarding to establish a secure communication channel between the device and the Gateway. 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 its availability in supporting software tools, and henceforth refer to it as "TLS".

We use RSA [21] with a key-size of 2048 bits for sending encrypted $IoTData_D$ ($w_D$ and $B_D$) from SmartApp to the Gateway device through the SUP. Due to the presence of another component (i.e., SUP) in between the SmartApp and the Gateway device, a direct key-establishment protocol is not feasible. There is no direct communication between the Gateway and the SmartApp. 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. In practice, the Elgamal-EC is preferred in IoT environments for encryption and decryption of data, but due to its unavailability in supporting software tools, we are using RSA here.

We use ECDH with Curve25519 [20] for establishing the session-key ($k$) between the device and the Gateway based on a shared secret ($K$) derived during key-establishment. We use SHA-256 for key-confirmation with k and W (i.e., W = f($w_D$)). We use long-term shared secret (i.e., AES-256 ($K$)) for encryption/decryption of session key. This reduces the computational burden on the Class IV device for public-key operations conducted multiple times a day. The following is




*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 II.*

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

6. A$\longrightarrow$ P: M, $N_D$; P add M and $N_D$ in the queue maintaining user IoT devices list
7. G$\longrightarrow$ P: Next M; P checks the next available M in the queue maintaining user IoT devices list and if available, dequeues it and store locally.
8. P$\longrightarrow$ G: M; 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$\longrightarrow$ 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$\longrightarrow$ 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 $d1_G$. Use $K$ to recover session key k. G also generates W = f($w_D$).
11. G$\longrightarrow$ 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$\longrightarrow$ 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 $\longrightarrow$ D: $E_k(knownData_D)$; D recovers empty $knownData_D$ and populates it with device model number, manufacturer, $curVersion_D$ and $reqVersion_D$.
14. D $\longrightarrow$ G: $E_k(knownData_D)$; G recovers $knownData_D$ and stores it alongside $IoTData_D$

Fig. 3. Authenticated Key Establishment

the process for authenticated key management (Figure 3):

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 username, user-to-SUP-password, and

email address, and then logs in after successful account creation.

3. After a successful login, the user initializes the Gateway and configures the SUP's URL, user's username, and user's password of the SUP account with a laptop. User initializes the Gateway device. On initialization, the Gateway device generates an EC public-private key pair ($e1_G, d1_G$) and RSA public-private key pair ($e_G, d_G$).
4. The user scans the RSA public key $(e_G)$ of the Gateway device with the SmartApp (QR code) from the laptop screen during the Gateway configuration and saves it in local storage on the smartphone.
5. The user initializes the IoT device. On initialization, the device generates an EC key pair $(e1_D, d1_D)$ for ECDH. The user enters the serial number $(N_D)$ and $IoTData_D$ which includes the unique password $(w_D)$, and the Bluetooth MAC-address $(B_D)$ of the IoT device by scanning the QR bar code from the IoT device sealed booklet.
6. The SmartApp encrypts the $IoTData_D$ of the IoT device with the public key of the Gateway ($e_G$), and sends to the SUP over TLS along with serial number $(N_D)$ of device. The SUP stores $N_D$ and encrypted $IoTData_D$ in the user's queue of the IoT devices.
7. The Gateway device requests the SUP for next $IoTData_D$ over TLS. The SUP checks the next available encrypted $IoTData_D$ in the queue and dequeues it. This process continues until complete queue is empty and repeats after every 24 hours to check for new IoT device information .
8. The SUP sends the stored encrypted $IoTData_D$ of the next IoT device to the Gateway device over TLS. The Gateway device decrypts the $IoTData_D$ ($w_D$ and $B_D$) using the RSA private key $(d_G)$ and stores them.
9. The Gateway device sends its EC public-key $(e1_G)$ to the IoT device using the Bluetooth MAC-address $(B_D)$. The IoT device receives $(e1_G)$ and generates a shared secret key ($K_D$, AES-256) using the EC public-key $(e1_G)$ of the Gateway device 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 $K_D$.
10. The IoT device sends its public-key $(e1_D)$ and encrypted session key to the Gateway. The Gateway receives $(e1_D)$ and generates shared secret key ($K_G$, AES-256) using the EC public-key $(e1_D)$ of the IoT device and the EC private-key $(d1_G)$ of itself such that $K_G$ = $K_D$ = $K$. It decrypts session key (k).
11. The Gateway sends double hash of session key (k) concatenated with W (W = f($w_D$)) to the IoT device. The IoT device generates W = f($w_D$), verifies the hash, and aborts in case of mismatch. This convinces the IoT device that the Gateway knows W.
12. The IoT device sends the single hash of session key (k) concatenated with W to the Gateway. The Gateway validates the hash. If it matches, it accepts K as the session key; else protocol fails. This convinces the Gateway device that the IoT device knows W.
13. The Gateway device sends the request for the $knownData_D$ to the IoT device encrypting it with K. The IoT device recovers $knownData_D$ and populates it with device model number, manufacturer, $curVersion$, and $reqVersion$, and encrypts it using session key.
14. The IoT device sends the encrypted $knownData_D$ to the Gateway. The Gateway decrypts the $knownData_D$ and stores in its database with regards to the $IoTData_D$.
15. When the Gateway wants to communicate with the IoT device or vice versa, it generates a new session key (i.e., a symmetric key) and repeats steps 12 and 15.

