國立台灣科技大學
資 訊 管 理 系

碩 士 學 位 論 文

Skipchain-based Secure Firmware Update Framework for Offline IOT Device

研 究 生：Liem Peter Santoso
學　　號：M10515809

指導教授：Prof. Nai-Wei Lo

中 華 民 國 一 百 零 八 年 一 月 十 六 日

Abstract

Estimated twenty billions of IOT devices have already installed today. With the huge amount of installed IOT devices, it is hard to do the maintenance process. One of the most important maintenance process is the device's firmware update. Firmware update is required to fix the device's bug, and the process in-between is needed to be secure. But in reality, not all of the IOT devices capable to connect to the Internet. In this thesis, we propose a secure skipchain-based firmware update framework for offline IOT device without trusting the gateway.

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 (賴孟泫, 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

List of Figures

List of Tables

List of Pseudocodes

Chapter 1 Introduction

According to [5], Internet of Things (IoT) has become the main standard of low-power lossy network having constrained resource. Internet of Things (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 installed IoT devices will exceed 20 billions by 2020 [6]. With rapid growth of installed embedded devices in past few years, there are many security challenges in IoT environment [5]. One of an important security issue in IoT environment is the security of software/firmware of the embedded devices.

Firmware is a certain set of data and commands to perform a specific task [7]. Embedded device as hardware need this set of instructions to perform their task. The instruction can be define as set of programmed routine to handle different component in the embedded device. Since, embedded device rely on its firmware to do its functionality, the firmware needs to be updated regularly in order to ensure the device works properly and to prevent the embedded device from any vulnerabilities by reducing the attack window time. It is also possible for vendor (device manufacturer) to add a new functionality to the device and to re-configure the device through the new firmware. Generally, embedded devices have several characteristics as follow: has low-power consumption, has small memory, and has limited computing capability. In addition, most of embedded 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 [8]. 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 [9]. These smart appliances are connected to home router to connect to the Internet, so once the home router is compromised then all connected devices those are belonging to the router will easily compromised as well. This issue proofed that rely on home gateway is not a good option for the embedded devices.

Blockchain [10] 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 tracable.

Using currently-deployed blockchains, users could trace all the transactions history recorded in the blockchain by synchronizing the block data to the latest version. This ability requires the user 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 Peer-to-Peer (P2P) firmware update protocol for offline IoT device. Skipchain technology is used to securely distribute the updated firmware to offline IoT devices. Skipchain is a cryptographically traversable, offline and p2p verifiable blockchain structure [11]. Detail information regarding skipchain technology will be explained in Chapter 2.

Offline sensor embedded devices are the focus of the proposed firmware update mechanism in this thesis. The term offline in the proposed scheme means that the devices do not have any mean to connect to the Internet except through a *gateway*. The proposed firmware update mechanism uses PUSH method to deliver the updated firmware from the device manufacturer to the targeted embedded device.

The update mechanism in our 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 embedded devices.

In the case that the gateway detects the new firmware update, it will check if there is belonging device that needs this update. The embedded device that requires the update will download the binary through a given url from the metadata given by its vendor. As our contribution, we design skipchain-based framework to support firmware update for offline device from untrusted-gateway.

The remainder of this thesis organized as follow: Chapter 2, presents literature study of skipchain and previous work on firmware update mechanism. System architecture, and protocol design are introduced in Chapter 3 Prototype Implementation will be discussed in Chapter 4. In Chapter 5, there are security analysis and evaluation discussion. Finally, a conclusion and future work in Chapter 6.

Chapter 2 Literature Review

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

Consideration that firmware was more secure than other software has been debunked in recent year. In 2011, a cyber attack on embedded battery controller firmware can possibly cause a security hazard such as overheating the battery or even make it catch on fire [12]. Embedded battery controller is used in Li-Ion and Lithium Polymer batteries, these batteries with embedded battery controller usually used in embedded devices. This overheat issue can cause embedded devices permanently broken.

In 2013, Ang Cui [13] demonstrate how firmware update process can be exploited. It allow the attacker to inject malicious firmware modifications into the vulnerable embedded device. Ang Cui et. al successfully exploit the third-party libraries in HP printer firmware images, and inject malicious modification into the library. Their implementation proofed that running malware in printer is capable of 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. In replay or duplication attack, attacker can duplicate the firmware binary from the vendor server and make the victim device check the unnecessary firmware. This redundant checking process will drain the victim device battery. To resist this replay attack, [5] use timestamp and nonce mechanism. Using these techniques, embedded device do not need to check the duplicate firmware within the small interval of time.

There is also man-in-the-middle attack, where attacker can intersect the firmware binary from the vendor. Using reverse engineering algorithm, attacker can modify the firmware and inject malicious code then send it to the victim device. To keep the integrity and the authenticity of the firmware, firmware file can digitally signed using vendor private key. After downloading the firmware file from the 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 process and automatic process. Manual process is vulnerable and not efficient, because user needs to manually download the firmware file and installs it to embedded device. Downloaded firmware integrity and authenticity can not be confirmed by un-experienced end user nor the embedded device because it manually downloaded and installed. On the other hand, automatic update over the air 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 or usually called remote firmware update will update the latest firmware binary to the targeted device without user operation.

