



國立台灣科技大學
資訊管理系

碩士學位論文

Video Transition Smoothing for ASL Dataset

研 究 生：Yulia

學 號：M10609803

指導教授：Prof. Chuan-Kai Yang

中華民國一百零八年六月二十一日

Abstract

The rapid growth of IoT devices in past few years brings convenience in human life. The huge amount of installed IOT devices makes the device manufacturer (vendor) difficult to maintain the IoT devices. One way to maintain the IoT device is through the firmware update. Vendor can add new functionality, enhance the security and re-configure the IoT device through a firmware update. However, the firmware update process is not without risk. There are several well-known attacks those targets the firmware update process. In this thesis, we propose a robust and lightweight framework that ensure the firmware update security using skipchain technology. Utilizing the skipchain's forward link, the proposed protocol can do peer-to-peer firmware update verification efficiently. A prototype based on the proposed framework is constructed and evaluated. Moreover, our proposed framework is also proven to be secure and could withstand some well-known attacks.

Keywords: Skipchain, Firmware Update, Blockchain, Peer-to-Peer Verification

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Lo Nai-Wei, whose encouragement, thoughtful guidance, brilliant ideas and wholehearted supports enabled me to develop an understanding of this research subject. His every advice has been very valuable to my own professional growth and will not be forgotten. I would also like to extend my gratitude to all committee members in my doctoral defense for their valuable comments.

It is a pleasure to thank the members of Integrated Digital Service Laboratory (賴孟洸, Power Li, Eric Tan) and all my friends for their supports and inspirations. Special thanks to Alexander Yohan, because of his passionate guidance, advice, and contributions in everything since my first day in Integrated Digital Service Laboratory.

Finally, I am heartily thankful to my family for their unconditional love, continuous supports and cheers.

Contents

Recommendation Letter	i
Approval Letter	ii
Abstract	iii
Acknowledgements	iv
Contents	v
List of Figures	viii
List of Tables	x
List of Pseudocodes	xi
1 Introduction	1
2 Literature Review	7
2.1 Cyber Attack on Firmware and Remote Firmware Update for Embedded Device	7
2.2 Blockchain	10
2.3 Skiplist	12
2.4 Skipchain	14
2.5 Blockchain-based Firmware Update Framework	16
3 System Environment and Protocol Designs	18
3.1 Assumptions	18

3.2	Skipchain Overview	19
3.3	Architecture Design	22
3.4	Protocol Design	30
3.4.1	Firmware Update Verification Protocol	32
3.4.2	Firmware Update Peer-to-Peer Verification Protocol	34
3.4.3	Firmware Update Execution Protocol	37
4	Prototype Design and Implementation	39
4.1	Prototype Design	39
4.1.1	Key Exchange Procedure	39
4.1.2	AES Encryption Function	43
4.1.3	AES Decryption Function	44
4.2	Prototype Implementation	45
5	Security and Performance Analyses	47
5.1	Informal Security Analysis	49
5.2	Protocol Verification Using Scyther Tool	54
5.2.1	Data Secrecy	55
5.2.2	Aliveness	55
5.2.3	Non-injective Agreement and Non-injective Syn- chronisation	56
5.3	Performance Analysis	56

5.4	Discussion	59
6	Conclusions	61
	References	63

List of Figures

2.1	Firmware Over The Air (FOTA) process diagram	9
2.2	Iterating nonce that make hashes value begin with a number of zero value [1]	11
2.3	Nakamoto's chaining block diagram 2.3	12
2.4	Skiplist structure with span between two same-level nodes 2.4	13
2.5	(a) There is a node with height of two between two nodes with height of three. (b) There are two nodes with height of one between two nodes of height of two [2]	13
2.6	Skipchain structure [3]	15
3.1	CoSi protocol architecture [4]	21
3.2	Firmware Over-the-Skipchain (FOTS) process diagram . .	23
3.3	Skipblock structure diagram	24
3.4	Firmware update verification process	26
3.5	Firmware update peer-to-peer verification process	28
3.6	Peer-to-peer verification process	29
3.7	Pre-installed skipblock diagram	30
3.8	Firmware update verification protocol	32
3.9	Firmware update peer-to-peer verification protocol	35

3.10	Firmware update execution protocol	37
5.1	Scyther tool verification result	54

List of Tables

3.1	Notation used in the proposed protocol	31
4.1	The device specification for each role	45
5.1	Architecture design comparison of four blockchain-based firmware update framework	47
5.2	The comparison of security mechanism against cyber at- tack of four blockchain-based firmware update framework	49
5.3	The performance comparison of four blockchain-based firmware update framework	57
5.4	Notations used for time consumption on differing comput- ing operations	57
5.5	The execution time of several cryptographic operations . .	58

List of Pseudocodes

1	Client Key Exchange Initiation	40
2	Server Key Exchange Response	42
3	Client Key Exchange End	43
4	AES Encryption Function	44
5	AES Decryption Function	45

Chapter 1 Introduction

According to World Health Organization (WHO) [5], there are over 5% or about 466 million of world's population have suffered of hearing disability. This number consists of 432 million adults and 34 million children. It is also predicted that by 2050, there will be over 900 million people or one in every ten people will have the hearing disability.

A person can be said to have a hearing loss if he or she is not able to hear with a normal hearing thresholds of 25dB or better in both ears. The hearing loss may be divided into mild, moderate, severe or profound category. 'Hard of hearing' term refers to the individuals who have hearing loss ranging from mild to severe category. These people usually still can communicate using the spoken language and can get the benefit of hearing aids, cochlear implants and other devices. 'Deaf' refers to the people who mostly have profound hearing loss, which indicates very little or no hearing. These people often use Sign Language to communicate. Although Sign Language is used as the primary language for the Deafs, it also can be a way to communicate for: the Hard of Hearing, hearing individuals which unable to physically speak, the people who have trouble with spoken language due to a disability or condition, and the Deaf family members.

Sign languages are the natural languages developed by each Deaf community all over the world with their own grammar and lexicon, which means each community has their own grammar and lexicon [6]. This means that sign languages are unique for each region, not universal and not mutually intelligible.

American Sign Language (ASL) is a natural language which domi-

nates sign language of Deaf communities in the United States and most of Anglophone Canada. Not only North America, the dialects of ASL and ASL-based creoles are also used in many countries, including much West Africa and parts of South East Asia. ASL have some phonemic components, including movement of the face and torso together with the hands.

Regarding the importance of ASL for Deaf, Athitsos et al. [7] made the availability of ASL Lexicon Video Dataset, a public dataset containing video sequences of thousands of distinct ASL signs, along with the annotations of those sequences.

According to [?], Internet of Things (IoT) has become the main standard of low-power lossy network having constrained resource. IoT represents a network of sensor(s) embedded devices with the capability to communicate with other devices and perform a specific task. In 2014, Gartner predicted the number of connected IoT devices will exceed 20 billions by 2020 [8]. The r devices [9–11]. Cui et al. [9] showed a scenario of an attack on a remote printer through a modified firmware which resulting in covert network scanning, data ex-filtration, and malware propagation to other devices in the network. Miller [10] reverse engineered the original firmware and the firmware flashing process of a particular smart battery system, and installed a modified firmware to the corresponding system to control the operation of the corresponding smart battery and smart battery host. In 2015, Zetter [11] reported a hacking tool created by spy group *Equation* that is able to re-flash the firmware of a hard drive and replace it with a malicious firmware. These three attack scenarios show the importance to maintain the device's firmware up-to-date and to securely deliver the firmware to corresponding IoT devices.

Firmware is a certain set of data and basic commands installed in a IoT device to perform a specific task [?,12]. Every IoT device needs firmware to associate with other software to interpret the collected raw data into some meaningful context. The instruction in the firmware can be defined as set of programmed routines to handle different components in the associate IoT device.

In order to ensure the IoT device works properly and to reduce any existing vulnerabilities in the device, the device's firmware needs to be updated regularly. In addition, the vendor could add new functionalities and re-configure the corresponding device through the new firmware version. Generally, IoT devices have several characteristics as follow: low-power consumption, limited memory and computing capability. Furthermore, most of IoT devices could not establish direct connection to the Internet and require the assistance from gateway or router to perform the firmware update process.

In general, the device firmware is considered more secure compared to other software due to its proprietary nature [13]. However, it is reported more than 750.000 consumer "smart" appliances including home router, fridge, air conditioner, television, etc. had been compromised and used as bot to spam emails and distribute phishing in 2014 [14]. As these smart appliances are connected to the Internet through a home router, once the router is compromised then all the connected devices could be easily affected as well.

Blockchain [15] was originally created for peer-to-peer electronic cash payment transaction based on cryptography proof instead of trusting third party, such as Bank. Blockchain uses distributed ledger to record digital

transactions, in which these ledgers are distributed on decentralized peer-to-peer network. Each transaction stored in the blockchain's ledger is immutable, in which once the transaction is recorded in the ledger then it could not be modified by any entity in the blockchain network. In addition, all the transaction history recorded in the blockchain ledger are publicly traceable.

In the current blockchain technology, users could trace all transactions history recorded in the blockchain by synchronizing the block data to the latest version. This ability requires the user's device to be connected to the Internet and synchronize with the latest block. However, almost all IoT devices are unable to join the blockchain network due to their limitation in Internet connectivity and storage capacity. Therefore, this thesis proposes a secure and trusted peer-to-peer (p2p) firmware update framework for IoT environment.

In this thesis, IoT devices are categorized into two groups based on the device capability to connect with Internet. The first group is called online IoT device, in which the device belong to this category is able to connect to the Internet without the assistance of any intermediary devices such as router. Any IoT device that can only connect to the Internet through the assistance of intermediary device is belonged to the second group, called offline IoT device. As most of the existing firmware update mechanism targets the online IoT device, it is difficult to guarantee the integrity of installed firmware on offline IoT device during the offline firmware update. The firmware update mechanism proposed in this thesis is designed to perform on both online and offline IoT device through the use of skipchain technology. Detail information regarding the skipchain technology [16] will be explained in Chapter 2.

The update mechanism in the proposed scenario begins when the device manufacturer publishes the information of new firmware inside a smart contract to the blockchain network. Then, the newly created smart contract is verified by other nodes in the blockchain network. In this scenario, each gateway is connected to the blockchain network as a passive node, able to check all the verified smart contracts, and manages one or more IoT devices. In the case that the gateway detects the new firmware update, it will check if there is any connected device that needs this update. The IoT device that requires the firmware update will download the binary through a given url from the metadata given by its vendor.

The contribution of this thesis are as follow:

1. This thesis introduce a new design of skipchain-based firmware update framework for IoT devices, especially for device with limited Internet connection.
2. A secure firmware update distribution protocol is designed to support the proposed framework and firmware update mechanism.
3. The security strength of the proposed protocol design is evaluated against well-known attacks: man-in-the-middle (MITM) attacks, replay attacks, firmware modification attacks, impersonation attacks, and isolation attack.

The remainder of this thesis is organized as follow: introduction to skipchain and existing firmware update mechanisms are described in Chapter 2. The assumptions used in this thesis, overview of the skipchain technology, the proposed system architecture and protocol designs are explained in Chapter 3. Prototype implementation of the proposed firmware update

framework is discussed in Chapter 4. Chapter 5 describes the security and performance analyses conducted on the proposed protocol. Finally, this thesis is concluded in Chapter 6.

Chapter 2 Literature Review

2.1 Cyber Attack on Firmware and Remote Firmware Update for Embedded Device

In 2011, cyber attack on embedded battery controller firmware could possibly cause a security hazard such as overheated the battery and made it catch fire [10]. Embedded battery controller is usually used in Li-Ion and Lithium Polymer batteries. These types of battery are used in embedded devices, therefore the overheating battery can cause embedded devices be permanently broken.

In 2013, Cui et al. [9] demonstrated how firmware update process can be exploited. The vulnerability allow the attacker to inject malicious firmware modifications into the embedded device. Cui et al. successfully exploited the third-party libraries in HP printer firmware images and injected malicious modification into the library. Their experiment result proved the capabilities of the running-malware in printer, such as: network reconnaissance, data exfiltration, and propagation to general purpose computer or other embedded device types.

The other cyber attack that can be done in firmware update process is replay attack. During the firmware update process, attacker can duplicate the firmware binary from the vendor server and make the victim device check the unnecessary firmware update. The redundant checking process will drain the victim device battery. To resist replay attack, [?] use times-tamp and nonce mechanism. Embedded device does not need to check the duplicate firmware update within the small interval of time by checking the

timestamp of the request.

In the firmware update process, adversary can intersect the firmware binary from the vendor. Using reverse engineering algorithm, attacker can modify the firmware or inject malicious code and send it to the victim device. To keep the integrity and the authenticity of the firmware, firmware binary can be digitally signed by vendor using its private key. After downloading the firmware binary from the vendor's repository, embedded device can validate the digital signature using the attached public key. Once the firmware is validated, the update process is started.

There are currently two methods to update the device's firmware, manual and automatic process. Manual process is vulnerable and not efficient, because user needs to manually download the firmware binary and install it to embedded device. Downloaded firmware integrity and authenticity can not be confirmed by un-experienced end user nor by the embedded device because it is manually downloaded and installed. On the other hand, automatic update over the Internet needs secure channel to ensure the integrity and authenticity of the firmware. Firmware Over The Air (FOTA) is typically used for automatic firmware update, general FOTA process is described in Figure 2.1. This automatic update process (remote firmware update) will update the latest firmware binary to the targeted device without manual operation.