In case of a manual reset of any D, D and G both must repeat steps 9 through 15 for each D. In case of the G manually reset, all configuration is deleted which is stored in EEPROM, and steps 1 through 15 are repeated for all IoT devices. In the case of reboot of either D or G, there is no data loss except session key $(k)$ and long-term shared secret key $(K)$. D and G both must repeat steps 9 through 15 for each D. We are using this chain of custody of evidence $(w_D)$ for the establishment of trust between the IoT device and the Gateway device and key establishment.

### *C. Software Update*

Smart home devices commonly require use of a smartphone for the initial configuration of a device that does not have any user input or output interface. 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 [20]. Software updates for IoT devices might occur monthly or annually. Therefore, even though public-key algorithms are computationally costly on Class IV devices, they are feasible with Ed25519 with a key-size of 256 bits (but problematic for RSA with 2048 bit key).

We use the concept of key-locking [7], where each software image includes a next-update verification key for verifying the integrity and authenticity of the next software image. Each software version has its own public private key pair for signature and verification. Each image verification key is sent to the IoT device embedded in the previous image. Therefore, the lifespan of each signature-verification key pair is the period until availability of a new update image. A manufacturer provisions the base image with embedded verification key in the ROM during the manufacturing process. All IoT devices with the same model number use the same software image version. Any SUP who wants to provide updates for IoT devices has to ask the manufacturer of the devices for the signing key of the base image, and it is the responsibility of the manufacturer to approve the SUP. The manufacturer authorizes the SUP by sharing its signature key of the base image of the particular IoT device model. The manufacturer also publishes the list of trusted SUPs for the user, so that the user can configure the gateway device for future updates. Figure 4 shows a flow chart of the software update after authenticated key management. One assumption related to the