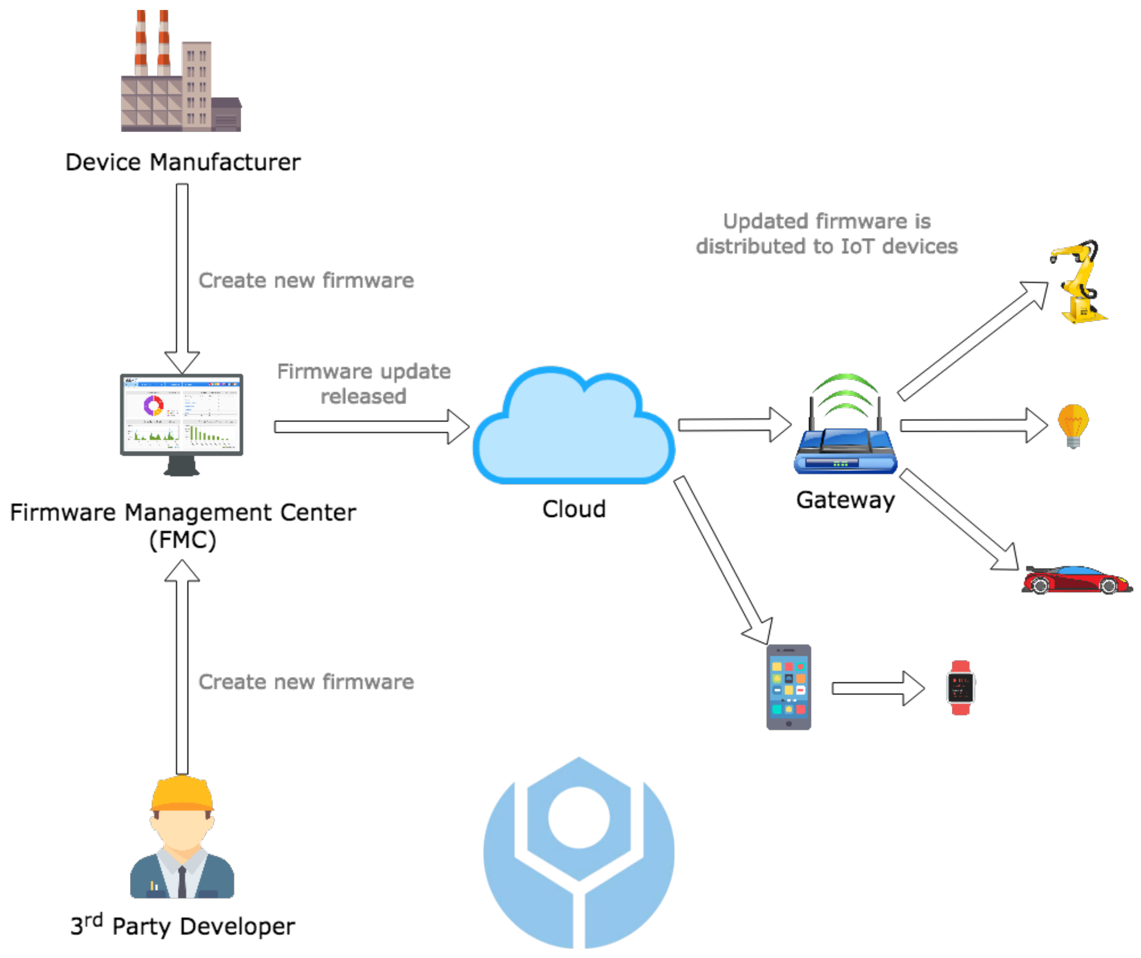


Figure 2.1: Firmware Over The Air (FOTA) process diagram

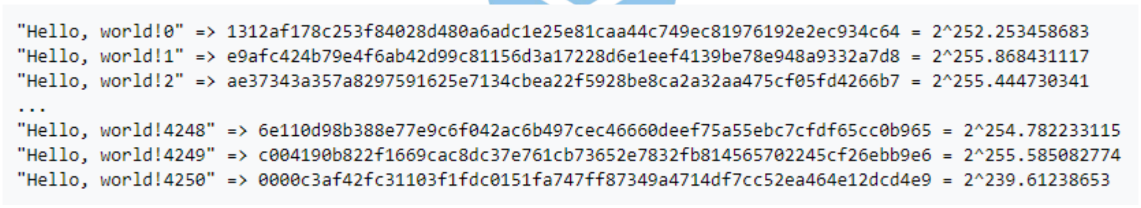
Remote firmware update use two methods in its process, push method and pull method. While push method is applicable for resource constrained device, pull method is more practical for more powerful with stable connection device [14]. Both method use client-server architecture, in push method, vendor repository as server will send the firmware file as chunks to the device as client. After receiving the whole firmware file, client will notify the server an error or success message. If there is no error found, client can do the firmware update. In pull method, vendor repository as server will provide url to download the firmware file to the gateway as client. Once the whole firmware file is downloaded, the error or success message will be sent to the server. Then, client can do the firmware update if the download is successful.

2.2 Blockchain

Based on Satoshi Nakamoto's paper [10], 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 [15] 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" => 6e110d98b388e77e9c6f042ac6b497cec46660deef75a55ebc7cfdf65cc0b965 = 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.

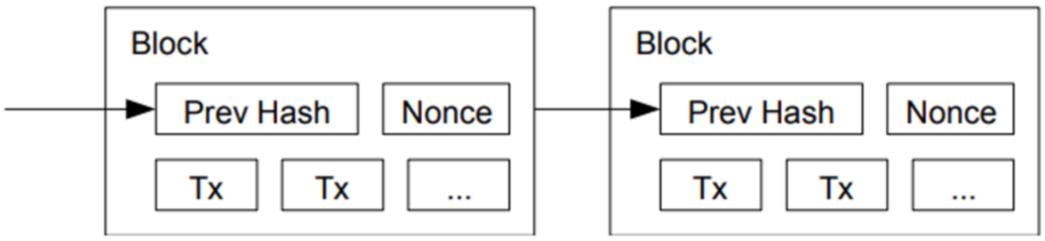


Figure 2.3: Nakamoto's chaining block diagram 2.3

2.3 Skip Lists

Skip lists were first described in 1989 by William Pugh [16]. Skip lists is a data structure that allows fast search within an ordered sequence of elements [17], fast search is possible by skipping over a few element rather than searching linear number of steps. The structure of a skip list is shown in Figure 2.4. The search process to find fifth node (as example), traverse a link of width 1 at the top level. Now four more step are needed, but the next span on this level is too large (10), so drop one level. Traverse one link of width 3, since the next step of width 2 would be too far, drop down to the bottom level. Last traverse the final link of width 1 to reach the fifth link (target).

```
  1                           10
 o---> o---------------------------------------------------------> o   Top level
  1          3              2                    5
 o---> o---------------> o---------> o---------------------------> o   Level 3
  1        2        1        2              3              2
 o---> o---------> o---> o---------> o---------------> o---------> o   Level 2
  1     1     1     1     1     1     1     1     1     1     1
 o---> o---> o---> o---> o---> o---> o---> o---> o---> o---> o---> o   Bottom level

Head   1st   2nd   3rd   4th   5th   6th   7th   8th   9th   10th  NIL
       Node  Node  Node  Node  Node  Node  Node  Node  Node  Node
```

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

There are two types of skip lists based on how the insertion process. Deterministic skip list define how the insertion will be done, for example in J. Ian Munro [2] defines 1-2 skip lists. It require that between any two nodes of height *h* (h > 1) or higher, there exists either one or two nodes of height h-1. The deterministic skip list structure is shown at Figure 2.5

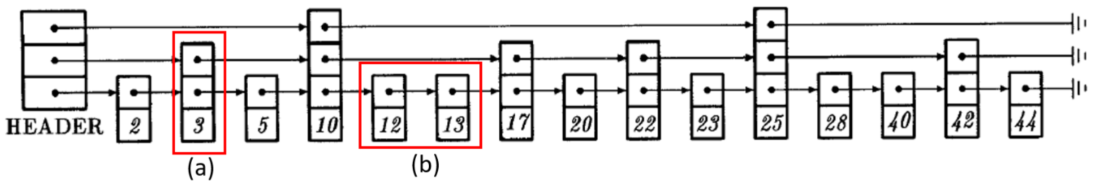


Figure 2.5: (a) There is a node with height of two between two nodes with height of three. (b) There are two nodes with height of one between two nodes of height of two [2]

A randomized skip list will define the new node height based on probabilistic of a coin toss. It will add the height of the newly inserted node until it reach the maximum defined height or once it land on tail. This resulting the average insert and update cost for the skip lists are low. Randomized skip lists can achieve simple implementation but larger storage is needed.

2.4 Skipchain

One of the key characteristic of blockchain is 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 and connected to the Internet and to pay bandwidth and power costs of maintain the connectivity with multiple nodes on blockchain network [11]. User also need to join as a full node, maintaining a mirror copy of the entire blockchain to verify or trace an asset. Even if user can join as a lightweight node and download partial amount the entire blockchain data from connected full node, this process quite bothersome for low-power, small storage and low-connectivity devices.

When lightweight node download the header of the blockchain data from the single compromised node, this compromised node can isolate the lightweight node and present ”fake” view of the blockchain. Lightweight node can do more secure verification if synchronizing with multiple full node. Even full node and well-connected lightweight node client 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] introduce skipchain, a novel cryptographic blockchain structure that adapts the skip list 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. Chainiac also provide long-distance forward link via collective signatures [4]. With long-distance forward and backward link, skipchain becomes cryptographically traversable in both directions.