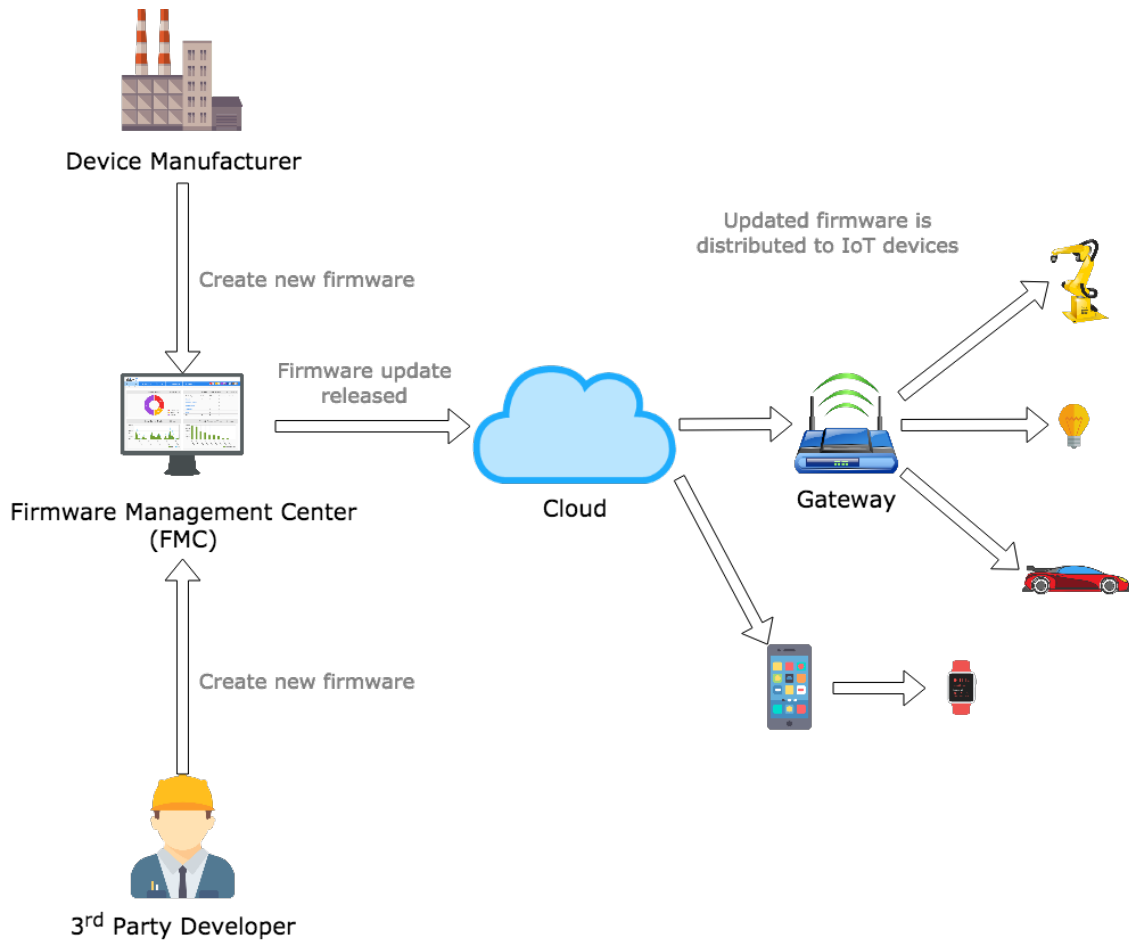


Figure 2.1: Firmware Over The Air (FOTA) process diagram

Remote firmware update uses two methods in its process, push and pull method. While push method is applicable for resource constrained device, pull method is more practical for more powerful with stable connection device [17]. Both methods use client-server architecture, in the push method, vendor repository as the server sends the firmware binary as chunks to the device as the client. After receiving the whole firmware binary, client will notify the server by sending an error or success message. If there is no error, client can do the firmware update. In pull method, vendor repository as a server will provide a url to download the firmware binary to the gateway as the client. Once the whole firmware binary is downloaded, the error or

success notification message will be sent back to the server. Then, client can do the firmware update if the download is successful.

2.2 Blockchain

Based on Satoshi Nakamoto [15], blockchain was originally created for peer-to-peer electronic cash payment transaction based on cryptographic proof instead of trust (to third party, such as Bank). Satoshi Nakamoto proposes a solution to prevent the double-spending problem in peer-to-peer network without a trusted third party. Each transaction data in the blockchain network is verified, time-stamped and linked with the previous data in the form of chained hash (proof-of-work). By doing so, blockchain technology can create a record that cannot be changed without redoing the hash-based proof-of-work.

Bitcoin's proof-of-work uses Adam Back's Hashcash [18] which is originally created to limit email spam and denial of service attack (DDoS). The block generation process in Bitcoin uses Hashcash as its proof-of-work. In order for a block to be accepted by network participants, miners must complete a proof-of-work which covers all of the data in the block. Miner is any node that lends their computational resource in order to secure the network and is rewarded with some bitcoin. The first miner who finds the nonce and successfully creates the new block will be rewarded.

Bitcoin implement the proof-of-work [1] by incremental nonce in the block until a value (nonce) is found, this nonce together with the transaction data must give block's hashed value (using SHA-256) the required zero bits. The example of how Bitcoin's proof-of-work iterates nonce is shown

in Figure 2.2. In this example, the required zero value is 3 and it must be in the beginning of the hash value. Total of 4251 hashes iteration is needed in this case, which is not a very hard computation (most computer can achieve at least 4 million hashes per second). That is why Bitcoin automatically varies the difficulty to keep a roughly constant rate of block generation. The difficulty of this proof-of-work is adjusted so as to limit the rate at which a new block can be generated in the network for every 10 minutes. Due to the very low probability of successful generation of a block, this makes it unpredictable which worker computer (miner) in the network will be able to generate the next block.

```
"Hello, world!0" => 1312af178c253f84028d480a6adc1e25e81caa44c749ec81976192e2ec934c64 = 2^252.253458683
"Hello, world!1" => e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be78e948a9332a7d8 = 2^255.868431117
"Hello, world!2" => ae37343a357a8297591625e7134cbea22f5928be8ca2a32aa475cf05fd4266b7 = 2^255.444730341
...
"Hello, world!4248" => 6e110d98b388e77e9c6f042ac6b497cec46660deef75a55ebc7cfd65cc0b965 = 2^254.782233115
"Hello, world!4249" => c004190b822f1669cac8dc37e761cb73652e7832fb814565702245cf26ebb9e6 = 2^255.585082774
"Hello, world!4250" => 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9 = 2^239.61238653
```

Figure 2.2: Iterating nonce that make hashes value begin with a number of zero value [1]

After the correct nonce and its corresponding hash value is found in the mining process, chaining algorithm is depicted in Figure 2.3. Each block's hash value contains the hash value of previous block. As all blocks in the ledger are chained with its previous block, any modification on the value of any block will affect the whole blocks structure in the network. This mechanism makes blockchain tamper-proofing, means that no one can modify nor tamper a transaction once it is validated and recorded into blockchain.

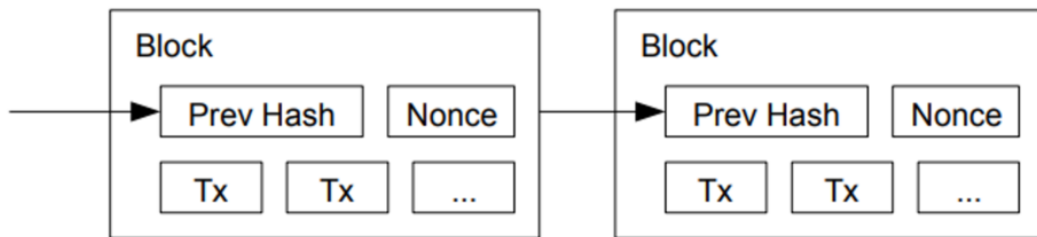


Figure 2.3: Nakamoto's chaining block diagram 2.3

2.3 Skiplist

Skiplist was first described in 1989 by William Pugh [19]. Skiplist is a data structure that allows fast search within an ordered sequence of elements [20], fast search is possible by skipping over a few element rather than searching linear number of steps. The structure of a skiplist is shown in Figure 2.4. As an example, the search process to find the fifth node, the skiplist firstly traverse a link of width 1 at the top level. Now four more steps are needed, but the next span on this level is too large (10), so it will drop one level to level 3. The algorithm once again traverse one link of width 3. Since the next step of width 2 is skipping too far (becomes 6), instead of traversing in the same level, it will drop down to the bottom level. Lastly the skiplist will traverse the final link of width 1 to reach the fifth link (target).

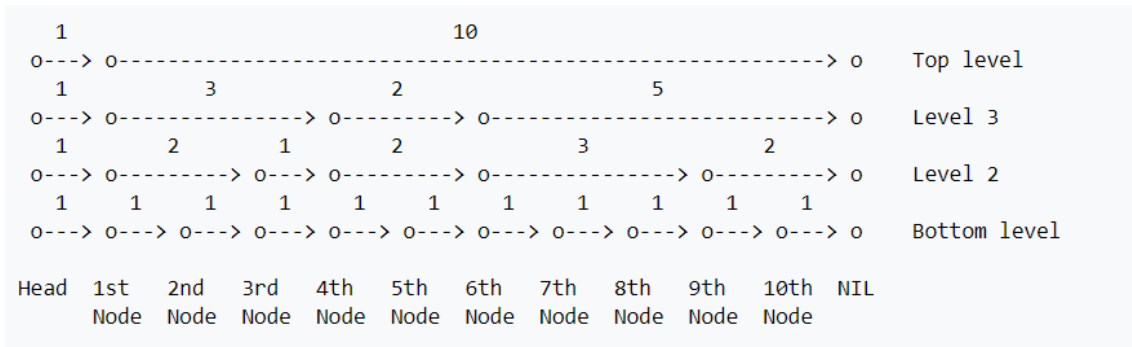


Figure 2.4: Skiplist structure with span between two same-level nodes 2.4

Based on how the insertion process, there are two types of skiplist. In deterministic skiplist, user needs to define how the insertion process needs to be done. For example J. Ian Munro [2] defined "1-2 skiplist", "1-2 skiplist" structure requires either one or two nodes of height $(h-1)$ exist between any two nodes of height h ($h > 1$). The deterministic skiplist structure is shown at Figure 2.5

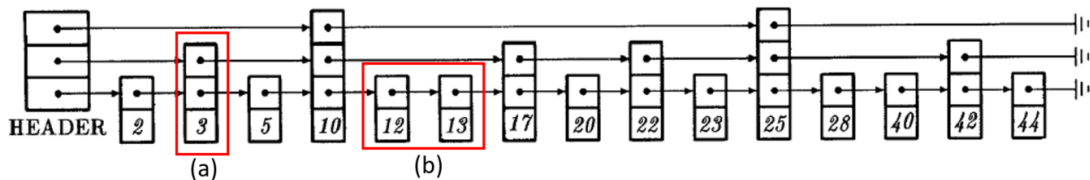


Figure 2.5: (a) There is a node with height of two between two nodes with height of three. (b) There are two nodes with height of one between two nodes of height of two [2]

On the other hand, a randomized skiplist defines the new node height based on probabilistic of a coin toss. It will add the height of the newly inserted node until the coin-toss lands on tail or the height reach the maximum-defined-height. The randomness makes the average insert and update cost for the skiplist is low. Randomized skiplist can achieve simple implementation but requires larger storage.

2.4 Skipchain

One of the key characteristic of blockchain is the traceable asset. It allows all blockchain's participants know where the asset come from and how it change overtime. However, to verify an asset will require user's device to be online, connected to the Internet and to pay bandwidth and power costs to maintain the connectivity with multiple nodes on blockchain network [16]. User can to join as a full node, maintaining a mirror copy of the entire blockchain to verify or trace an asset. As an option, user can join as a lightweight node and download partial amount of the entire blockchain data from connected full node. However, it is difficult for low power, limited storage and limited connection device to join the blockchain network even as lightweight node.

Joining a blockchain network as a lightweight node is not without risk, a compromised full node can isolate the connected lightweight node using a "fake" view of blockchain. The lightweight node can do a secure verification by synchronizing with multiple full node, but the compromised full node can achieve it goals by deceiving the isolated lightweight node during the "offline" state. The blockchain node remain vulnerable to isolation attack because the fundamental problem is that the current blockchain can never be validated in any absolute ways, but only relative to the perspective from another node.

To tackle this issue, Chainiac [3] introduced skipchain, a novel cryptographic blockchain structure that adapts the skiplist idea by adding long-distance links both forward and backward in time as illustrated in Figure 2.6. When Chainiac creates new block, it will create additional hash link

to farther backward in time. On the same time, Chainiac also creates long-distance forward link via collective signatures [4]. With long-distance forward and backward link, skipchain technology becomes cryptographically traversable in both directions.

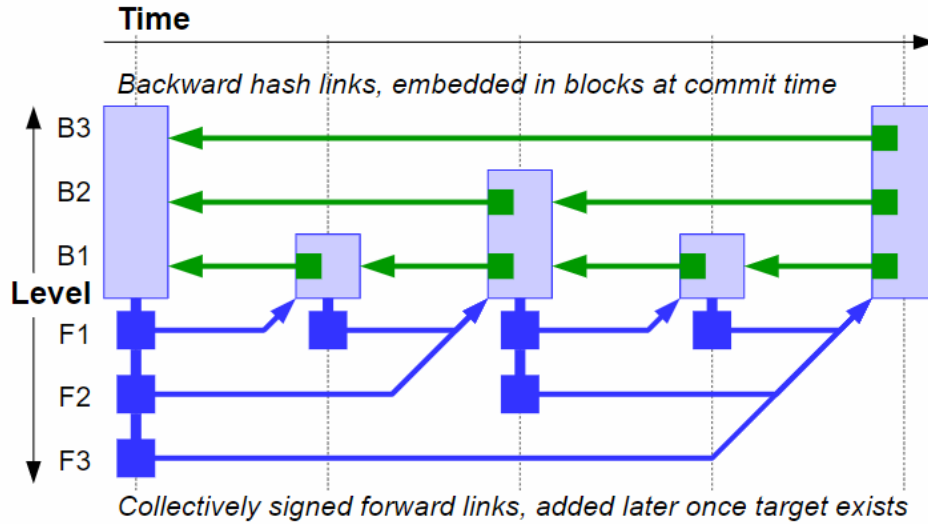


Figure 2.6: Skipchain structure [3]

Using this skipchain characteristic, node as a client can efficiently prove the correctness of a transaction anywhere in time with respect to other party reference point on the blockchain, in a logarithmic number of steps, regardless of which node has a more up-to-date data view of the blockchain [16]. For example, a gateway in a smart home environment wants to show a new firmware update to a connected IoT device. In this case, the IoT device has limitations to connect to the Internet and to synchronize with the other node. The IoT device can not store partial part of the blockchain data due to its limited storage. The gateway can simply sends to the IoT device a small number of collectively signed forward links through a peer-to-peer connection to prove securely that the new firmware

updates is indeed on the blockchain.