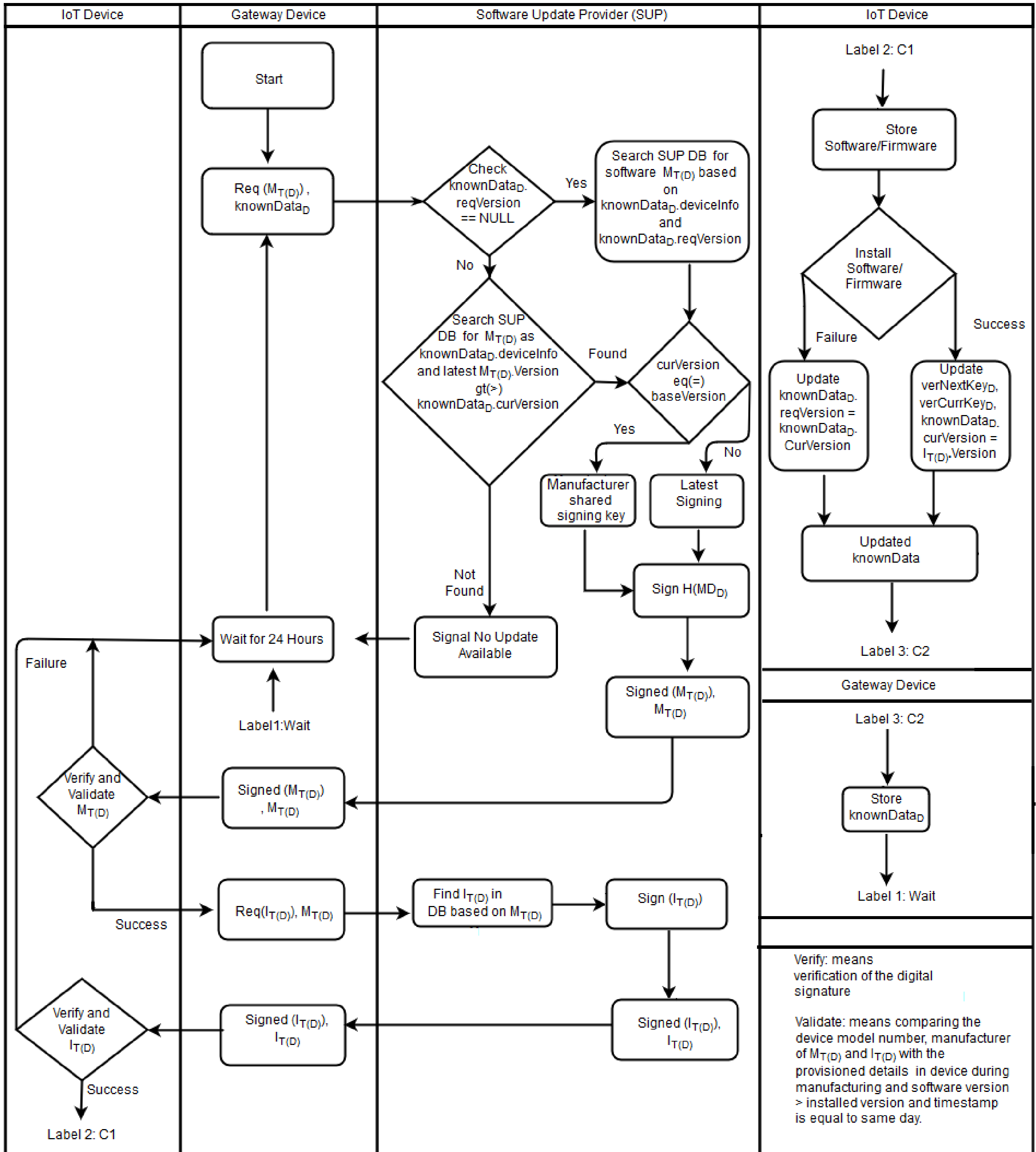



Fig. 4. Software update flow chart. See Table II for notation

Also, SUP is responsible for verifying a base level of functionality and software quality, essentially checking that the software update is acquired from trustworthy sources. Each new software image contains the verification key to allow signature verification of the subsequent software image. In our model the IoT device maintains only two verification keys in EEPROM, for the current working software image ($verCurrKey_D$) and the upcoming software image ($verNextKey_D$). With every successful update, the device

updates the verification keys. We query SUP for the new software updates every 24 hours; but this is configurable (actual updates are expected at most monthly).

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 the higher software version, it generates a time stamp and appends it with $M_{T(D)}$. It then checks that 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 timestamp. *Timestamp* 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 timestamp which is filled before signature. The SUP signs the $I_{T(D)}$, with the same signature private key used to sign $M_{T(D)}$. The SUP sends M2 = (($S_{s_P}(I_{T(D)})$), $I_{T(D)}$) to G. G forwards the 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 ($V_{v_P}$(M2)) the digital signature using $verNextKey_D$ and validates the information about the device (e.g., checks D's model number, manufacturer, software version greater than $curVersion$ and timestamp). 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 to G of the successful software update with new $knownData_D$ by modifying the $curVersion$ field. 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 during the update process, our device cannot work either with a current image or new image as some files have changed. D can only store one software image in EEPROM other than the base image provisioned in ROM during manufacturing. In the case of an update installation failure, D reverts back to run the base (initial) image from ROM and sends a failure notice to G along with $knownData_D$ after modifying the $reqVersion$ with $curVersion$ (which is working software version before update failure). 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. The SUP then searches in its database for the manifest data and software image if version equal to $reqVersion$ sends it to the G after signing it with the respective signature key. D uses its $verCurrkey$ of the last working version to verify the digital signature of the software image in case of update failure.