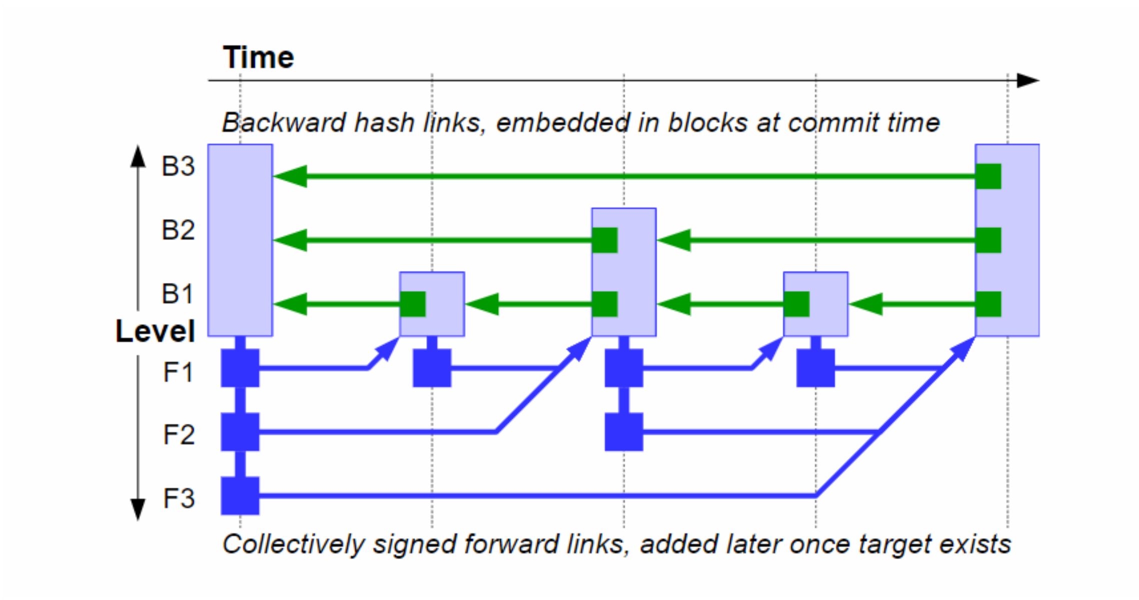

Time
Backward hash links, embedded in blocks at commit time
B3
B2
B1
Level
F1
F2
F3
Collectively signed forward links, added later once target exists

Figure 2.6: Skipchain structure [3]

Using this skipchain characteristic, a node as client can efficiently prove the correctness of a transaction anywhere in time with respect to other party reference point on the blockchain, in a logaritmic number of steps, regardless of which node has a more up-to-date data view of the blockchain [11]. For example, a gateway in smart home environment wants to show a new update to embedded device in the same environment. In this case, embedded device has limitation to connect to the Internet and synchronize with the other node. It also can not store even partial part the blockchain data due to its small storage. Gateway can simply sends to embedded device a small number of collectively signed forward links through a peer-to-peer connection to prove securely that the updates is indeed on the blockchain.

2.5 Blockchain-based Firmware Update Framework

In this section, previous research on the firmware update mechanism for embedded devices based on blockchain infrastructure is introduced. Lee et al. [18] introduce the blockchain-based firmware verification and update schemes for embedded devices in IoT environment. In [18], mentions 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 [18] architecture, embedded device acts as normal node in the blockchain network which is difficult in real world scenario. Because embedded device with low-power, small storage and connection-restricted is difficult to maintain connection with the other node in blockchain network. It is also not possible for embedded device to store growing blockchain data. Moreover, this scheme uses poll-method, when there are many kind of embedded device with different requirement of firmware, it will take a long time to wait the download request made by each device. Thus, this scheme is might not efficient for heterogeneous IoT ecosystem.

Boudguiga et al. [19] investigate that it is possible to use blockchain infrastructure to provide firmware update to several IoT devices belonging to different manufacturer. This scheme rely 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 architecture introduced by [19], in the first architecture, each vendor is expected to provide at least one worker node in the blockchain network. An IoT device can periodically poll the blockchain by random picks the node and check the firmware version. The second architecture as a refinement from the first architecture, the trusted node role takes place. The trusted node directly receive new firmware binary from the vendor, the trusted node then notifies the corresponding devices after the validation process complete.

Chapter 3 System Environment and Protocol Designs

This chapter describes the assumptions, overview on the skipchains, and how we design the system architecture. In Section 3.1, we explain the assumptions about the environment in which our framwework will be used. Section 3.2 will be a brief overview of the technology we use, skipchain. Section 3.3 covers our framework's architecture design. In Section 3.4, we describe our protocol design.

3.1 Assumptions

The assumptions used in our proposed framework are listed as follows:

1. Use push-update method, where vendor repository will send the firmware update metadata to the skipchain network. Once verified, gateway will get synchronize with the skipchain network overtime and get the new firmware update notification and check if the update is required by the embbeded devices connected to it.

2. The embedded devices in the deployed environment are low-power, small storage and connection-restricted devices. But it has computational capacity to do simple hash function and simple comparison.

3. Embedded device has a pre-installed vendor's public key to authenticate signed firmware

4. Each embedded device has pre-installed small part of the skipchain data to provide the gateway a link to join the network. The pre-

installed data also enables embedded device to securely verify the more recent skipchain data from gateway.

5. We assume the connection to download the firmware binary through TLS is secure.

6. The connection between vendor's repository and vendor's node that join blockhain network does not need to be secure.

7. Peer-to-peer connection between gateway and embedded devices is not secure.

3.2 Skipchain overview

According to the previous chapter, we know that skipchain implements skip list idea into a blockchain structure by adding long-distance links both forward and backward in time. Backward link has been implemented in traditional blockchain, which contain a hashed-address to previously created block. This backward link allows user to trace a transaction history back in time from the latest block. User can use this tracing feature when user's device contains more recent block data than the transaction history's block.

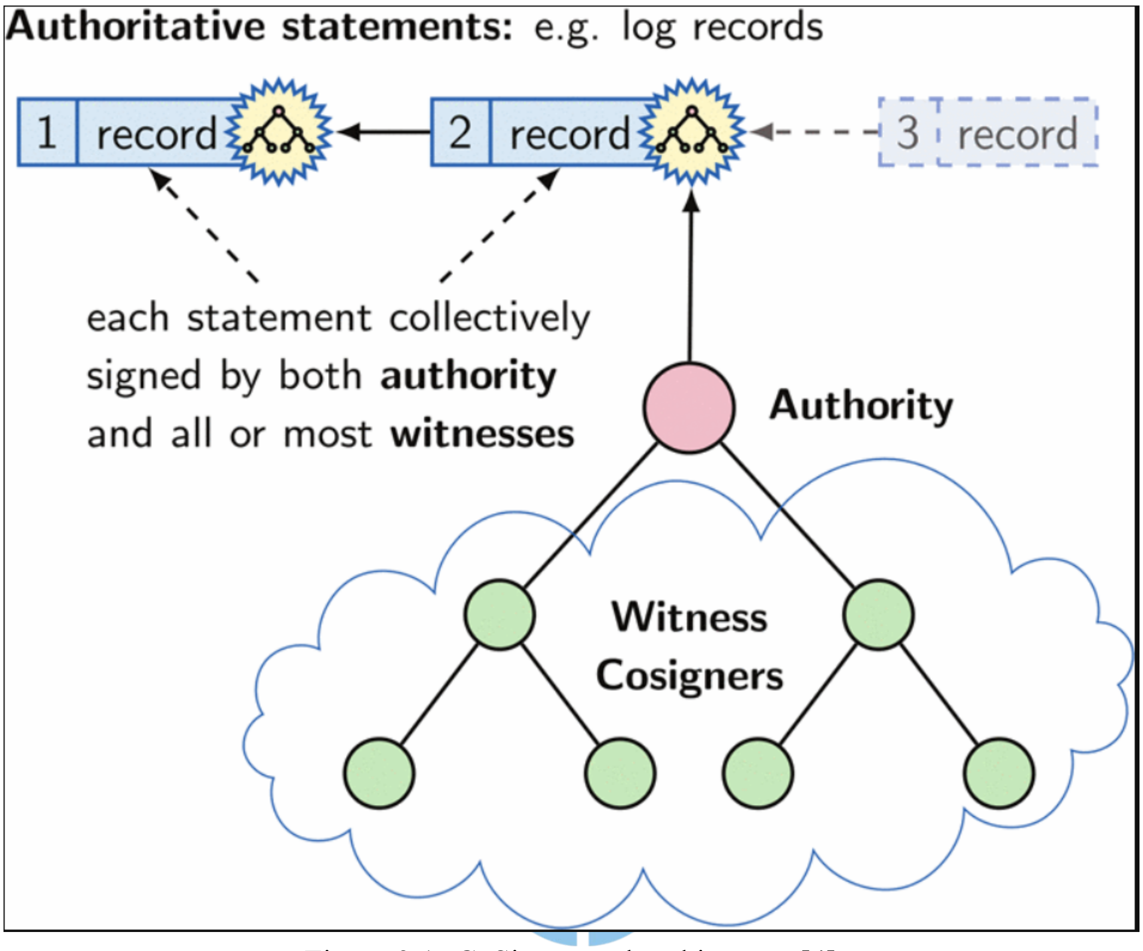
The unique forward link in skipchain structure enables user to verify transaction history even the user's blockchain data is behind the recent blockchain data. Forward link contains two pieces of information:

1. A secure hash-pointer to the first block committed by the next consensus group

2. A description of how the consensus group changed, which cosigner's public keys were added and/or removed

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

Skipchain uses a tree-based collective signing (CoSi) [4] schemes on which each signer, called *cosigner*, join Schnorr signatures [20] together to agree on a valid statement. The group of cosigner, called collective authority (cothority), do the consensus-like process where each cosigner will verify and sign a valid transaction. The CoSi architecture is illustrated in Figure 3.1. From the figure below, we can see that each record could be collectively signed by different cothority member.