2.5 Blockchain-based Firmware Update Framework

In this section, previous researches on the firmware update mechanism for embedded devices based on blockchain infrastructure are introduced. Lee et al. [21] introduced the blockchain-based firmware verification and update schemes for embedded devices in IoT environment. Lee et al. mentioned that excessive network traffic may occur when large number of embedded devices downloading the latest firmware simultaneously from a dedicated firmware update server. This problem leads Lee et al. to utilize blockchain network concept, where a device can request a firmware update from a decentralized peer-to-peer network.

In [21] architecture, embedded device acted as full node in the blockchain network which was difficult to be implemented in real world scenario. Because the embedded devices with low-power, limited storage and limited connection to the Internet are difficult to maintain connection with the other node in blockchain network. It is also not possible for the embedded device to store growing blockchain data. Moreover, when there are many kind of embedded device with different requirement of firmware, it will take a long time to make the download request because this scheme used pull-method. In result, this scheme might not efficient for heterogeneous IoT ecosystem.

Boudguiga et al. [22] investigated that it was possible to use blockchain infrastructure to provide firmware update to several embedded devices belonging to different manufacturer. This scheme relied on trusted node in

the blockchain to validate an update innocuousness before its transmission to the end objects. The trusted node checks that the new update does not contain bugs and resist to known attack.

There are two architectures introduced by [22]. In the first architecture, each vendor was expected to provide at least one worker node in the blockchain network. An embedded device could periodically pull the blockchain by random picks the node and check the firmware version. The mentioned trusted node takes place in the second architecture as a refinement from the first architecture. The trusted node directly receives new firmware binary from the vendor, then notifies the corresponding embedded devices after the validation process completed. The corresponding device can download the valid firmware binary and do the firmware update.

Chapter 3 System Environment and Protocol Designs

This chapter describes the assumptions used in this thesis, an overview on Skipchain technology, the proposed system architecture design and the proposed protocol design. All assumptions used in the proposed firmware update framework is described in Section 3.1. An overview of Skipchain technology is explained in Section 3.2. The system architecture design and the proposed protocol design for the firmware update mechanism are explained in Section 3.3 and Section 3.4, respectively.

3.1 Assumptions

The assumptions used in this thesis are listed as follows:

1. The proposed firmware update mechanism uses push-update method, in which the vendor repository sends the metadata of a new firmware in the form of smart contract to the skipchain network. Once the corresponding smart contract is verified, all the nodes in the skipchain network will synchronize their smart contract data with the latest data.
2. In the proposed scenario, the gateway as a passive node in the skipchain network will obtain a notification on the newly released firmware from the verified smart contract. Afterward, the gateway will check if the newly released firmware is required by the IoT devices connected to it.
3. The scope of IoT devices in this thesis have low-power consumption,

small storage capacity, limited computation capability, and limited Internet connectivity. However, the IoT devices are able to perform simple Hash computation and simple value comparison.

4. The IoT devices have pre-installed the public key of vendor to authenticate the firmware signature.
5. Each IoT device have pre-installed a small portion of skipchain data to allow the associated gateway to join a specific skipchain network. In addition, the pre-installed skipchain data enable the IoT device to securely verify the more recent skipchain data from the associated gateway.
6. The firmware binary is distributed from the vendor repository to the targeted IoT device through a secure channel.
7. The vendor repository and the corresponding vendor node are communicating through a non-secure communication channel.
8. The peer-to-peer connection between gateway and IoT devices is not in secure channel.

3.2 Skipchain Overview

Skipchain is a cryptographically traversable, offline and p2p verifiable blockchain structure [16]. Skipchain implements the concept of skiplist into a blockchain structure by adding long-distance forward and backward links. The long-distance backward link has been introduced in Chainiac [3]. The long-distance backward link contains a hashed-address of the previously created block. This backward link allows users to trace a transaction

history starting from the latest block data. The transaction trace feature could be performed when the user's device contains more recent block data than the block data of searched transaction.

The unique forward link in skipchain structure enables user to verify transaction history even when the user's blockchain data is behind the more recent blockchain data. This forward link contains two pieces of information as follow:

1. A secure hash-pointer to the first block committed by the next consensus group.
2. A description of how the consensus group has changed, such as which cosigner joins or leaves the consensus group (indicated by the addition or reduction on the number of cosigner's public keys).

Once the successor of a block is committed, the consensus group responsible for the block creates and collectively signs a forward link. This process is similar to securely issues an information of the current consensus group's members for the prior, already-committed block. The security of forward link is assured by a collective signature.

Skipchain uses a tree-based collective signing (CoSi) [4] schemes, in which each signer (*cosigner*) joins the Schnorr signature [23] to agree on a valid statement. The group of cosigner, known as collective authority (cothority), will perform the consensus-like process where each cosigner will verify and sign a valid transaction. The CoSi architecture is illustrated in Figure 3.1. In the CoSi architecture, each record could be collectively signed by different cothority member.

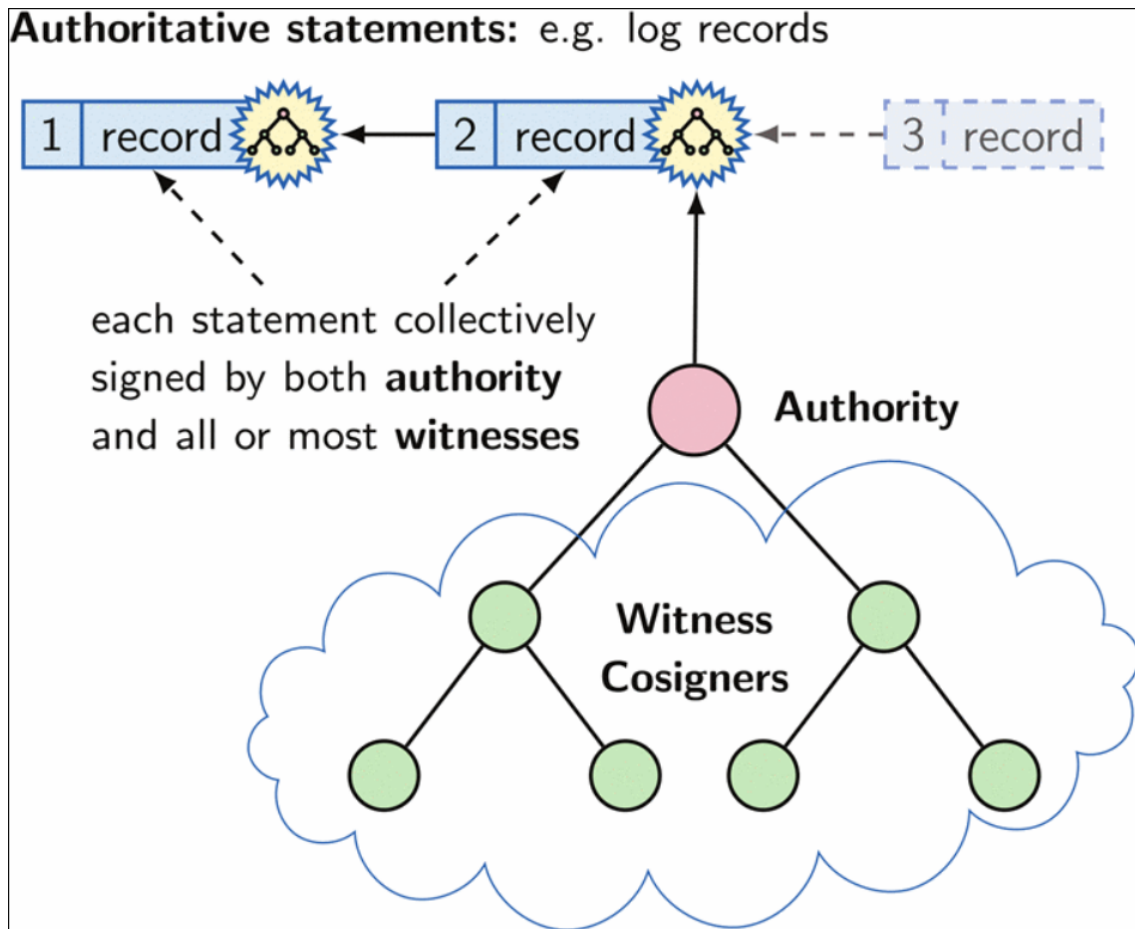


Figure 3.1: CoSi protocol architecture [4]

As the forward links are signed by different cothority members, the user could securely perform forward tracing on the more recent blockchain data without having to trust the other parties. In addition, the user could learn all the changes on the consensus group and would be able to know the correct set of public keys while performing the forward tracing. The set of public keys will be used to verify the corresponding collective signatures in the blockchain. Since all of the forward links are collectively signed by cothority members, an attacker can not create a fake blockchain branch unless the attacker compromises or colludes with large number of cothority members.

3.3 Architecture Design

In this thesis, we proposed the skipchain-based firmware update framework for IoT environments. The architecture of our framework is illustrated in Figure 3.2. There are three roles in the architecture design, described as follow:

1. Vendor: the manufacturer of the specific IoT device and acts as active node in the skipchain network. The active node in the skipchain network is actively do the verification process as a Cothority's member. Each time a vendor pushes a new version of firmware, the vendor repository needs to broadcast the vendor-signed firmware metadata to the Cothority nodes via the vendor's active node.
2. Gateway: the gateway for connected IoT devices and connected to the skipchain network as passive node. The passive node in skipchain network does not participate in any verification process, however, it only synchronizes the block data with the most recent block data in the skipchain network.
3. IoT device: the sensor-embedded device that connects to a specific gateway and does not need to fully trust the gateway device. Each IoT device can verify the notification of firmware update from the gateway by using its pre-installed skipchain data.

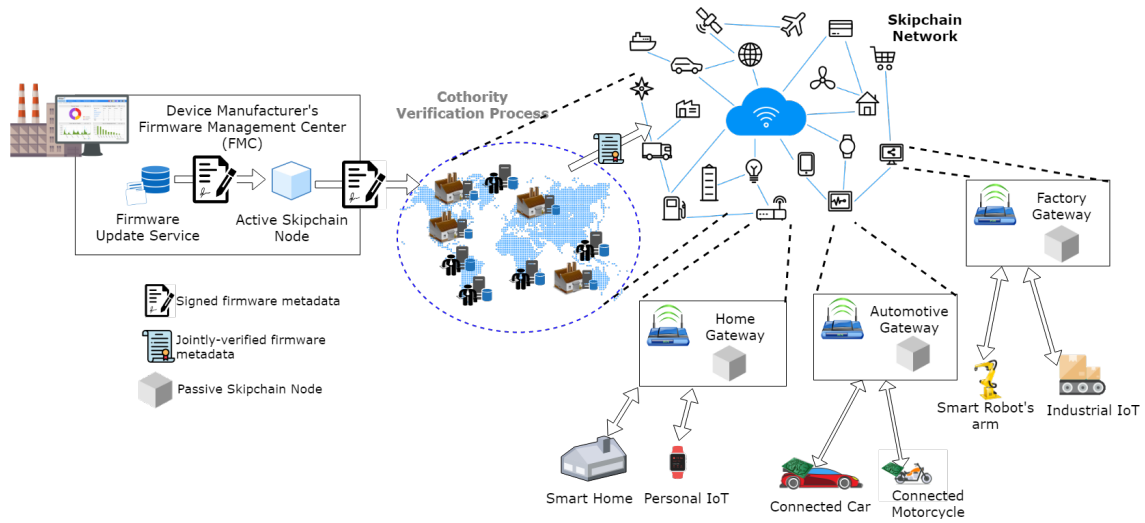


Figure 3.2: Firmware Over-the-Skipchain (FOTS) process diagram

From the Figure 3.2, there are two kinds of node in the skipchain network namely: active node and passive node. The active nodes are actively performing the verification process on any new smart contract data containing the metadata of a new version of firmware. In addition, the active node acts as Cothority's member in the proposed skipchain network. In the contrary to the active node, the passive node only continuously synchronizes the block data with the more recent block data in the skipchain network and does not involve in any verification process of a newly released smart contract. Each gateway in the proposed architecture design manages at least one IoT device in local network environment. For example, there are smart home appliances and personal IoT devices connected to a home router as gateway. The connection between the gateway and the IoT device is peer-to-peer connection, such as Bluetooth connection.