As an IoT device has an expected lifespan of 5-15 years, an initial public-private pair for signing and verification may become insecure due to the computational capabilities of a future attacker. 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 and clears its EEPROM. However, after reset, D requests G for software update, and 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.

### D. Software Update Provider Change

The ability to accommodate changing the SUP addresses scenarios where the manufacturer or current SUP goes out of business or the user is not satisfied with the current SUP. The user selects the new SUP from the list of trusted SUPs shared by the manufacturer, who already has the signature private key of the corresponding verification key embedded in the base image of the IoT device. SUP change requires the user to reset the IoT device which clears its EEPROM that contains the relevant information and sends it back to the base image provisioned during manufacturing. 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 new SUP information. The IoT device and Gateway need to repeat the process of onboarding discussed in section IV. After successful onboarding, G requests a software update for D from the new SUP. The SUP provides D, through G, with the latest software image for D, and this allows the user to change the software update provider.

## V. Analysis

This section discusses the evaluation and limitations of the model, and provides an informal security analysis.

### A. Implementation 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 and we built a test application for the IoT device. We 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 ($V1 \longrightarrow V2 \longrightarrow V3$). Note that on testing, we did not actually 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 reinstall the previous working version (i.e., V3) on the IoT device as the two verification keys were kept in the EEPROM of the device. The IoT device automatically performed the update and recovery from update installation failure. Our model is also suitable for all classes I-IV of the IoT devices as there is no particular hardware requirement. A Class I or Class II IoT device with higher processing power would be able to perform the functionality of the Gateway device as well (which includes communicating with the SUP over TLS). With the Class III and Class IV IoT device, we still need a Class I device serving as a gateway.

**Performance Evaluation (Timing)**

The time used for public key operations is non-trivial but tolerable. Key generation operations are performed once during the initialization or after reset. Similarly, signature verification of software images is relatively infrequent, with expected release of software updates for Class IV IoT devices once or twice a year, and at most, monthly. Bernstein [22] 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 key-size of 256 bits. Bernstein performed his test on 64-bit 2.4 GHz Intel Westmere (Xeon E5620) CPU and 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. In our design, the IoT device is only performing the signature verification, but for completeness we also tested the time to carry out the digital signature.

ECDH with Curve25519 is used for key establishment. ECDH algorithm [23] takes two scalar multiplication. Bernstein [24] showed that curve25519 is faster than other elliptic curves.

AES counter mode (AES-GCM) is used with a key of 256 bits as a session key for encryption and decryption. There is no time difference in the encryption and decryption process when using AES-GCM [25]. Table III summarizes the time taken by symmetric and asymmetric algorithms to process a single byte or operation. The method used to record these timings was as follows: We performed encryption and decryption using AES-GCM with a key size of 256 bits. This was done ten times over 64 bytes of data. The average result was taken (over the 10 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 took the average result. In the case of ECDH 256 bits public-private key generation and key agreement, we 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 same. In signature and verification with increase in data size from 100bytes to 1KB-3KB, there is a subsequent small increase in timings due to hashing (approx. 50 to 150 msec). Timing is measured using the serial monitor tool of Arduino IDE software calculated over Arduino ATMega2560 micro-controller with 16MHz clock frequency.

TABLE III
TIMING EVALUATION ON 8-BIT ATMEGA2560 WITH 16 MHz CLOCK

| **Operation** | **Time Taken** |
|---|---|
| AES-GCM-256 encryption | 123.52 $\mu$sec per byte |
| AES-GCM-256 decryption | 123.06 $\mu$sec per byte |
| Digital Signing (Ed25519, 256 bit ECC) | 6.0035s per signature |
| Signature verification (Ed25519, 256 bit ECC) | 9.7786s per verification |
| ECDH 256 bits public-private key generation (Curve 25519) | 3.3309s per pair generation |
| ECDH (key agreement, Curve 25519) | 3.3325s per key agreement |

### B. *Security Analysis*

In this section, we argue informally how our design addresses the earlier-stated threats (section III).