Authoritative statements: e.g. log records
1
record
2
record
3
record
each statement collectively signed by both authority and all or most witnesses
Authority
Witness Cosigners
Figure 3.1: CoSi protocol architecture [4]

Using the collectively signed forward links, user can securely walk forward in time from his last blockchain reference without having to trust another party. User can learn about all relevant consensus group changes along the way, so that user knows exactly correct set of public keys against which to check each collective signature in the chain. Since all of these forward links are collectively signed by many cosigner, attacker can not create a fake blockchain unless it compromises or colludes with large number of cothority member.

3.3 Architecture Design

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

1. Vendor: Manufacturer of the specific embedded device, join skipchain network as active node. Active node means it is actively do the verification process as a Cothority's member. Each time vendor pushes a new firmware update, vendor repository need to broadcast his signed-firmware-metadata to the Cothority via his active node.

2. Gateway: Gateway for connected embedded device, needs to join the skipchain network as passive node. Passive node means it will not be participated in any verification process, but only synchronize with most recent skipchain data.

3. Embedded device: Need to connect to specific gateway, and do not need to trust the gateway. Each embedded device can verify the firmware update notification from the gateway using its pre-installed skipchain data.

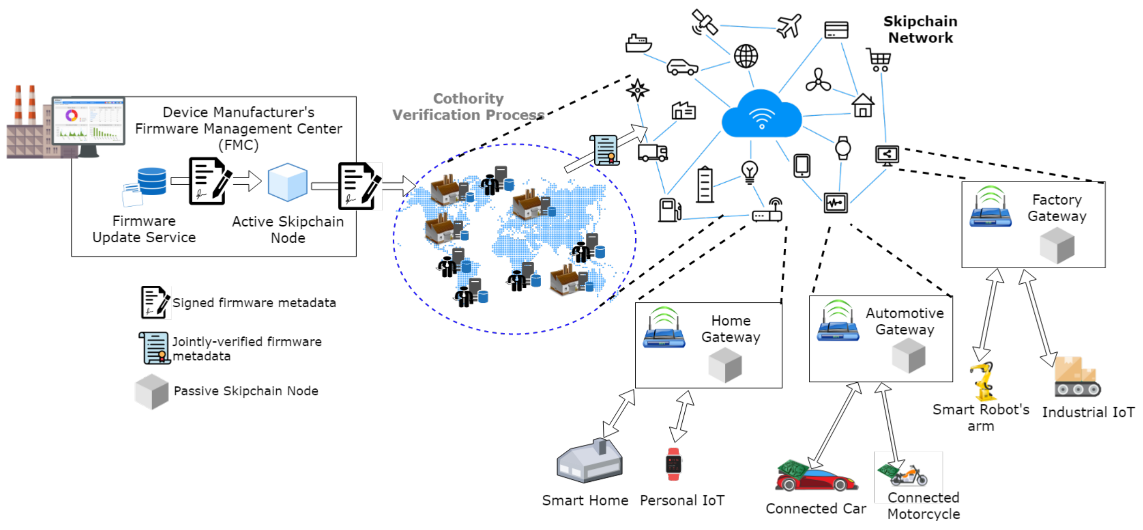


Figure 3.2: Firmware Over The Skipchain (FOTS) process diagram

From the Figure 3.2, there are two kinds of node in the skipchain network. First, active node, will actively do the verification process and join as Cothority's member. Each vendor need to join the skipchain network as an active node. Second, passive node, will continuously synchronize with the skipchain network, fetching the newest data from the network. Gateway need to join the skipchain network as passive node, gateway does not need to verify the newly created firmware metadata. Each gateway has connected embedded device in the local network, for example there are smart home appliances and personal embedded devices connects to home router as gateway. The connection between gateway and embedded device is peer-to-peer connection, like Bluetooth connection.

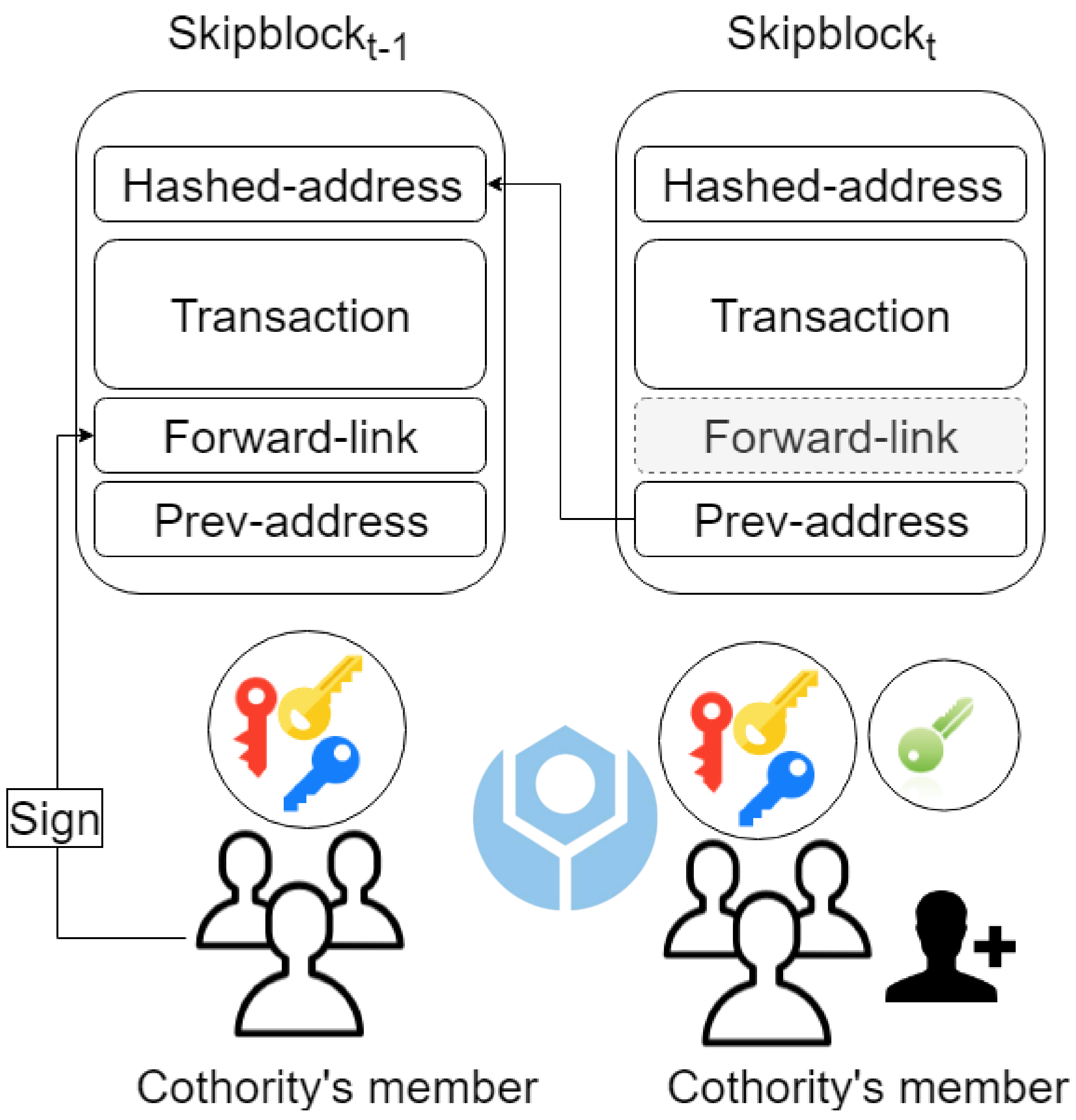


Figure 3.3: Skipblock structure diagram

The skipblock structure as shown in Figure 3.3, consist of three main content. The three contents will be described as follow:

1. Hashed-address: hashed-value that is unique and will be the address for skipblock. Skipblock is accessible through this hashed-address, and the newer created block will contain this value as the previous address.