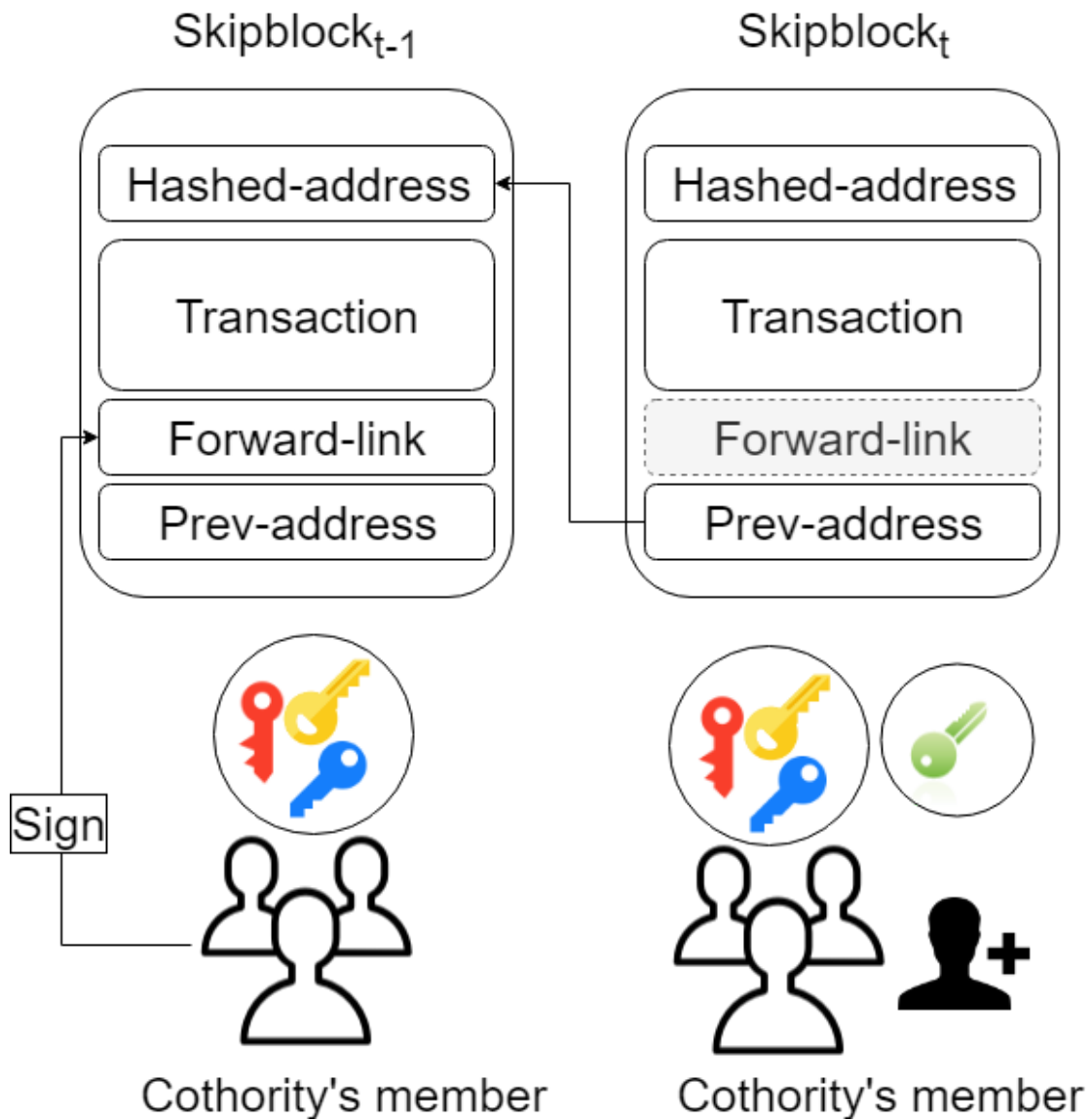


Figure 3.3: Skipblock structure diagram

The skipblock structure as shown in Figure 3.3, consists of four main contents: hashed-address, transaction, forward link, and prev-address. These four contents are described as follow:

1. Hashed-address: a hashed-value that is unique and will be the address for a skipblock. Each skipblock is accessible through this hashed-address.
2. Transaction: the transaction contains the metadata of a firmware, such

as the targeted device model, the firmware version, and the url to download the corresponding firmware binary in the vendor repository. A skipblock could contain one or more transaction data.

3. Forward Link: the unique structure from skipchain itself, it contains the next block's address and the difference of the current cothority's members that verify the $skipblock_t$. The cothority's members those verify the $skipblock_{t-1}$ (red, yellow and blue key) need to sign the forward link. In Figure 3.3, the forward link will contain the information about the new cothority's member with the green key. Later, client will check the forward link, and know which keys do the client need to use to verify the skipblock. And the dashed-line forward link in $skipblock_t$ means that it contain no information of the next block since t is the newest block.
4. Prev-address: the hashed-address of a previous skipblock.

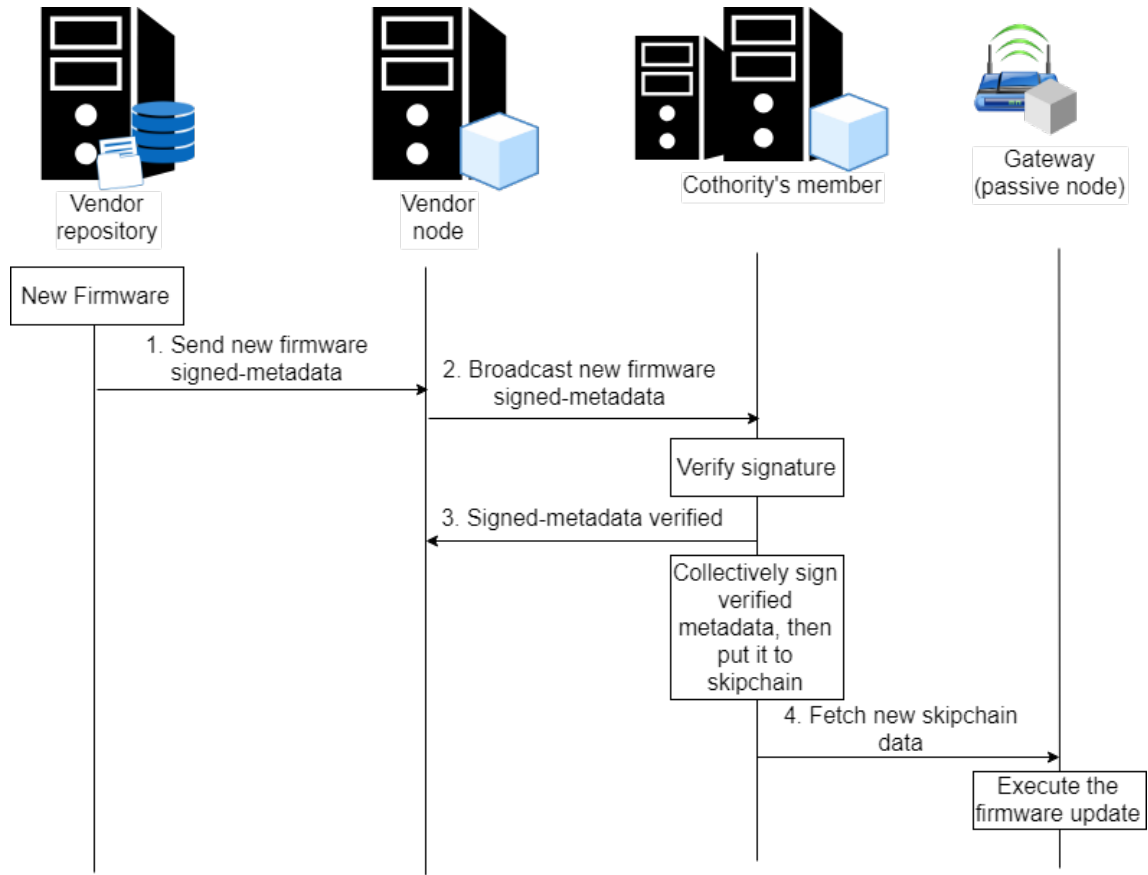


Figure 3.4: Firmware update verification process

In our proposed scheme, we divide the overall process into two parts. The first part is firmware verification process, in which this process will leverage how the metadata of a new version of firmware is verified and how to store the corresponding metadata into the skipchain. After the metadata of the corresponding firmware is sent to the skipchain network, the peer-to-peer verification process will begin. Afterward, the IoT device verifies the firmware update notification from the gateway (in a peer-to-peer connection) before the firmware update is performed by the IoT device.

The firmware verification process of the proposed framework is shown in Figure 3.4 and described as follow:

1. Each time a vendor releases a new version of firmware, the ven-

dor uses his private key to sign the metadata of the newly released firmware. The metadata of a firmware contains information such as the url to download the corresponding firmware, the version of a firmware, the targeted device model, and other detailed information which will be discussed in Section 3.4.

2. The vendor node broadcasts the signed-metadata to the other cothority member to be verified using the corresponding vendor's public key.
3. After the verification process for the firmware metadata, the cothority will collectively sign the valid metadata and store the signed firmware metadata into the skipblock in the skipchain network.
4. The gateway as a passive node will continuously synchronize with the skipchain network to fetch the new firmware update information and execute the firmware update mechanism.

The firmware update peer-to-peer verification process is illustrated in Figure 3.5. After obtaining the firmware update notification, the gateway checks the targeted device model for the firmware update from the newly created skipblock. Afterward, the gateway will notify the corresponding connected IoT devices. Then, the IoT device can verify if the updated firmware comes from the valid skipblock and whether the firmware is published by the correct vendor. After the updated firmware is verified, the IoT device sends its current firmware version to the gateway. Upon receiving the installed firmware version from the IoT device, the gateway will check whether the installed version of firmware is older than the new version of firmware recorded in the skipblock. In the case that the installed firmware

version in the IoT device is older than the firmware version in the skip-block, then the gateway will download the new version of firmware binary from the url provided in the metadata. The downloaded firmware binary is sent to the corresponding IoT device to be verified and installed to the device.

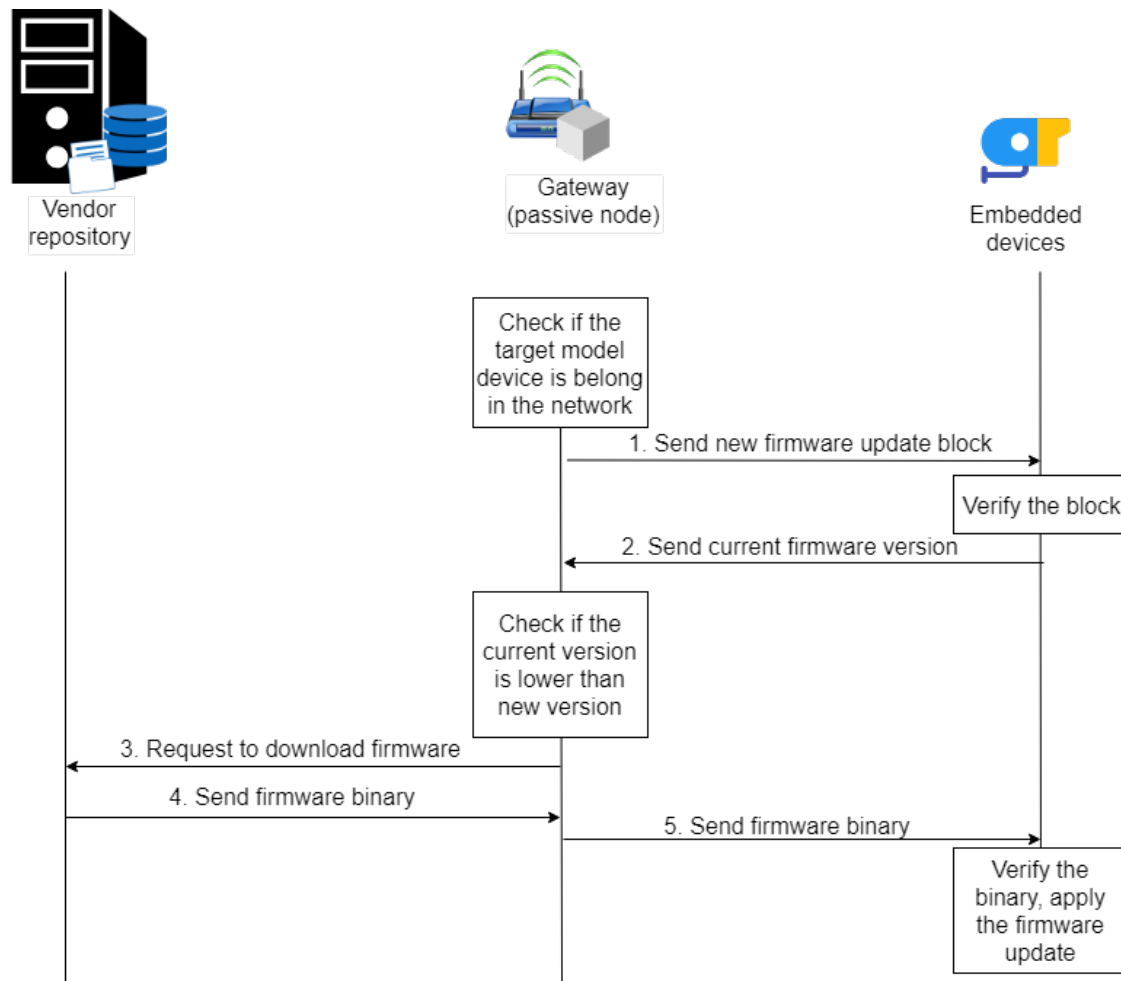


Figure 3.5: Firmware update peer-to-peer verification process

In the peer-to-peer verification process, the IoT device uses the skipchain's forward link to securely verify the new version of firmware without synchronizing and storing any additional data. Instead, the gateway as a more capable device with more storage capacity and more stable Internet connection can do the synchronization process. The peer-to-

peer verification also helpful when a gateway is not connected to the Internet because the offline-gateway can get the firmware update notification from another online-gateway via peer-to-peer notification. In such case, the offline-gateway is still able to verify the new firmware update notification and does not blindly trust the online-gateway. The peer-to-peer verification process is illustrated in Figure 3.6

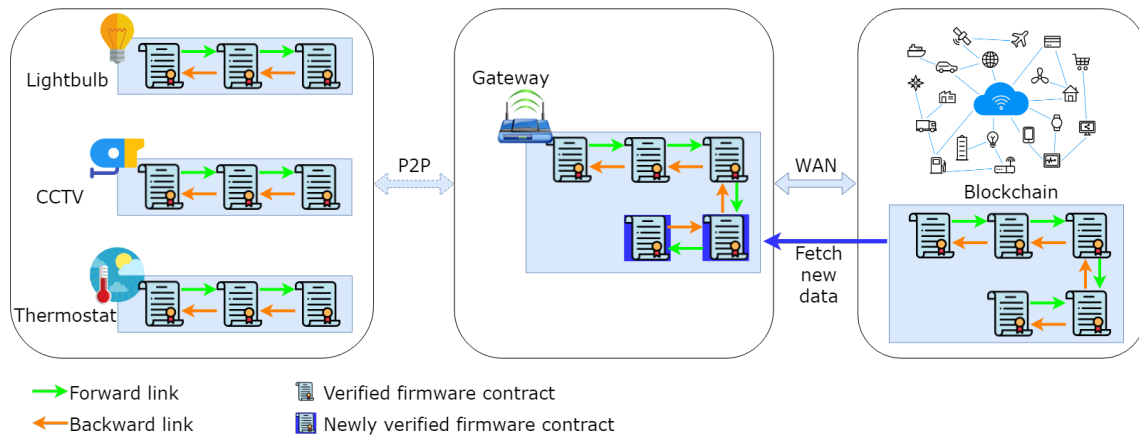


Figure 3.6: Peer-to-peer verification process

In the peer-to-peer verification process, the pre-installed skipblock in the IoT device takes an important role as it contains several valid skipblocks data. The pre-installed skipblock structure is illustrated in Figure 3.7. In addition, the pre-installed skipblocks contain list of public keys (illustrated by the red, yellow and blue keys in the Figure 3.7) to verify the given forward link via peer-to-peer connection.