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 timestamp (filled in during signature of $M_{T(D)}$) and $I_{T(D)}$ for the software update. $M_{T(D)}$ and $I_{T(D)}$ are signed by the SUP. Each software version has its own corresponding public-private key pair. T1 and T2 are mitigated by the verification of signature and validation of software version and timestamp present in $M_{T(D)}$ and $I_{T(D)}$.

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

SA3: [Threats T4] Verification of the signature on $M_{T(D)}$ and $I_{T(D)}$ during software update mitigates this threat.

SA4: [Threats T5] ECDH key exchange addresses man-in-the-middle attacks. Knowledge of W is tested as part of onboarding in steps 11-12.

SA5: [Threat T6] T6 is mitigated by the key-locking mechanism.

SA6: [Threat T7] T7 is reduced somewhat by keeping the IoT device information encrypted at the SUP to avoid exploitation of the IoT device secret ($w_D$) used in key confirmation between the device and Gateway.

SA7: [Threat T8] T8 is mitigated by generating new RSA and EC keys for the Gateway and IoT device during every reset. A15 rules out physical device attack.

**Gateway Device Analysis**

The Gateway device is a Class I device (Table I). By assumption A14, the Gateway device and the IoT device can only be reset manually. Therefore, no remote attacker is able to reset the device. The Gateway device can only be configured using a physical connection between the laptop/desktop and the port of the Gateway device. The end-user configures the Gateway device with the URL of the SUP from the IoT device manufacturer's website (assumption A9). Due to pull model, the Gateway device is the one requesting software update on behalf of the IoT device. The communication between the Gateway device and the SUP is over TLS 1.2.

### C. Limitations

L1: Our design is unable to handle legitimate software version downgrade.

L2: The SUP must maintain all signature private keys for every IoT device's type/model.

L3: We use the pull model for a software update. End-device polling consumes gateway device power resources.

L4: We do not handle the passive man-in-the-middle attack. An attacker might read the unencrypted image file and later exploit any software vulnerabilities present in the image.

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 on device reset and remains static otherwise.

L6: RSA key-pair is also not ephemeral. A new RSA key pair is generated during initialization or on device reset.

L7: If the IoT device booklet is lost, we are unable to recover the unique password used in the ECDH key confirmation, which cause in protocol fails.

L8: We do not address vulnerabilities in software for legacy devices.

L9: A complete and formal analysis has not been done.

L10: Baud rate (i.e., 9600) of IoT devices is a limitation if there is a large number of IoT devices communicating with Gateway device.

## VI. Related Work

We focus our discussion of related work on academic and industrial research exploring IoT security related to onboarding and software/firmware update.

### A. Software Update

For IoT devices, secure software and firmware update is a major challenge. Tschofenig et al. [8] discussed the challenges and proposed solutions faced for software and firmware updates. The main focus of the workshop was to find methods for secure software updates for constrained devices [5]. Tschofenig et al. [11] [12] explored firmware update architectures for constrained devices [14]. In these drafts, a common approach is the use of public-key infrastructure for digital signature for checking the integrity and data origin authentication, and using secure boot for keeping cryptographic keys and basic functionality in case of update failure. Wurster [6] proposes a method for protecting binaries already installed on the system from any malicious modification using a digital signature and enhanced this concept for key-locking [7]. Public keys are embedded in the binaries to verify the integrity of future software update. They use a trust on first use (TOFU) model for basic initial installation. Baton [26] suggested a modification in smartphones app installation framework which helps application developers to transfer the signing authority of an application to the new developer in a secure manner without any user intervention. They consider renewing signing keys by chaining them, but the solution does not address stolen signing keys. They also put their trust of checking the integrity and authenticity of the software updates on the application developer and OS.

BRSKI [27] provides a solution for securing devices with zero-touch methods using X.509 certificate. This solution is generally designed for non-constrained devices [14] such as large router platforms in data centers.

Choi [28] introduced a secure firmware validation and update scheme for consumer devices in a home networking systems by utilizing ID-based mutual authentication and key derivation to distribute a firmware image securely. Firmware is divided into a series of chunks called fragments, used to create the hash chain [29]. Hash chaining is used for authenticity of the fragmented firmware image.