2. Transaction: the transaction section contains firmware update meta-

data, such as targeted device model, the latest firmware version, and url to download the firmware binary

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.

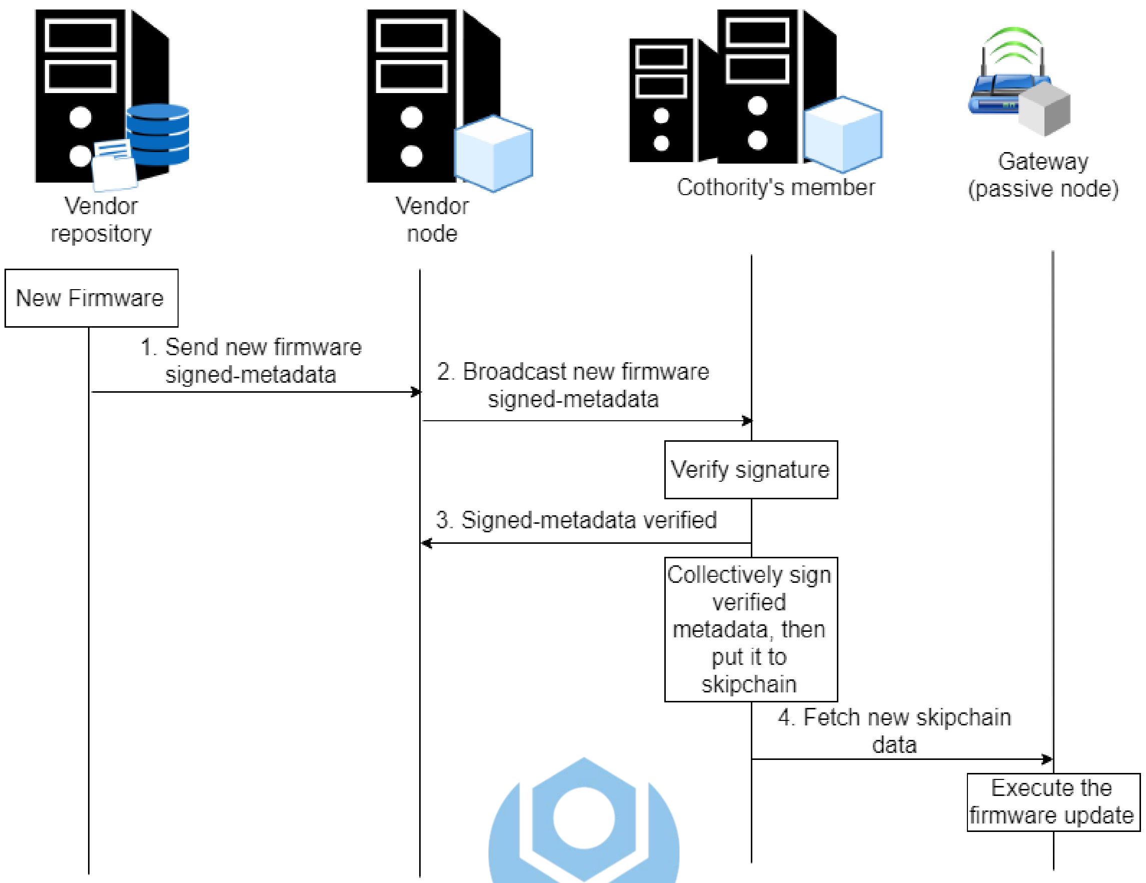
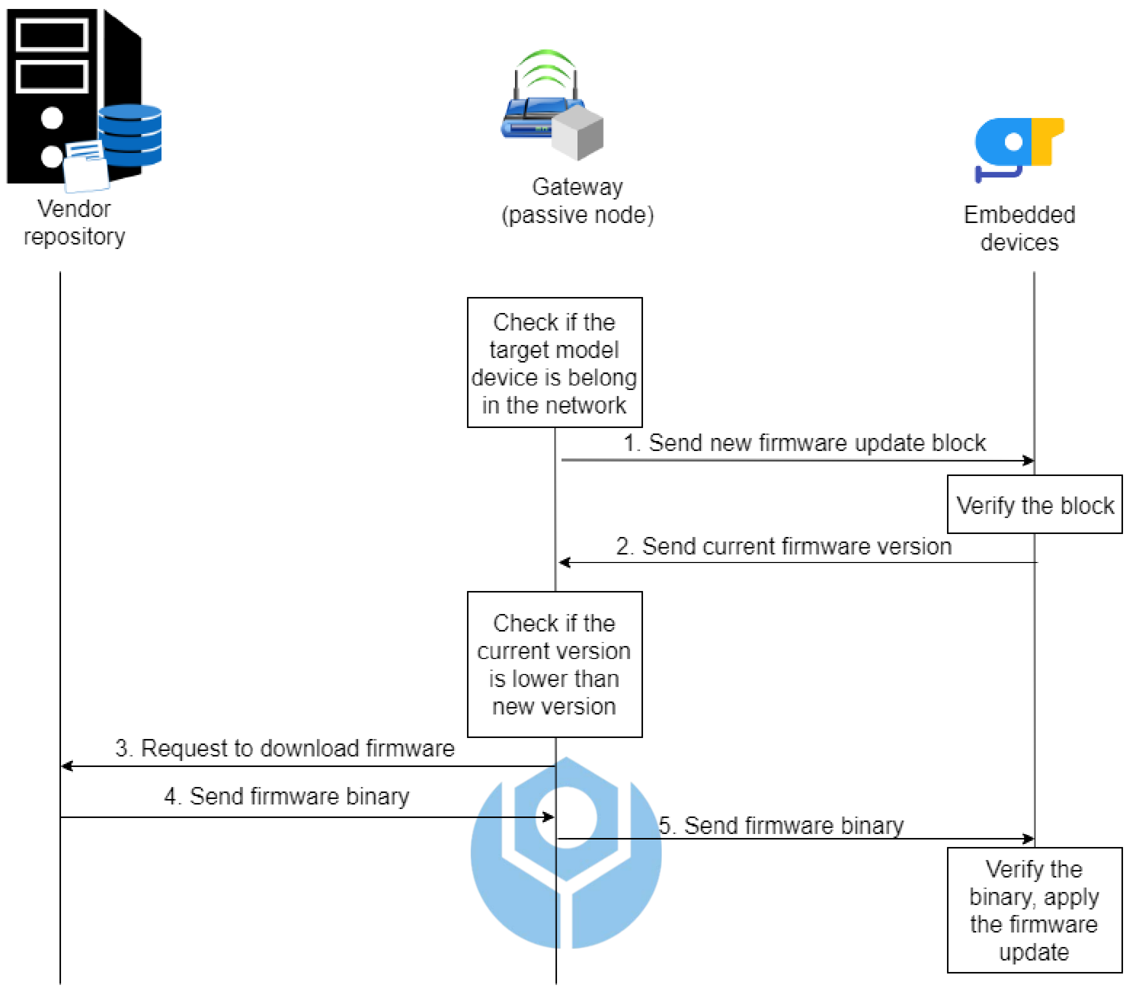