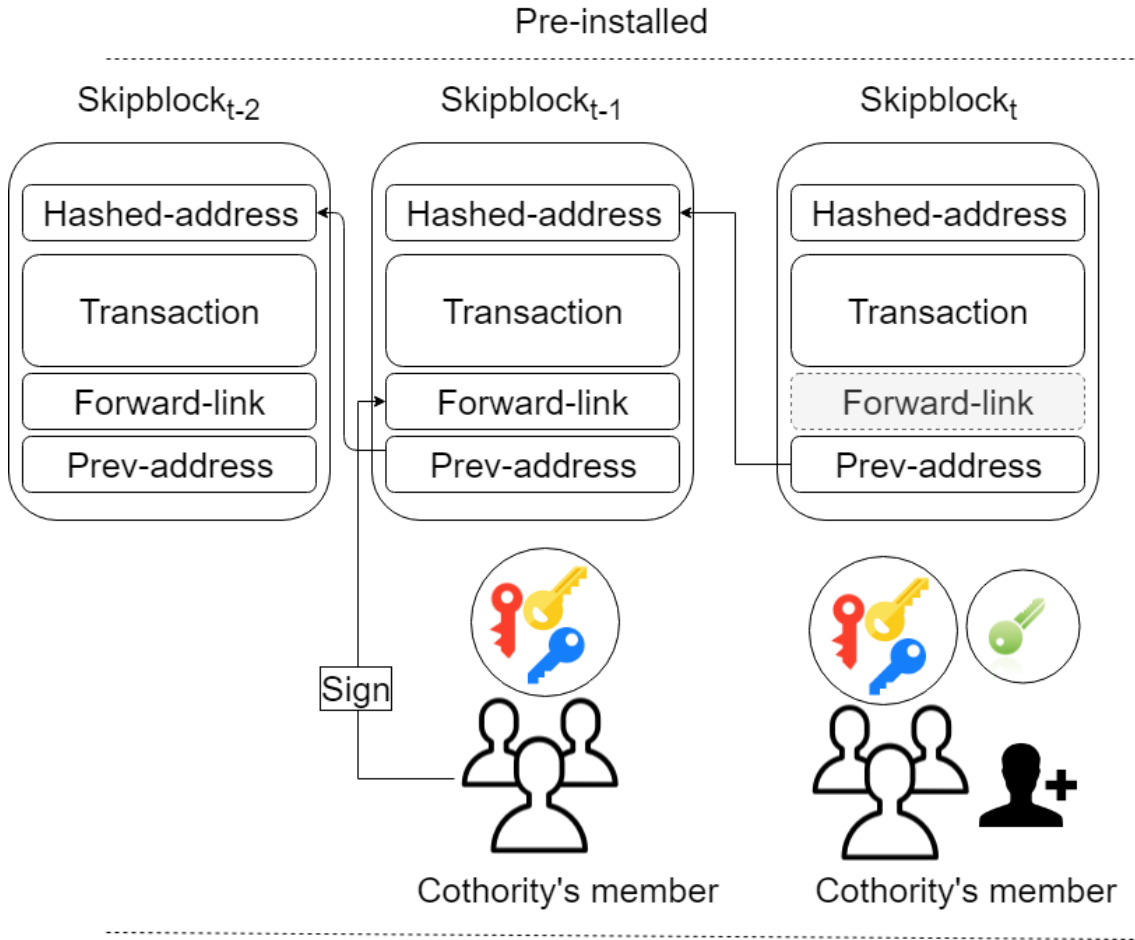


Figure 3.7: Pre-installed skipblock diagram

3.4 Protocol Design

In this section, our proposed firmware update protocol is introduced and explained. There are two stages of verification before the firmware binary is executed in the proposed protocol: firmware update verification process and peer-to-peer verification process between IoT device and gateway. Table 3.1 shows the notations that we use in our protocol.

Table 3.1: Notation used in the proposed protocol

Notation	Definition
ID_d	The unique ID of an IoT device d
ID_g	The unique ID of a gateway g
ID_v	The unique ID of a vendor v
ID_b	The unique ID of a skipblock b
ID_c^v	The unique ID of cosigner node c owned by a vendor v
k	The session key
sid	The unique ID of a specific session
p	Randomly generated shared prime number in key exchange process
g	Shared base number in key exchange process
s	Shared secret between key exchange process
a, b	Secret owns by Alice and Bob in key exchange process
URL_d^v	The url to download the corresponding firmware binary for the targeted IoT device d from a specific vendor v
CFV_d^v	The current firmware version installed in the IoT device d from vendor v
LFV_d^v	The latest firmware version for IoT device d from vendor v
LFB_d^v	The latest firmware binary for IoT device d from vendor v
M_d^v	The device model for targeted IoT device d from vendor v
$BLOCK_s$	The pre-installed skipchain s block from the vendor
$BLOCK_t$	The skipchain block in newest timestamp t
$KDF()$	The key derivation function. The inputs of this function are passphrase and salt and the output of this function is a session key k
$E_k()$	Symmetric key encryption using session key k
$D_k()$	Symmetric key decryption using session key k
$H()$	One-way hash-chain function
$ $	String concatenation operation

3.4.1 Firmware Update Verification Protocol

This verification protocol is initiated each time a vendor (v) has new updated firmware to be published in the skipchain network (URL_s). This protocol is running under the assumption that the connection between the vendor repository and vendor node (as cothority's member) is not secure. Cothority will do the consensus-like verification protocol before a new block ($BLOCK_t$) is created. Vendor node will broadcast the new firmware update metadata to each cothority member, then each member will verify the signed-metadata. Vendor node will collect the signature of cothority member, then put the metadata into the new block ($BLOCK_t$). Figure 3.8 shows the firmware update verification protocol.

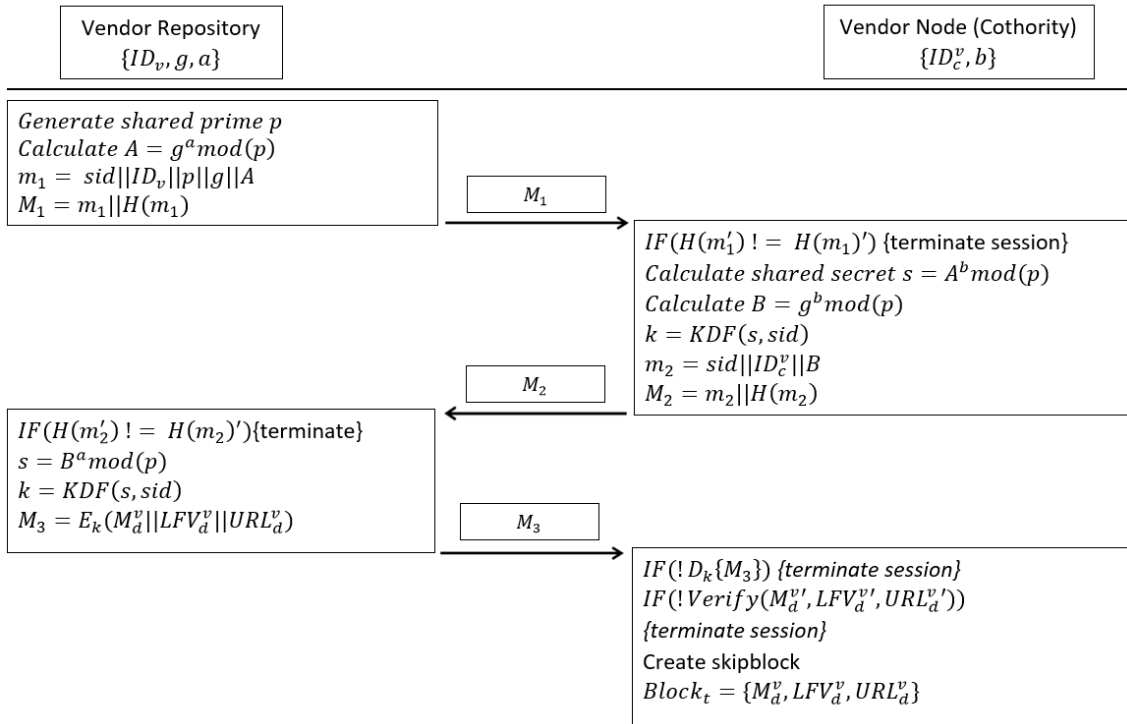


Figure 3.8: Firmware update verification protocol

Step 1: Vendor Repository \rightarrow Vendor Node

Vendor repository v will generate a shared prime p , and calculate a

shared number $A = g^a \text{mod}(p)$. This shared number later will be used by vendor node to generate same shared secret. Vendor repository then create a message $m_1 = (sid || ID_v || p || g || A)$, then concatenate the message m_1 with the hash value of the message $H(m_1)$ and send it to the vendor node.

Step 2: Vendor Node \rightarrow Vendor Repository

After receiving message m'_1 from vendor repository, vendor node will validate the message by applying the same hash function $H()$ to the message m'_1 . If the hashed-value do not match $H(m_1)'$, vendor node will terminate the session. Once validated ($H(m'_1) == H(m_1)'$), vendor node will obtain session id sid' , vendor repository's ID ID'_v , shared prime p' , shared base g' , and shared number A' . Next, vendor node calculate shared secret $s = A'^b \text{mod}(p')$ and shared number $B = g'^b \text{mod}(p')$. Vendor node uses key derivation function $KDF()$, passing shared secret s and session id sid as salt to create new session key k . Vendor node then create a message $m_2 = (sid || ID'_c || B)$, then concatenate the message m_2 with the hash value of the message $H(m_2)$ and send it to the vendor repository.

Step 3: Vendor Repository \rightarrow Vendor Node

Once receive m'_2 , vendor repository will do the validation toward the message m'_2 and hashed-value $H(m_2)'$. If the message is valid ($H(m'_2) == H(m_2)'$), vendor repository will obtain session id sid' , vendor node's id ID'_c and shared number B' then calculate shared secret $s = B'^a \text{mod}(p)$. It will also use KDF to create same session key k . Last, vendor repository encrypt message $m_3 = E_k(M_d^v || L F V_d^v || U R L_d^v)$ using symmetric encryption protocol with session key k , then send the message m_3 back to vendor node.

Step 4: Vendor Node \rightarrow Vendor Repository

Vendor node will decrypt the encrypted message m'_3 using its session key k , and obtain the new firmware update metadata. Vendor node will broadcast the metadata to cothority member to be verified. Once the verifying process is done, vendor node will collect the signature from participated cothority member. The collected signature proof that the metadata has already verified, and ready to be put in the new skipchain block $BLOCK_t$.

3.4.2 Firmware Update Peer-to-Peer Verification Protocol

Peer-to-peer verification protocol occurs between gateway g and its belonging IoT device d . This process is needed everytime gateway fetch new block data from the skipchain network (URL_s). Each IoT device can verify if the firmware update notification from the gateway comes from a valid block from valid a skipchain network. The peer-to-peer verification protocol is shown in Figure 3.9.