Uptane [6] is the first software update framework for automobiles to address automotive-specific vulnerabilities. Uptane has additional features over TUF like additional storage for recovery in case of software infection of the Electronic Control Unit (ECU), maintaining a vehicle manifest to keep track of versions installed on different ECU's and a time server in case an attacker tries to delay the update for an indefinite time.

### B. Onboarding

The NIST report on lightweight cryptography [16] for 2017 mainly suggested the use of symmetric cryptography for 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 2015, in which performance of different cryptography algorithms was analyzed on ARM-based microprocessors [30].

Ozmen [17] proposed a low-cost asymmetric algorithm for wireless IoT devices using micro ECC library and NIST-recommended secp192 curve. In implementation, they tested the proposed solution with 8-bit microcontroller and show the performance of key-exchange, integrated encryption and hybrid construction and use the concept of self-certification.

J-PAKE [31] is password-authenticated key exchange protocol using juggling. It offers a security proof and is built on an earlier mechanism of Schnorr [32]. It is also compatible

with elliptic curves. It protects against the online and offline dictionary attacks and maintains forward secrecy.

CBOR web Token (CWT) [33] is built from the JSON Web Token (JWT). Concise Binary Object Representation (CBOR) is mostly used for IoT devices. It is a concise means of secure data transfer between two parties using CBOR encryption and signature. CWT is used with AES-128, AES-256 and ECDSA.

Device pairing is an important concept for trust establishment of wireless devices. The Resurrecting Duckling model [34] is based on the idea of trusting, and pairing with, the first device that makes contact with a newly 'woken up' device. Kumar [35] compared secure device pairing methods. LED button or vibrate-button is preferred in the scenarios where the device does not have any audio-visual interface.

## VII. Future Work and Conclusion

The ubiquity of software vulnerabilities on all classes of IoT devices illustrates the need for a mechanism for (ideally, 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 a Class IV level IoT device (i.e., ATMega2560 8-bit micro-controller with 16MHz clock) by providing a practical instantiation which has been tested for onboarding and software update. Our design uses the concept of key-locking for software update, a particularly good fit to IoT software update, as it addresses the issue of a manufacturer going out of business. Our model demonstrates that EC-based public key algorithms (security equivalent to RSA-2048) can be used with Class IV IoT devices. Our onboarding system design allows secure transport of a secret from the IoT device to the Gateway device through the conventional internet and maintains integrity and data origin authenticity of the software update. This model can be enhanced to use the push model for software update and encryption of the image.

**Acknowledgement:** The authors thank the anonymous referees whose comments helped improve this work. Paul C. van Oorschot is Canada Research Chair in Authentication and Computer Security, and acknowledges NSERC for an NSERC Discovery Grant.