Vendor repository
Vendor node
Cothority's member
Gateway (passive node)
New Firmware
1. Send new firmware signed-metadata
2. Broadcast new firmware signed-metadata
Verify signature
3. Signed-metadata verified
Collectively sign verified metadata, then put it to skipchain
4. Fetch new skipchain data
Execute the firmware update

Figure 3.4: Firmware update verification process

Once embedded device deployed to specific environment, for example smart air conditioner deployed in the smart home environment, the smart air conditioner will connect to the home router as an gateway. Gateway will join the skipchain network as passive node, and continuously synchronize with the skipchain network. In our proposed scheme, we devide the overall process into three parts. The first part is firmware verification process, this process will leverage how the new firmware update metadata verified and put into the skipchain. After putting the metadata into the skipchain, the second process, peer-to-peer verification process begin. After the embedded device verifies the firmware update notification (in peer-to-peer connection) from gateway, embedded will execute the firmware update. The firmware verification process of the proposed framework is shown in Figure 3.4, described as follow:

1. Each time vendor make a new firmware update, it sign the new firmware update's metadata, including url to download the firmware, what is the new firmware version, what is targeted device model for this firmware (more detailed data structure will be discussed in chapter 4) using its private key.
2. Then vendor node will broadcast the signed-metadata to the other cothority's member to be verified using the corresponding public key.
3. After the verification process, cothority will collectively sign the valid metadata and put it to the skipchain network.
4. Gateway as passive node, which is continuously synchronize with the skipchain network will fetch the new firmware update information and execute the firmware update.

The firmware update execution process is illustrated in Figure 3.5 below. When notified, gateway will check the newly put metadata. It will notify the target device if targeted device model is within its belonging embedded devices. Targeted embedded device then can verify if the update is collected from the valid skipchain. Once the update verified, embedded device will send its current firmware version to the gateway. Gateway will check if the current version is older than the new version, and download the new firmware from the url provided in the metadata. Downloaded firmware will be passed, verified, and applied to the targeted embedded device.



Vendor repository
Gateway (passive node)
Embedded devices
Check if the target model device is belong in the network
1. Send new firmware update block
Verify the block
2. Send current firmware version
Check if the current version is lower than new version
3. Request to download firmware
4. Send firmware binary
5. Send firmware binary
Verify the binary, apply the firmware update

Figure 3.5: Firmware update execution process

Most crucial part in the firmware update execution is the block verification by the embedded device. This process is happened in peer-to-peer condition, allowing offline embedded device to verify the given block. Using the skipchain's forward link, embedded device can securely verify the newly created block without synchronizing and store any additional data. Instead, gateway as stronger device with more storage capacity and internet connection can do the synchronization process. This peer-to-peer verification also helpful when a gateway is not connected to the Internet. This offline-gateway can get the firmware update notification from another online-gateway via peer-to-peer notification. The offline-gateway still able to verify the new firmware update notification not blindly trust the online-gateway. The peer-to-peer verification process is illustrated in Figure 3.6

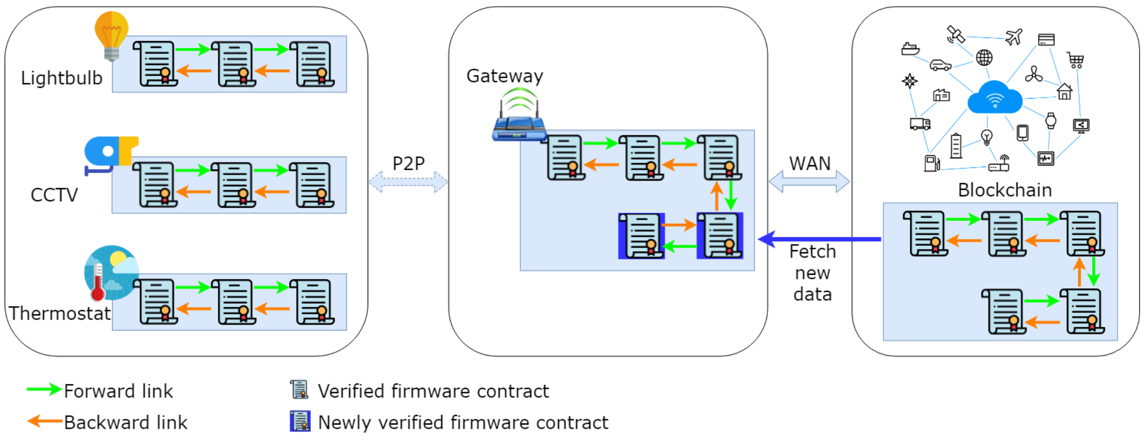


Figure 3.6: Peer-to-peer verification process

In the peer-to-peer verification process, the pre-installed skipblock is important. The pre-installed skipblock structure is illustrated in Figure 3.7. It will contain few skipblocks till a reference point of time. The pre-installed skipblock also contain list of public key (red, yellow and blue key) to verify the forward link given to it via peer-to-peer connection.