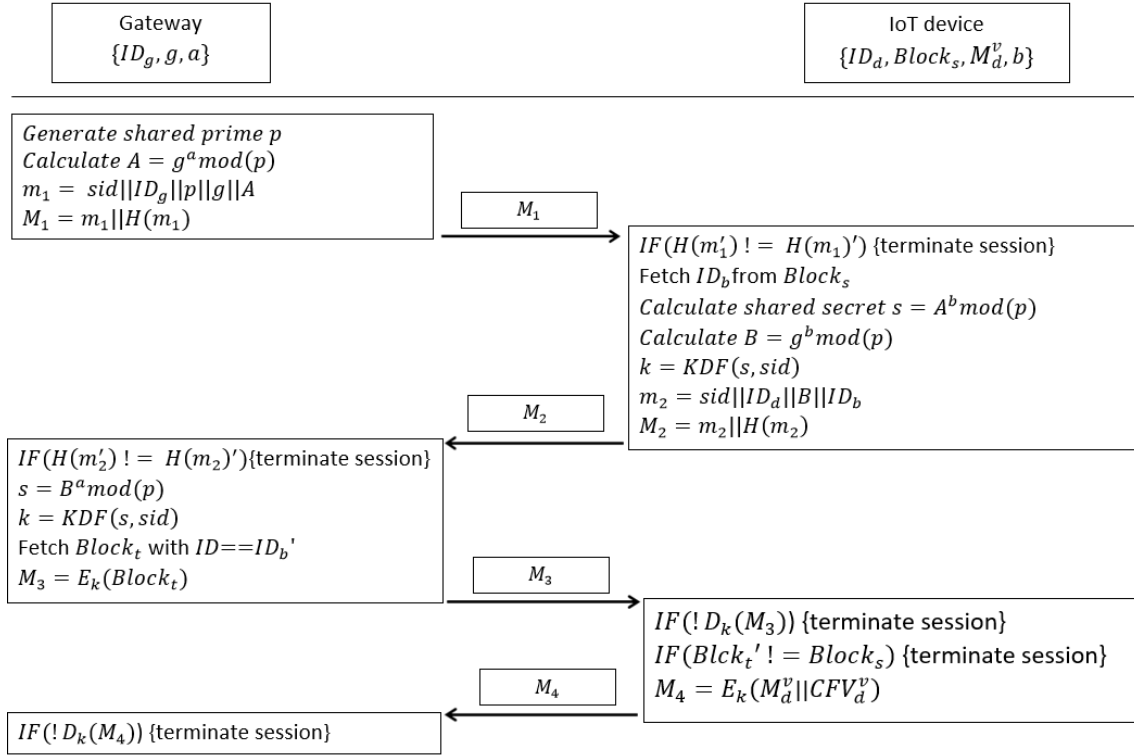


Figure 3.9: Firmware update peer-to-peer verification protocol

Step 1: Gateway \rightarrow IoT Device

Gateway will generate a shared prime p , and calculate a shared number $A = g^a \text{mod}(p)$. Gateway then create a message $m_1 = (sid || ID_v || p || g || A)$, then concatenate the message m_1 with the hash value of the message $H(m_1)$ and send it to the IoT device.

Step 2: IoT Device \rightarrow Gateway

Upon received message m'_1 from gateway, IoT device will validate the message by applying the same hash function $H()$ to the message m'_1 . If the hashed-value do not match $H(m_1)'$, vendor node will terminate the session. Once validated ($H(m'_1) == H(m_1)'$), vendor node will obtain session id sid' , gateway's id ID'_g , shared prime p' , shared base g' , and shared number A' . Next, IoT device calculate shared secret $s = A'^b \text{mod}(p')$ and shared

number $B = g^b \text{mod}(p')$. IoT device uses key derivation function $KDF()$, passing shared secret s and session id sid as salt to create new session key k . IoT device then create a message $m_2 = (sid || ID_d || B || ID_b)$, then concatenate the message m_2 with the hash value of the message $H(m_2)$ and send it to the gateway.

Step 3: Gateway \rightarrow IoT Device

Once receive m'_2 , gateway will do the validation toward the message m'_2 and hashed-value $H(m_2)'$. If the message is valid ($H(m'_2) == H(m_2)'$), vendor repository will obtain session id sid' , IoT device's id ID_d^v , skipblock id ID_b' and shared number B' then calculate shared secret $s = B'^a \text{mod}(p)$. It will also use KDF to create same session key k . Gateway fetches the $BLOCK_t$ with id equals to ID_b' . Last, gateway encrypt message $m_3 = E_k(BLOCK_t)$ using symmetric encryption protocol with session key k , then send the message m_3 back to IoT device.

Step 4: IoT Device \rightarrow Gateway

IoT device will decrypt the encrypted message m'_3 using its session key k , and obtain the new firmware update block. IoT device will verify the obtained $BLOCK'_t$ if it is matched with its pre-installed block $BLOCK_s$. Once verified, IoT device encrypt its model information and current firmware version it has $m_4 = E_k(M_d^v || CFV_d^v)$ using session key k . Last, IoT device will send the message m_4 to the gateway.

Step 5:

Gateway will decrypt the message m'_4 and obtain model information $M_d^{v'}$ and current firmware version $CFV_d^{v'}$ of the IoT device d . These information will be used for the next process, the firmware execution protocol in Section 3.4.3.

3.4.3 Firmware Update Execution Protocol

The firmware update execution will be processed once the peer-to-peer verification process is done. This process uses some variables collected from the peer-to-peer verification process, like session key k , device model information $M_d^{v'}$ and device current firmware version $CFV_d^{v'}$. With our assumption in Section 3.1, the IoT device will verify the firmware binary using its pre-installed public key of its vendor. The firmware update execution protocol is shown in Figure 3.5.

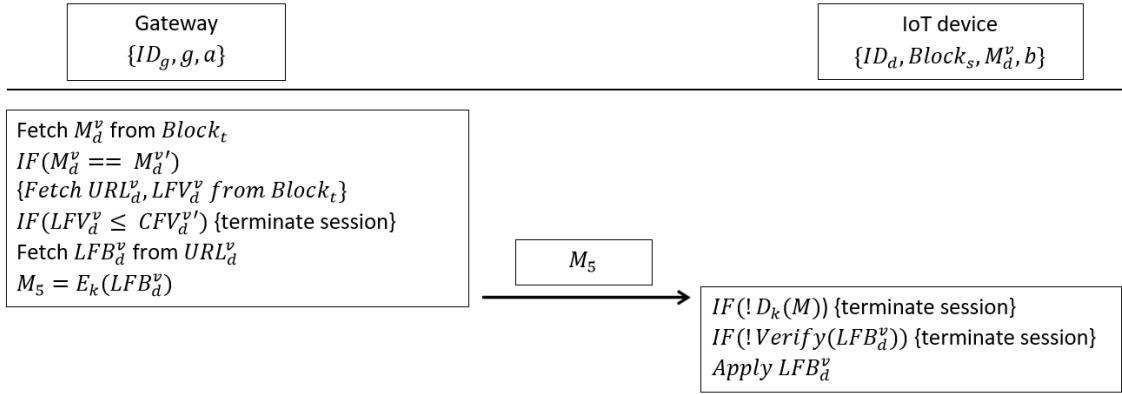


Figure 3.10: Firmware update execution protocol

Step 1: Gateway \rightarrow IoT Device

Gateway will fetch the target device model M_d^v from the newly created block $BLOCK_t$. It will check if the targeted model device is same as the IoT device model $M_d^{v'}$ belonging to it. If there is targeted device model within its belonging IoT device ($M_d^v == M_d^{v'}$), it will fetch the download url URL_d^v and the latest firmware version LFB_d^v from the block $BLOCK_t$. Gateway will check if the latest firmware version is greater than the device current firmware version ($LFB_d^v \geq CFV_d^{v'}$). If the update is required, gateway will fetch the firmware binary LFB_d^v from the down-

load url URL_d^v . Last, gateway encrypt message $m_5 = E_k(LFB_d^v)$ using session key k , then send the message m_5 to IoT device.

Step 2: IoT Device \rightarrow Gateway

IoT device will decrypt the encrypted message m_5^l using its session key k , and obtain the new firmware update binary. IoT device will verify the obtained $LFB_d^{v'}$ using its pre-installed vendor's public key. Once verified, IoT device will execute the firmware binary $LFB_d^{v'}$ and do the update process.

Chapter 4 Prototype Design and Implementation

This chapter describes prototype design of our proposed skipchain-based firmware update framework in Section 4.1. The implementation of the proposed framework and the result of the experiments described in Section 4.2.

4.1 Prototype Design

This section describe two main procedures used in the prototype implementation. First, the key exchange procedure is described in Subsection 4.1.1. Second, encryption and decryption procedures are described in Subsection 4.1.2 and Subsection 4.1.3 respectively.

4.1.1 Key Exchange Procedure

We have implemented two type of prototypes with two different involved parties. The first prototype design (called the repository-node prototype) simulates protocol between vendor repository as a client and vendor node as a server, this prototype uses the Internet as the information exchange connection protocol. Second prototype (called the gateway-IoT device prototype) implements client-server architecture between gateway and IoT device. The second prototype uses Bluetooth connection to exchange the information. The gateway takes the role of Bluetooth server and the IoT device takes role of Bluetooth client.

Both prototypes use Diffie-Hellman [24] key exchange

protocol. The three stages of key exchange protocol are described in Algorithm 1,2,3. In initiation procedure from Algorithm 1, **vendor repository** (in repository-node prototype) uses *WIFI_CLIENT_SOCKET* and **IoT device** (in gateway-device prototype) uses *BLUETOOTH_CLIENT_SOCKET*. The client sets some pre-defined variables used in key exchange procedure. After the pre-setting is complete, the client concatenates the variables as one string message. The message will be concatenated with its hashed-value and send via *CLIENT_SOCKET* to the server.

Algorithm 1 Client Key Exchange Initiation

```

1: procedure KEY_EXCHANGE_PROCEDURE_CLIENT
2:   SERVER_ADDRESS : 140.118.109.81
3:   INIT CLIENT_SOCKET(HOST : 140.118.109.81, PORT : 8080)
4:   SET IDv : "vendorA_repo_1"
5:   SET a : 6  $\leftarrow$  secret_number
6:   SET g : 5  $\leftarrow$  shared_base
7:   SET p : RANDOM()  $\leftarrow$  random shared_prime
8:   SET sid : TODAY()
9:   CALCULATE  $A = g^a \text{mod}(p)$ 
10:   $m = (sid || ID_v || g || p || A) \leftarrow \text{message}$ 
11:  CLIENT_SOCKET.SEND( $m || \text{SHA256}(m)$ )

```

The response for Algorithm 1 is described in Algorithm 2. In the response procedure, **vendor node** (in repository-node prototype) uses *WIFI_SERVER_SOCKET* and **gateway** (in gateway-device prototype) uses *BLUETOOTH_SERVER_SOCKET*. *SERVER_SOCKET* to receive the message from client. The server fetch the concatenated message by using split function.

After applying the split function to the message, the server gets an array value. If the array length does not match the required length in the procedure (6), the server will terminate the session. Otherwise, server put each array component into different variables. The server will check if the hashed-value in the last array component matches with the hashed-value of newly created variables.

Once the message is verified, the server calculates the shared secret s and uses PBKDF2 [25]. PBKDF2 function is a key derivation function that applies hash-based message authentication code (HMAC) to the input passphrase (s) along with a salt value (sid). The derivative function repeats many times to produce a derived key, the key can be used as a cryptographic key in subsequent operations. Cryptographic key produced from KDF will be the session key for AES encryption-decryption key in Algorithm 4,5.

The last step in the server-side is sending the required data B , that is used for the *SHARED_SECRET* creation in the client-side. Server concatenates the message and its hashed-value, then send it to the client. Key exchange procedure in server-side is ended here, and the server has acquired the shared secret s .

Algorithm 2 Server Key Exchange Response

```
1: procedure KEY_EXCHANGE_PROCEDURE_SERVER
2:   INIT SERVER_SOCKET(HOST : 127.0.0.1, PORT : 8080)
3:   SET IDc : "id_vendorA_cothority_1"
4:   SET b : 15  $\leftarrow$  SECRET_NUMBER
5:   SERVER_SOCKET.LISTEN(1)
6:   DATA = SERVER_SOCKET.RECEIVE(1)
7:   if (DATA.LENGTH() > 0) then
8:     ARRAY_DATA = DATA.SPLIT()
9:     if (ARRAY_DATA.LENGTH()  $\neq$  6) then
10:      TERMINATE_SESSION()
11:      SET sid' = ARRAY_DATA[0]
12:      SET ID'v = ARRAY_DATA[1]
13:      SET g' = ARRAY_DATA[2]  $\leftarrow$  shared base
14:      SET p' = ARRAY_DATA[3]  $\leftarrow$  shared prime
15:      SET A' = ARRAY_DATA[4]  $\leftarrow$  shared number
16:      SET hash_value' = ARRAY_DATA[5]
17:      if (SHA256(sid'||ID'v||g'||p'||A') == hash_value') then
18:        s = A'(b) mod(p')  $\leftarrow$  shared secret
19:        k = PBKDF2(s, sid')  $\leftarrow$  session key
20:        B = g'(b) mod(p')  $\leftarrow$  shared number
21:        m = (sid'||IDc||B)  $\leftarrow$  message
22:        SERVER_SOCKET.SEND(m||SHA256.H(m))
23:      else
24:        TERMINATE_SESSION()
```

The last procedure in the client-side is described in Algorithm 3. In the first step, the client splits the response message from the server. If the array length matched the required length (4), the client put each array component into different variables then runs verification process to check if the hashed-value in the last array component matches with the hashed-value of newly

created variables. If the message is verified, the client calculates the shared secret uses the shared number B . In the last step, the client uses PBKDF2 to produce same session key as the server owns.

Algorithm 3 Client Key Exchange End

```

1: procedure KEY_EXCHANGE_PROCEDURE_CLIENT
2:   SERVER_ADDRESS : 140.118.109.81
3:   INIT CLIENT_SOCKET(HOST : 140.118.109.81, PORT : 8080)
4:   DATA = CLIENT_SOCKET.RECEIVE(1)
5:   if (DATA.LENGTH() > 0) then
6:     ARRAY_DATA = DATA.SPLIT()
7:     if (ARRAY_DATA.LENGTH() != 4) then
8:       TERMINATE_SESSION()
9:     SET sid' = ARRAY_DATA[0]
10:    SET ID'c = ARRAY_DATA[1]
11:    SET B' = ARRAY_DATA[2]  $\leftarrow$  shared number
12:    SET hash_value' = ARRAY_DATA[3]
13:    if (SHA256(sid'||ID'c||B') == hash_value') then
14:       $s = B'^{(a)} \bmod(p') \leftarrow$  shared secret
15:       $k = \text{PBKDF2}(s, \text{sid}') \leftarrow$  session key
16:    else
17:      TERMINATE_SESSION()

```

4.1.2 AES Encryption Function

After both the client and the server have the same session key, they can use the session key as AES symmetric key. Client and server use the AES encryption architecture to send sensitive information. AES encryption function described in Algorithm 4. The encryptor need to prepare a symmetric key k to create an AES object. The encryptor

uses session id sid as salt and generate nonce value IV . The function encrypts the plain text using the nonce IV , then concates the hex-value of the result with the hex-value of the salt and the nonce ($HEX(SALT)||HEX(IV)||HEX(AES_RESULT)$)

Algorithm 4 AES Encryption Function

```

1: procedure AES_ENCRYPTION
2:   GET  $sid', k \leftarrow$  defined in algorithm 3
3:   INIT  $AES\_ = AESGCM(k)$ 
4:   INIT  $IV = RANDOM() \leftarrow$  nonce value
5:   INIT  $SALT = sid'$ 
6:   INIT  $PLAINTEXT = "this is message"$ 
7:   #None associated_data need to be authenticated
8:   SET  $AES\_RESULT = AES\_ENC(IV, PLAINTEXT, None)$ 
9:   SET  $CIPHER = HEX(SALT)||HEX(IV)||HEX(AES\_RESULT)$ 
10:  RETURN CIPHER

```

4.1.3 AES Decryption Function

By using the same symmetric key k , the other party can decrypt the cipher text that created using Algorithm 4. The AES decryption function is described in Algorithm 5. In the first step, the decryptor creates same AES object as encryptor using session key k . The decryptor splits the received cipher text, then decode the cipher text using $UNHEX()$ function. Afterward, the decryptor removes the salt from the decoded cipher text and uses the decryption function from the AES object to decrypt the decoded cipher text. The decryption function $AES_DEC()$ requires the nonce value (IV) and the decoded cipher text and return the plain text as result.