## References

[1] M. Antonakakis *et al.*, “Understanding the Mirai botnet,” in *USENIX Security Symposium*, 2017.
[2] K. Zetter, “How the NSA’s firmware hacking works and why it’s so unsettling.” https://www.wired.com/2015/02/nsa-firmware-hacking/, Feb 2015.
[3] A. Cui, M. Costello, and S. Stolfo, “When firmware modifications attack: A case study of embedded exploitation,” in *NDSS*, 2013.
[4] Jackson, “Sb-327 information privacy: connected devices,” *California Legislative Information*, Sep 2018.
[5] C. Bormann, M. Ersue, and A. Kernen, “Terminology for Constrained-Node Networks.” RFC 7228, May 2014.
[6] G. Wurster and P. C. van Oorschot, “Self-signed executables: Restricting replacement of program binaries by malware,” in *HotSec*, 2007.
[7] P. C. van Oorschot and G. Wurster, “Reducing unauthorized modification of digital objects,” *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 191–204, 2012.
[8] H. Tschofenig and S. Farrell, “Report from the Internet of Things Software Update (IoTSU) Workshop 2016.” RFC 8240, Sept. 2017.
[9] P. Kotzias, S. Matic, R. Rivera, and J. Caballero, “Certified PUP: Abuse in Authenticode Code Signing,” in *ACM*, CCS ’15, 2015.
[10] R. Wash, E. Rader, K. Vaniea, and M. Rizor, “Out of the loop: How automated software updates cause unintended security consequences,” in *10th Symposium On Usable Privacy and Security*, 2014.
[11] B. Moran, M. Meriac, and H. Tschofenig, “A Firmware Update Architecture for Internet of Things Devices,” Internet-Draft draft-moran-suit-architecture-02, IETF, Jan 2019.
[12] B. Moran, H. Tschofenig, and H. Birkholz, “Firmware Updates for Internet of Things Devices - An Information Model for Manifests,” Internet-Draft draft-ietf-suit-information-model-02, IETF, Jan. 2019.
[13] J. Zhu, “A Secure and Automatic Firmware Update Architecture for IoT Devices,” Internet-Draft draft-zhu-suit-automatic-fu-arch-00, IETF, Mar 2018.
[14] D. Barrera and P. van Oorschot, “Secure software installation on smartphones,” *IEEE Security Privacy*, vol. 9, pp. 42–48, May 2011.
[15] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, “Operating systems for low-end devices in the internet of things: A survey,” *IEEE Internet of Things Journal*, vol. 3, pp. 720–734, Oct 2016.
[16] H. Tschofenig and M. Pegourie-Gonnard, “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, Jul 2015.
[17] M. O. Ozmen and A. A. Yavuz, “Low-cost standard public key cryptography services for wireless iot systems,” in *Workshop on Internet of Things Security and Privacy*, ACM, 2017.
[18] A. Salman, W. Diehl, and J. Kaps, “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)*, Dec 2017.
[19] S. Josefsson and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA).” RFC 8032, Jan. 2017.
[20] A. Langley, M. Hamburg, and S. Turner, “Elliptic Curves for Security.” RFC 7748, Jan. 2016.
[21] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, “PKCS #1: RSA Cryptography Specifications Version 2.2.” RFC 8017, Nov. 2016.
[22] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, “High-speed high-security signatures,” *J. Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.
[23] D. R. L. Brown, “Standards for Efficient Cryptography.SEC 1: Elliptic Curve Cryptography.” http://secg.org/sec1-v2.pdf, May 2009. SECG-SEC-v2.
[24] D. J. Bernstein, “Curve25519: New Diffie-Hellman Speed Records,” in *PKC, Apr 24-26*, pp. 207–228, 2006.
[25] STMicroelectronics, “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, Sep 2013. (Accessed on 06/23/2019).
[26] D. Barrera, D. McCarney, J. Clark, and P. C. van Oorschot, “Baton: Certificate agility for android’s decentralized signing infrastructure,” in *ACM*, WiSec, 2014.
[27] M. Pritikin, M. Richardson, M. H. Behringer, S. Bjarnason, and K. Watsen, “Bootstrapping Remote Secure Key Infrastructures (BRSKI),” Internet-Draft draft-ietf-anima-bootstrapping-keyinfra-18, IETF, Jan. 2019.
[28] B. Choi, S. Lee, J. Na, and J. Lee, “Secure firmware validation and update for consumer devices in home networking,” *IEEE Transactions on Consumer Electronics*, vol. 62, pp. 39–44, Feb 2016.
[29] A. J. Menezes, S. A. Vanstone, and P. C. van Oorschot, *Handbook of Applied Cryptography*. CRC Press, 1996.
[30] P. A. Grassi, M. E. Garcia, and J. L. Fenton, “NIST Special Publication 800-63B: Digital identity guidelines,” Jun 2017.
[31] F. Hao, “J-PAKE: Password-Authenticated Key Exchange by Juggling.” RFC 8236, Sept. 2017.
[32] F. Hao, “Schnorr Non-interactive Zero-Knowledge Proof.” RFC 8235, Sept. 2017.
[33] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, “CBOR Web Token (CWT).” RFC 8392, May 2018.
[34] F. Stajano and R. J. Anderson, “The resurrecting duckling: Security issues for ad-hoc wireless networks,” in *Security Protocols Workshop, Apr 19-21*, 1999.
[35] A. Kumar, N. Saxena, G. Tsudik, and E. Uzun, “A comparative study of secure device pairing methods,” *Pervasive and Mobile Computing*, vol. 5, no. 6, pp. 734–749, 2009.