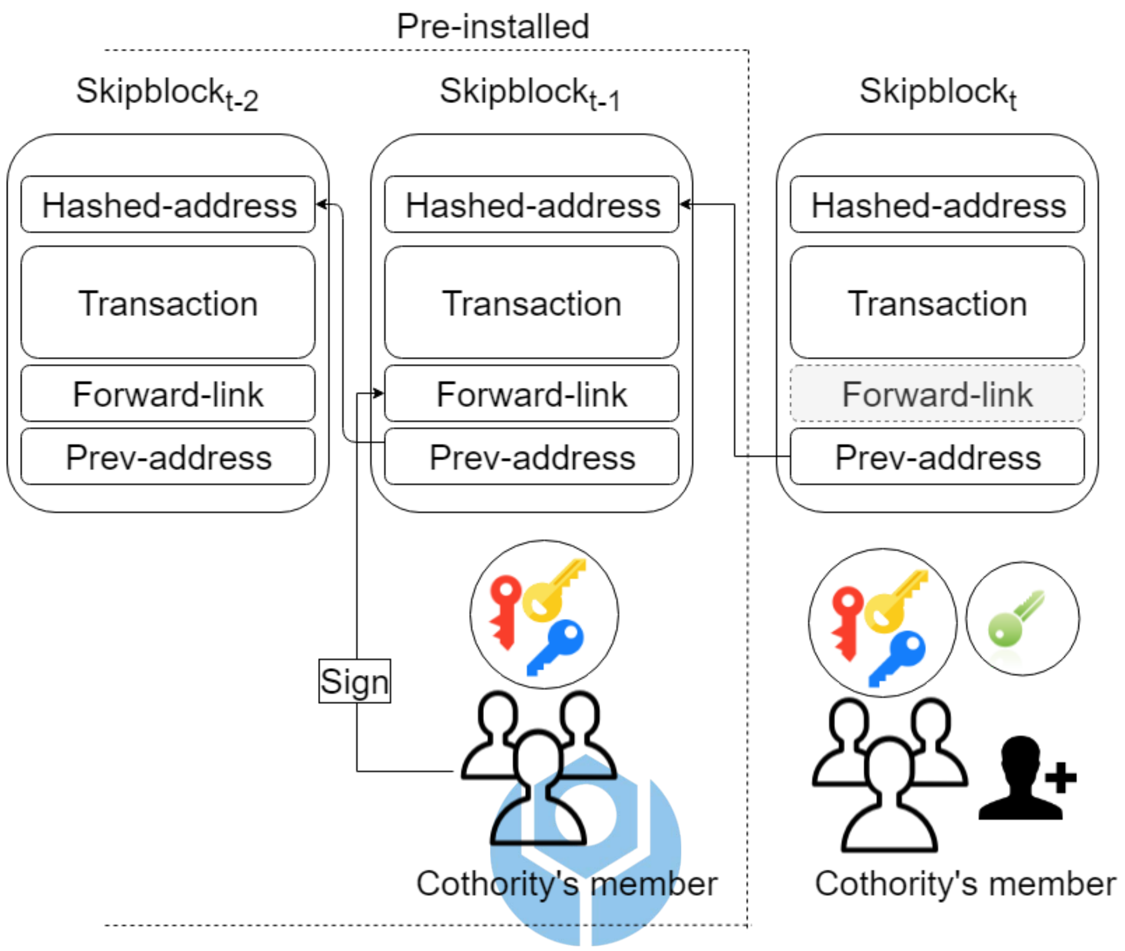


Figure 3.7: Pre-installed skipblock diagram

3.4 Protocol Design

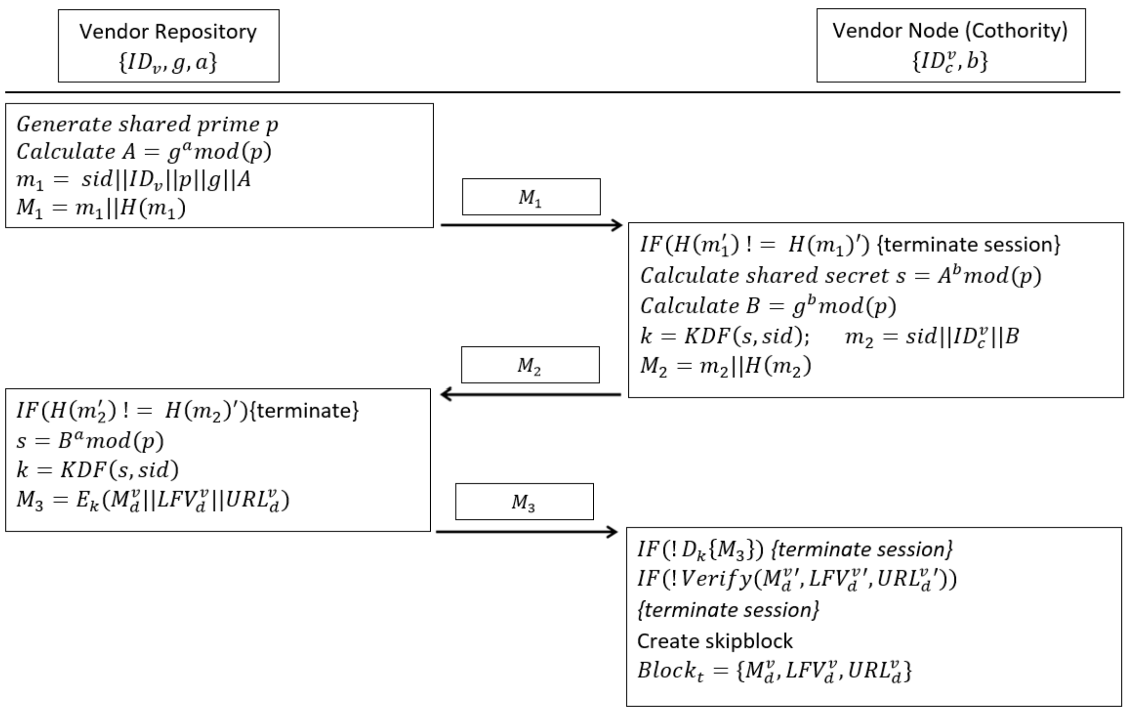
In this section, proposed firmware update protocol is introduced and explained. There are three stages in the firmware update protocol: firmware update verification process, peer-to-peer verification between embedded device and gateway, execution process. Table 3.1 shows the notation that we use in our protocol.

Table 3.1: Notation used in the proposed protocol

Notation	Definition
$\boldsymbol{ID_d}$	ID of embedded device *d*
$\boldsymbol{ID_g}$	ID of gateway *g*
$\boldsymbol{ID_v}$	ID of vendor *v*
$\boldsymbol{ID_b}$	ID of skipblock *b*
$\boldsymbol{ID_c^v}$	ID of cosigner node *c* owns by vendor *v*
$\boldsymbol{k}$	session key
$\boldsymbol{sid}$	ID of specific session
$\boldsymbol{p}$	Randomly generated shared prime number in key exchange
$\boldsymbol{g}$	Shared base number in key exchange
$\boldsymbol{s}$	Shared secret between key exchange
$\boldsymbol{a, b}$	Secret owns by Alice and Bob in key exchange
$\boldsymbol{URL_d^v}$	Url to download firmware binary for device *d* from vendor *v*
$\boldsymbol{CFV_d^v}$	Current firmware version for device *d* from vendor *v*
$\boldsymbol{LFV_d^v}$	Latest firmware version for device *d* from vendor *v*
$\boldsymbol{LFB_d^v}$	Latest firmware binary for device *d* from vendor *v*
$\boldsymbol{M_d^v}$	Device model for target device *d* from vendor *v*
$\boldsymbol{BLOCK_s}$	Pre-installed skipchain *s* block from the vendor
$\boldsymbol{BLOCK_t}$	Skipchain block in newest timestamp *t*
$\boldsymbol{KDF}()$	Key derivation function input: passphrase and salt; output: session key *k*
$\boldsymbol{E_k}()$	Symmetric encryption using session key *k*
$\boldsymbol{D_k}()$	Symmetric decryption using session key *k*
$\boldsymbol{H}()$	Hash-chain function
$\vert\vert$	String concatenation

3.4.1 Firmware Update Verification Protocol

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


Vendor Repository $\{ID_v, g, a\}$
Vendor Node (Cothority) $\{ID_c^v, b\}$
Generate shared prime p
Calculate $A = g^a mod(p)$
$m_1 = sid||ID_v||p||g||A$
$M_1 = m_1||H(m_1)$
M_1
$IF(H(m_1') != H(m_1)')$ {terminate session}
Calculate shared secret $s = A^b mod(p)$
Calculate $B = g^b mod(p)$
$k = KDF(s, sid)$; $m_2 = sid||ID_c^v||B$
$M_2 = m_2||H(m_2)$
M_2
$IF(H(m_2') != H(m_2)')$ {terminate}
$s = B^a mod(p)$
$k = KDF(s, sid)$
$M_3 = E_k(M_d^v||LFV_d^v||URL_d^v)$
M_3
$IF(!D_k\{M_3\})$ {terminate session}
$IF(!Verify(M_d^v{}', LFV_d^v{}', URL_d^v{}'))$
{terminate session}
Create skipblock
$Block_t = \{M_d^v, LFV_d^v, URL_d^v\}$
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 $A = g^a mod(p)$. 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 $s = A'^b mod(p')$ and shared number $B = g'^b mod(p')$. 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 $m_2 = (sid||ID_c^v||B)$, 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