Algorithm 5 AES Decryption Function

```
1: procedure AES_DECRYPTION
2:   GET  $k \leftarrow$  defined in algorithm 3
3:   GET  $CIPHER' \leftarrow$  defined in algorithm 4
4:   INIT  $AES\_ = AESGCM(k)$ 
5:   SET  $SALT', IV', CIPHER\_TEXT' = UNHEX(CIPHER'.SPLIT())$ 
6:   INIT  $PLAINTEXT = AES\_ .DEC(IV, CIPHER\_TEXT, None)$ 
7:   RETURN  $PLAINTEXT$ 
```

4.2 Prototype Implementation

For our prototype implementation, we use the following devices:

Table 4.1: The device specification for each role

Roles	Device
IoT Device	Raspberry Pi 3 with the following specification: <ul style="list-style-type: none">- CPU: ARM Cortex 1,4GHz- RAM: 1GB SRAM- Bluetooth version: 4.0
Vendor Repository	MSI GL62 laptop with the following specification: <ul style="list-style-type: none">- CPU: Intel i7 2,6GHz- RAM: 16GB- SSD: 512GB- Bluetooth version: 4.0
Vendor Node and Gateway	Asus PC with the following specification: <ul style="list-style-type: none">- CPU: Intel i5 2,3GHz- RAM: 12GB- Bluetooth version: 4.0

We use Python as programming language and run the python script in

each device. The skipchain node within the network is created using the github code from the Chainiac paper [3]. As the open source code for the skipchain is still under hard development, we could not find a more suitable API by the time of this thesis writing. As a result, we use python library (**subprocess**) to access the skipchain via terminal.

We conduct two experiments on the implemented prototype as follow:

1. First, we perform experiment between vendor repository and vendor node using the Internet connection. The average running time after running the first experiment 10 times is one second. The process of putting the data to the skipchain takes 90% of the running time.
2. Second, we conduct experiment between gateway and IoT device using Bluetooth connection. Average running time after running the second experiment 10 times is one second. It takes 50 microseconds to send 10 megabytes of firmware data over the Bluetooth connection.

Chapter 5 Security and Performance Analyses

In this chapter, security analysis of the proposed firmware update framework and protocol are provided and discussed. There is security analysis against attack within information exchange over insecure connection described in Section 5.1. In the Section 5.2 discussed how we use the analysis tools and the result. Performance analysis also conducted in Section 5.3 to compare the performance of the proposed protocol against existing firmware update protocol. In last section, Section 5.4 summary the security analysis result from the previous sections.

Table 5.1: Architecture design comparison of four blockchain-based firmware update framework

Architecture design	Proposed framework	Yohan et al. [26]	Lee et al. [21]	Boudguiga et al. [21]
Firmware update method	Push-method	Push-method	Pull-method	Pull-method
Vendor platform	Multiple vendors	Multiple vendors	Single vendor	Multiple vendors
IoT device platform	Heterogeneous devices	Heterogeneous devices	Specific device	Heterogeneous devices
Peer-to-peer firmware file sharing	Not allowed	Not allowed	Allowed	Allowed
Blockchain platform	Skipchain	Ethereum	Bitcoin	Bitcoin
Smart Contract	Available	Available	Not available	Not available
Trusted node	Not-required	Not-required	Not-required	Required
Peer-to-peer verification	Yes	No	No	No

Our proposed framework uses push-method to distribute the updated firmware, where a vendor can publish an update and notify the targetted device within the blockchain network without specific request from the device. The other two blockchain-based firmware achitectures [21,22] applied pull-method, where an IoT device needs to send a firmware update request to check for the new available firmware version. Push-method can

reduce attack-window time by updating the firmware immediately after the vendor publishes a new firmware update. Push-method is also more energy and network efficient than pull-method, because in pull-method, power-constrained and connection-limited IoT devices need to constantly request the availability of the new firmware version whether it exists or not.

Our framework architecture adopts skipchain technology, which enables smart contract creation. Skipchain as a permissioned blockchain platform protocol adapts PBFT concept in the consensus mechanism. In our proposed framework, PBFT is a better choice because the framework does not allow anyone to join the blockchain network but only for registered vendor and gateway. PBFT tackles the energy required by proof-of-work consensus mechanism with the drawback of no anonymity within the permissioned blockchain network.

Our proposed framework is able to provide peer-to-peer skipblock verification process by using the skipchain technology's forward link. It means that low-power with little or no Internet connectivity device does not need to maintain connections with many other network nodes. Low-power device can get the up-to-date block data from the stronger device (e.g. gateway) via peer-to-peer connection, and securely verify if the new block data exists in the valid skipchain network without storing the whole blockchain data.

The comparison of the security mechanism between proposed framework and the other related works is shown in Table 5.2. Our proposed framework is designed to protect the firmware update process from major cyber-attacks, respectively: firmware modification attack, impersonation attack, man-in-the-middle attack, replay attack and isolation attack. On the

Table 5.2: The comparison of security mechanism against cyber attack of four blockchain-based firmware update framework

Provide protection against	Proposed framework	Yohan et al. [26]	Lee et al. [21]	Boudguiga et al. [21]
Firmware modification attack	Yes	Yes	No	Yes
Impersonation attack	Yes	Yes	No	Yes
Man-in-the-middle attack	Yes	Yes	Yes	Not specified
Replay attack	Yes	Yes	Yes	Not specified
Isolation attack	Yes	No	No	No

other hand, the firmware update mechanism proposed by Lee et al. [21] could not prevent firmware modification and impersonation attack. Furthermore, our proposed framework provides additional security measure against isolation attack.

By nature, an offline blockchain node can not resist isolation attack, because there is no cryptographic means for any node to distinguish the real blockchain from a fake one. If an attacker can isolate (even temporarily) a node from the network, attacker can tricks the isolated node to accept the fake firmware update from newer block that never exist.

5.1 Informal Security Analysis

The following theorems are applied on both firmware verification protocol and peer-to-peer verification protocol. **Vendor repository** roles in firmware verification protocol and **IoT devices** roles in peer-to-peer verification protocol act as *client* in the theorems. **Vendor node** roles in firmware verification protocol and **gateway** roles in peer-to-peer verification protocol act as *server* in the theorems. The theorems are under assumption that the cryptographic hash function used in the proposed framework is assumed to be able to withstand all known types of cryptanalytic

attacks. This means the hash function has the capability of collision resistance, counteracts preimage attacks and second-preimage attacks.

In addition, the proposed protocol assumed to use a Cryptographically Secure Pseudorandom Number Generator (CSPRNG). CSPRNG must satisfied all statistical test and need to be unpredictable, means attacker unable to predict the next output of random value. Based on these assumption and annotations, the security analysis of the proposed protocol is defined as follow:

Theorem 1. *The proposed protocol supports mutual authentication and data integrity*

Proof. Because of the computational difficulty to collect shared secret from Diffie-Hellman key exchange protocol, only legitimate client and server can calculate the secret. Client uses the shared public key B from the server and its private key a , calculate $s = B^a \text{mod}(p)$. On the other hand, server uses the shared public key A from client and its private key b , to calculate the same shared secret $s = A^b \text{mod}(p)$. Client and server then use the shared secret s and session id sid to create session key $k = KDF(s, sid)$. Adversary who collects public material from the key exchange protocol will not be able to produce the same shared secret s without the private key (a, b) .

Furthermore, legitimate client and server can verify the hashed-value message from each other using predefined hash-chain function $H()$. Adversary who does not know the hash-chain function, will not be able to re-create the same hashed-value message and send it to the client or server. For example, if client send a message $M_1 = m_1 || H(m_1)$ to the server.

Server can use the same hash-chain function $H()$ and authenticate if the message comes from client, vice-versa $H(m'_1) == H(m_1)'$. Based on these two proofs, it can be concluded that the proposed protocol supports mutual authentication and data integrity.

Theorem 2. *The proposed protocol supports session key security*

Proof. In proposed protocol, both client and server always use newly generated shared prime p in key exchange protocol and session id sid as salt in session key k generation. The adversary needs to solve the Diffie-Hellman problem to get the current shared secret s and use the same key derivation function $KDF()$ as a client and server uses to get the current session key k . Because it is difficult to solve computational problem of Diffie-Hellman, and also hard to find accurate key derivation function $KDF()$, the proposed protocol provides session key security. This also proves that the protocol is able to defense against forward secrecy attack.

Theorem 3. *The proposed protocol defends against impersonation attack*

Proof. There are three targets for the adversary to impersonates to, namely: vendor repository, vendor node, and gateway. First, adversary can try to impersonate to a vendor repository and send invalid firmware update metadata to the vendor node. In order to deceive the vendor node, it need to send the message and its hashed-value to complete the authentication protocol. It is difficult to find the exact hash-chain function $H()$ using brute force, because after failing a session, server or client will catch the sender ip or mac address and blacklist the address if it fail up-to certain threshold. Additionally, adversary needs to get the vendor repository private key to sign the metadata. Because later, the metadata will be verified by the cothority

member before put into the skipchain.

Second, adversary can try to impersonate to a vendor node and deceive the cothority's member to sign malicious contract and put into skipchain network. Adversary need to join the permission blockchain platform as cothority's member. Because the cothority's member is predefined before the skipchain creation and the member's change need to be verified by the threshold of other member. It is presumably hard if adversary does not control more than one third of the cothority member.

Third, adversary can try to impersonate to a gateway and deceive IoT device that there is new fake firmware update. The forward link in skipchain enables IoT device to verify the new block that contain new firmware update in the peer-to-peer connection. In order to create fake block, adversary need to have a threshold of cothority member's private key and sign the fake block with it. This attack is not plausible, especially if the cothority member is changing overtime. Hence, it could be concluded that the proposed protocol defends against impersonation attack.

Theorem 4. *The proposed protocol defends against replay attack*

Proof. Assuming that adversary tries to perform replay attack on the proposed protocol, both firmware verification and peer-to-peer verification protocol. Adversary can send the collected valid-message M_1 from the previous session and replays the transaction repeatedly to cause an unnecessary transactions. However, our proposed protocol uses timestamped session id sid , so when an obsolete transaction is detected, it will be rejected and the session will be terminated. If an address failing session up-to cer-

tain threshold, it will be blacklisted from the server or client.

In addition, skipchain mechanism does not allow duplicate timestamped transaction. The duplicate transaction will be dropped during the validation process by cothority's member. Therefore, it is concluded that replay attack does not affect our proposed framework

Theorem 5. *The proposed protocol defends against man-in-the-middle attack*

Proof. Based on the proof given for Theorem 1, proposed protocol provides mutual authentication between legitimate client and legitimate server. In addition, adversary can not use the session key k to decrypt and get the important metadata and the firmware binary during the process. Thus, proposed protocol keeps the secrecy of the important data. Hence, the proposed protocol defends against man-in-the-middle attack.

Theorem 6. *The proposed protocol defends against firmware modification attack*

Proof. Based on the proof given for Theorem 1 and Theorem 5, proposed protocol provides mutual authentication and defends against man-in-the-middle attack. Furthermore, the firmware update contract contains firmware location URL_d^v . If an adversary aim to modify the firmware binary, adversary needs to aim non secure channel during the firmware binary transmission. And the only non secure channel is peer-to-peer connection between gateway and IoT device, since the gateway downloads the firmware binary from vendor repository using TLS secure channel. If somehow, adversary can modify the firmware binary and forward it, IoT

device can detect the firmware binary is invalid. IoT device with pre-installed vendor's public key can verify the signed-firmware-binary.

5.2 Protocol Verification Using Scyther Tool

Scyther is a tool used for security protocol verification, where it is assumed that all the cryptographic functions are perfect. Scyther tool can find security problem that arise from the way the protocol is constructed. The language used to write proposed protocol structure in Scyther is SPDL. Our proposed protocol then validated using 'automatic claim' and 'verification claim' procedures in the Scyther tool. The result of each claim will be explained in the following subsection.


Scyther results : verify							
Claim				Status		Commer	
protocolv4	Server	protocolv4,Server1	Secret data3	ok	Verified	No attacks.	
		protocolv4,Server2	Secret data4	ok	Verified	No attacks.	
		protocolv4,Server3	Alive	ok	Verified	No attacks.	
		protocolv4,Server4	Niagree	ok	Verified	No attacks.	
		protocolv4,Server5	Nisynch	ok	Verified	No attacks.	
	Client	protocolv4,Client1	Secret data3	ok	Verified	No attacks.	
		protocolv4,Client2	Secret data4	ok	Verified	No attacks.	
		protocolv4,Client3	Alive	ok	Verified	No attacks.	
		protocolv4,Client4	Niagree	ok	Verified	No attacks.	
		protocolv4,Client5	Nisynch	ok	Verified	No attacks.	
Done.							

Figure 5.1: Scyther tool verification result

5.2.1 Data Secrecy

In the proposed protocol, sensitive information (denoted as *data* in Figure 5.1) secrecy is achieved by encrypting each data using symmetric key k algorithm which is shared between client (C) and server (S). In firmware verification protocol, vendor repository acts as a client and vendor node acts as a server. In peer-to-peer verification protocol, IoT device acts as a client and gateway acts as a server. The successful claim for the data secrecy is shown in Figure 5.1. The claim used in the SPDL code is:

```
#server role
claim(Server, Secret, data3) #server send data3
claim(Server, Secret, data4) #server receive data4
#client role
claim(Client, Secret, data3) #client receive data3
claim(Client, Secret, data4) #client send data4
```

5.2.2 Aliveness

In the Figure 5.1, proposed protocol claims to ensure the server and client liveness during the information transmission. The claim used in the SPDL code is:

```
#server role
claim(Server, Alive)
#client role
claim(Client, Alive)
```

5.2.3 Non-injective Agreement and Non-injective Synchronisation

In the Figure 5.1, proposed protocol claims to ensure non-injective agreement and non-injective synchronisation during the information transmission. Based on [27], non-injective agreement means all variables sent as part of the message m are received as expected. Therefore, both parties will agree over the values of the variables that are sent. Non-injective synchronisation means the send event from the first run $send_1$ followed by corresponding read $recv_1$ and so on until the send-read event in the end of the protocol. The claim used in the SPDL code is:

```
#server role
claim(Server, Niagree)
claim(Server, Nisynch)
#client role
claim(Client, Niagree)
claim(Client, Nisynch)
```

5.3 Performance Analysis

In this section, comparison on performance between the proposed firmware update protocol and existing firmware update protocol are discussed. Table 5.3 shows the comparison of time consumption to do firmware verification protocol and peer-to-peer verification protocol between the proposed protocol and three existing firmware update protocol [21,22,26].

In order to compare the computation cost between the proposed firmware update protocol and the existing protocol, the following assump-

tions are applied:

1. The size for session key k in our proposed protocol is 256 bits
2. The symmetric encryption and decryption operation uses AES GCM with 256 bits key
3. The one way hash function uses SHA256
4. The key derivation function uses PBKDF2 with 1000 iterations in our proposed protocol, the output is session key k

Table 5.3: The performance comparison of four blockchain-based firmware update framework

	Firmware verification	Peer-to-peer verification
Proposed framework	$3T_H + 2T_{KDF} + T_{symm_enc} + T_{symm_dec}$	$4T_H + 2T_{KDF} + 3T_{symm_enc} + 3T_{symm_dec}$
Yohan et al. [26]	$6T_{sig}$	$2T_{sig} + 7T_H$
Lee et al. [21]	$3T_{symm_enc} + 3T_{symm_dec} + 3T_{KDF} + 10T_H + 4T_{sig}$	Not available
Boudguiga et al. [22]	Not available	$4T_{sig} + T_{asymm_enc} + T_{asymm_dec}$

Table 5.4: Notations used for time consumption on differing computing operations

Notation	Definition
T_H	The time required to perform one way hash function
T_{KDF}	The time required to perform key derivation function
T_{sig}	The time required to perform digital signature operation
T_{symm_enc} and T_{symm_dec}	The time required to perform encryption (Enc) and decryption (Dec) operations in symmetric cryptosystem
T_{asymm_enc} and T_{asymm_dec}	The time required to perform encryption (Enc) and decryption (Dec) operations in PKI cryptosystem

The execution time for the hash function (SHA256) and symmetric cryptographic operations (AES GCM) are calculated based on Crypto library benchmark using C++ [28]. The benchmark compiled with Microsoft Visual C++ 2005 SP1, ran on Intel Core 2 1.83 GHz processor under Windows Vista in 32-bit mode. The execution time for the digital signature scheme and asymmetric operation (ECC 192-bits) are calculated based Yeh

et al. [29] and Tanwar et al. [30], respectively. The execution time for the cryptographic operations are shown in Table 5.5.

Table 5.5: The execution time of several cryptographic operations

Cryptographic operation	Execution time
SHA256 operation on 3200 bits of data	28.8ms
AES GCM symmetric encryption or decryption on 4000 bits of plain text	39.21ms
PBKDF2 with 1000 times iteration	0.999ms
Digital signature process (ECC 192-bits)	11.52s
ECC 192-bits asymmetric encryption or decryption	1.064s

Based on Tabel 5.3, the proposed protocol requires 166.818ms to finish firmware verification process ($3T_H + 2T_{KDF} + T_{symm_enc} + T_{symm_dec}$). During the peer-to-peer verification process, the execution time for the proposed protocol is 352.458ms ($4T_H + 2T_{KDF} + 3T_{symm_enc} + 3T_{symm_dec}$). In total, the execution time for the proposed protocol is 519.276ms.

On the contrary, the execution time of the firmware verification proposed by Lee et al. [21] requires 46.606 seconds ($3T_{symm_enc} + 3T_{symm_dec} + 3T_{KDF} + 10T_H + 4T_{sig}$). Our proposed protocol has better performance regarding the execution time in the firmware verification process. The protocol proposed by [21] requires more time to execute the digital signature process, including the digital signature verification process.

The execution time for Yohan et al. [26] protocol of the firmware verification process is 69.12 seconds ($6T_{sig}$). For peer-to-peer verification, [26] need 23.2416 seconds, the protocol run time is 92.3616 seconds in total. Our proposed protocol also has better performance than [26] regarding execution time, [26] protocol required more time to execute the digital signature process.

The execution time for Boudguiga et al. [22] protocol of the peer-to-peer verification process is 48.208 seconds ($4T_{sig} + T_{asymm_enc} + T_{asymm_dec}$). Our proposed protocol has better performance than [22] regarding execution time, because [22] used digital signature and asymmetric cryptographic mechanism to secure their protocol.

5.4 Discussion

Our proposed framework leverages on the concept of skipchain to securely update the firmware in IoT environment. First, our proposed framework guarantee the secure sensitive information sharing during the process. Our proposed framework uses symmetric key algorithm to protect the sensitive information, the symmetric key algorithm uses session key produced from the shared secret from key sharing protocol. The symmetric key algorithm protects the sensitive data more computationally efficient than asymmetric key algorithm, which is more suitable for IoT environment. By using symmetric key algorithm it will also ensure the data integrity and privacy during the transmission process. Once the transaction is validated and recorded into the skipchain, this transaction can not be altered or deleted. Hence, our proposed framework could ensure end-to-end firmware integrity.

Second, our proposed framework uses skipchain, a permissioned blockchain platform, which only allows authorized vendor and gateway to join the skipchain network. Each time a vendor want to push a new firmware update, vendor need to sign the metadata with its private key. Later, the cothority member will verify the signature before put the meta-

data into the skipchain. Adversary can not impersonate a legitimate vendor and push a malicious firmware metadata into the skipchain, unless it has vendor's private key.

In addition, using of the long-distance backward link used in skipchain, vendor can rollback to previous firmware version efficiently if a problem occur in the new firmware (e.g. malfunction). Vendor can trace back previous stable firmware version, re-invoke the existing contract in the skipchain. It will notify the gateway to rollback based on the information in the re-invoked contract.

During the prototype implementation process, the open-source code in Github for skipchain is still under heavy development. Therefore, our prototype implementation could not use any API that calls the skipchain services. In the result, the prototype implementation uses command line interface to interact with the skipchain services. This could lead to performance issue, since we do not use direct API to call the skipchain service.

Chapter 6 Conclusions

Firmware update is an essential process for vendor to manage its manufactured embedded device. Vendor can add new functionality, enhance security or re-configure the device through the new firmware update. Nowadays, automatic firmware update process is more commonly used, but the automatic process over the Internet is not without risk. Thus, a robust and lightweight protocol is needed to ensure the firmware security within the IoT environment. The proposed skipchain-based firmware update framework can enhance the end-to-end security of the firmware during the update process.

We investigate that using skipchain, a permission blockchain platform, can remove the traditional centralized architecture. Skipchain's forward link enables efficient peer-to-peer contract verification. This feature is important for offline embedded devices to verify the given contract without requires it to maintain connection with multiple nodes or storing any blockchain data. Thus, our contribution is provide perfect feature for low-power, connection restricted embedded device to verify the given firmware update contract.

Moreover, our proposed framework uses push method to keep the embedded device up-to-date as soon as the new firmware update release, which can shorten the vulnerable time. Our proposed framework is also proven to be secure and could withstand against firmware modification attack, impersonation attack, replay attack, man-in-the-middle attack, and isolation attack.

The improvement for our implementation can be made in the future

works. By using the API to directly call the skipchain service, it is expected to shorten the protocol running time. It is also challenging to implement the peer-to-peer verification API, to verify a given contract.

References

- [1] S. Nakamoto, “Proof of work.” https://en.bitcoin.it/wiki/Proof_of_work.
- [2] J. I. Munro, T. Papadakis, and R. Sedgewick, “Deterministic skip lists,” in *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’92, (Philadelphia, PA, USA), pp. 367–375, Society for Industrial and Applied Mathematics, 1992.
- [3] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, J. Cappos, and B. Ford, “Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds.” Cryptology ePrint Archive, Report 2017/648, 2017. <https://eprint.iacr.org/2017/648>.
- [4] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, “Keeping authorities ”honest or bust” with decentralized witness cosigning,” in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 526–545, May 2016.
- [5] W. H. Organization, “Deafness and hearing loss,” 03 2019.
- [6] L.-M. D. Sandler Wendy, *Sign Language and Linguistic Universals*. 2006.
- [7] V. Athitsos, C. Neidle, S. Sclaroff, J. Nash, A. Stefan, , and A. Thangali, “The american sign language lexicon video dataset,” in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, June 2008.
- [8] Gartner, “Gartner Says 8.4 Billion Connected ”Things” Will Be in Use in 2017, Up 31 Percent From 2016.” <https://www.gartner.com/newsroom/id/3598917>, 2014. [Online; accessed 30-November-2018].
- [9] A. Cui, M. Costello, and S. J. Stolfo, “When firmware modifications attack: A case study of embedded exploitation.,” in *NDSS* [9].
- [10] C. Miller and A. Labs, “Battery firmware hacking.” https://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf/, 2011.
- [11] K. Zetter, “How the nsa’s firmware hacking works and why it’s so unsettling.” <https://www.wired.com/2015/02/nsa-firmware-hacking//>, 2015.
- [12] R. Hassan, K. Markantonakis, and R. N. Akram, “Can you call the software in your device be firmware?,” *IEEE 13th International Conference on e-Business Engineering (ICEBE)*, 2016.
- [13] B.-C. Choi, S.-H. Lee, J.-C. Na, and J.-H. Lee, “Secure firmware validation and update for consumer devices in home networking,” pp. 39–44, *IEEE Transactions on Consumer Electronics*, 2016.
- [14] P. Point, “Proofpoint uncovers internet of things (iot) cyberattack.” <https://www.proofpoint.com/us/proofpoint-uncovers-internet-things-iot-cyberattack>, 2014.

- [15] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system." <https://bitcoin.org/bitcoin.pdf>.
- [16] B. Ford, "How do you know it's on the blockchain? with a skipchain." <https://bford.github.io/2017/08/01/skipchain/>.
- [17] K. Doddapaneni, R. Lakkundi, S. Rao, S. G. Kulkarni, and B. Bhat, "Secure fota object for iot," in *2017 IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops)*, pp. 154–159, Oct 2017.
- [18] A. Back, "Hashcash - a denial of service counter-measure." <http://www.hashcash.org/papers/hashcash.pdf>, 2002.
- [19] William Pugh, "Concurrent maintenance of skip lists." <https://drum.lib.umd.edu/handle/1903/542>, 1989.
- [20] Wikipedia, "Skip list — Wikipedia, the free encyclopedia." [Online; accessed 2-December-2018].
- [21] B. Lee, S. Malik, S. Wi, and J.-H. Lee, "Firmware verification of embedded devices based on a blockchain," in *Quality, Reliability, Security and Robustness in Heterogeneous Networks* (J.-H. Lee and S. Pack, eds.), (Cham), pp. 52–61, Springer International Publishing, 2017.
- [22] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey, "Towards better availability and accountability for iot updates by means of a blockchain," in *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, April 2017.
- [23] C. P. Schnorr, "Efficient identification and signatures for smart cards," in *Advances in Cryptology — CRYPTO' 89 Proceedings* (G. Brassard, ed.), (New York, NY), pp. 239–252, Springer New York, 1990.
- [24] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, pp. 644–654, Sept. 2006.
- [25] Wikipedia, "Pbkdf2." [Online; accessed 20-December-2018].
- [26] A. Yohan, N.-W. Lo, and S. Achawapong, "Blockchain-based firmware update framework for internet-of-things environment," in *Proceedings of the 2018 International Conference on Information and Knowledge Engineering, IKE'18*, pp. 151–155, CSREA Press, 2018.
- [27] J. F. Cremers, C and Mauw, Sjouke and Vink, Erik, "Denying authentication in a trace model," 07 2004.
- [28] W. Dai, "Cryptopp 5.6.0 benchmarks." <https://www.cryptopp.com/benchmarks.html>, 2009.
- [29] K.-H. Yeh, C. Su, K.-K. R. Choo, and W. Chiu, "A novel certificateless signature scheme for smart objects in the internet-of-things," in *MDPI*, (Philadelphia, PA, USA), pp. 367–375, Society for Industrial and Applied Mathematics, 1992.
- [30] G. S. Tanwar, G. Singh, and V. Gaur, "Secured encryption - concept and challenge," *International Journal of Computer Application*, vol. 2, May 2010.