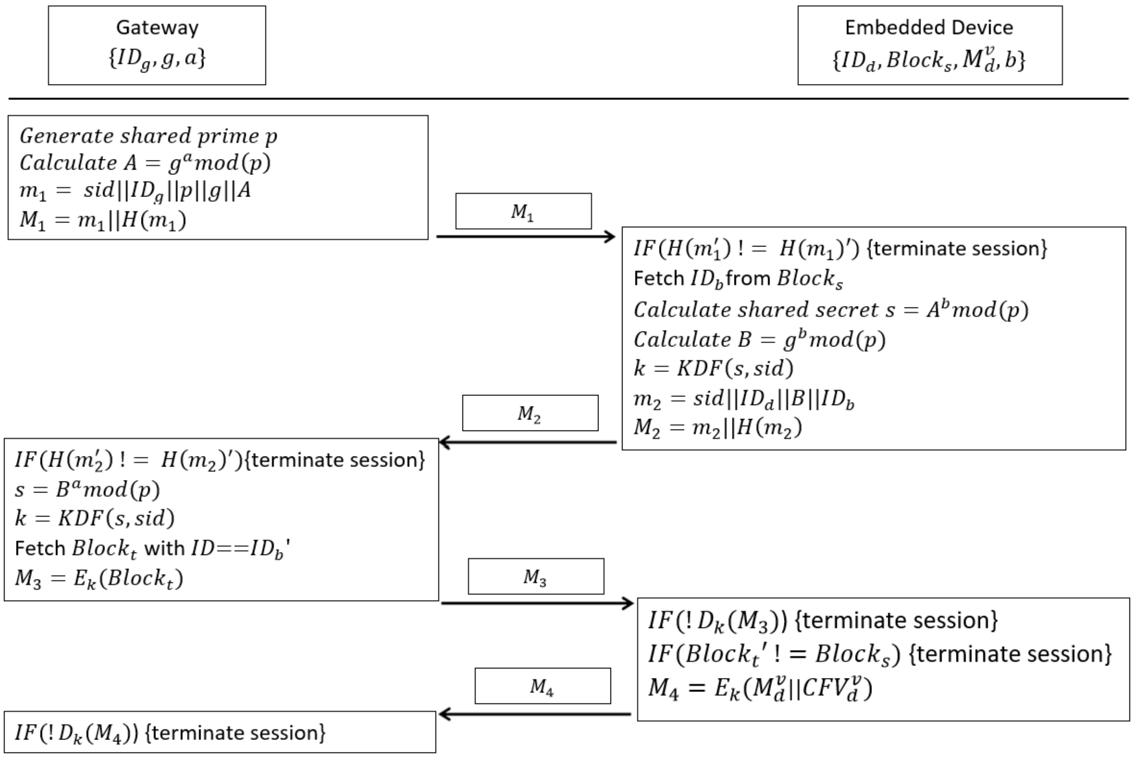
Once receive m_2', vendor repository will do the validation toward the message m_2' and hashed-value $H(m_2)'$. If the message is valid ($H(m_2') == H(m_2)'$), vendor repository will obtain session id *sid'*, vendor node's id $ID_c^{v'}$ and shared number *B'* then calculate shared secret $s = B'^a mod(p)$. It will also use KDF to create same session key k. Last, vendor repository encrypt message $m_3 = E_k(M_d^v||LFV_d^v||URL_d^v)$ 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 embedded device d. This process is needed everytime gateway fetch new block data from the skipchain network (URL_s). Each embedded 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.



Gateway $\{ID_g, g, a\}$
Embedded Device $\{ID_d, Block_s, M_d^v, b\}$
$Generate\ shared\ prime\ p$
$Calculate\ A = g^a mod(p)$
$m_1 = sid||ID_g||p||g||A$
$M_1 = m_1||H(m_1)$
M_1
$IF(H(m_1')\ != H(m_1)')$ {terminate session}
Fetch ID_b from $Block_s$
$Calculate\ shared\ secret\ s = A^b mod(p)$
$Calculate\ B = g^b mod(p)$
$k = KDF(s, sid)$
$m_2 = sid||ID_d||B||ID_b$
$M_2 = m_2||H(m_2)$
M_2
$IF(H(m_2')\ != H(m_2)')$ {terminate session}
$s = B^a mod(p)$
$k = KDF(s, sid)$
Fetch $Block_t$ with $ID == ID_b'$
$M_3 = E_k(Block_t)$
M_3
$IF(!\,D_k(M_3))$ {terminate session}
$IF(Block_t'\ != Block_s)$ {terminate session}
$M_4 = E_k(M_d^v||CFV_d^v)$
M_4
$IF(!\,D_k(M_4))$ {terminate session}

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

Step 1: Gateway $\rightarrow$ Embedded Device

Gateway will generate a shared prime p, and calculate a shared number

$A = g^a mod(p)$. 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 embedded device.

Step 2: Embedded Device $\rightarrow$ Gateway

Upon received message m_1' from gateway, embedded 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, embedded device calculate shared secret $s = A'^b mod(p')$ and shared number $B = g'^b mod(p')$. Embedded device uses key derivation function $KDF()$, passing shared secret s and session id sid as salt to create new session key k. Embedded 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 $\rightarrow$ Embedded 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'*, embedded device's id $ID_d^{v'}$, skipblock id ID_b' and shared number *B'* then calculate shared secret $s = B'^a mod(p)$. 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 embedded device.

Step 4: Embedded Device $\rightarrow$ Gateway

Embedded device will decrypt the encrypted message m_3' using its session key k, and obtain the new firmware update block. Embedded device will verify the obtained $BLOCK'_t$ if it is matched with its pre-installed block $BLOCK_s$. Once verified, embedded 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, embedded 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 embedded 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 embedded 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.

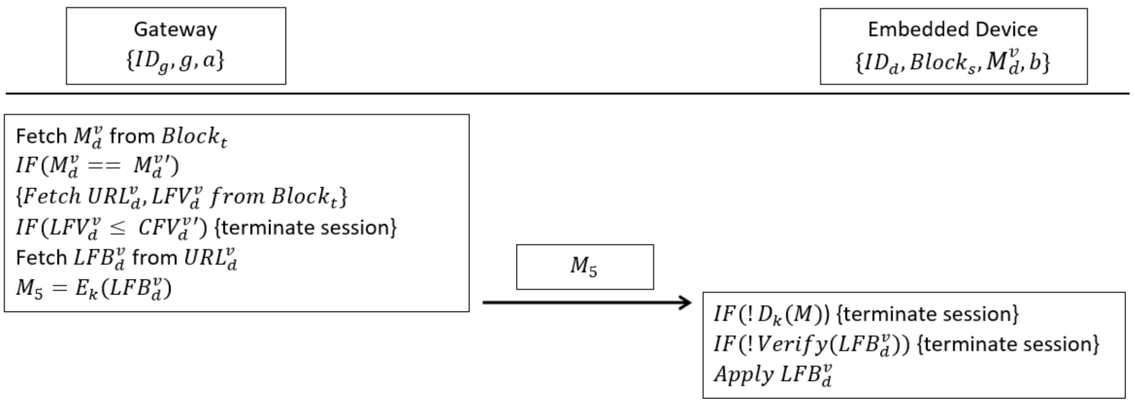

Gateway
$\{ID_g, g, a\}$
Embedded Device
$\{ID_d, Block_s, M_d^v, b\}$
Fetch M_d^v from $Block_t$
$IF(M_d^v == M_d^{v\prime})$
$\{Fetch\ URL_d^v, LFV_d^v\ from\ Block_t\}$
$IF(LFV_d^v \leq CFV_d^{v\prime})$ {terminate session}
Fetch LFB_d^v from URL_d^v
$M_5 = E_k(LFB_d^v)$
M_5
$IF(!\,D_k(M))$ {terminate session}
$IF(!\,Verify(LFB_d^v))$ {terminate session}
$Apply\ LFB_d^v$

Figure 3.10: Firmware update execution protocol

Step 1: Gateway $\rightarrow$ Embedded 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 embedded device model $M_d^{v'}$ belonging to it. If there is targeted device model within its belonging embedded device ($M_d^v == M_d^{v'}$), it will fetch the download url URL_d^v and the latest firmware version LFV_d^v from the block $BLOCK_t$. Gateway will check if the latest firmware version is greater than the device current firmware version ($LFV_d^v \geq CFV_d^{v'}$). If the update is required, gateway will fetch the firmware binary LFB_d^v from the download 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 embedded device.

Step 2: Embedded Device $\rightarrow$ Gateway

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

Chapter 4 Prototype Implementation

This chapter describes prototype design in Section 4.1. Implementation of the proposed framework and the result of the experiments described in Section 4.2.

4.1 Prototype Design

This section describe 2 main procedures used in the prototype implementation. First procedure, key exchange procedure is described in Subsection 4.1.1. Second procedure, encryption and decryption function is described in Subsection 4.1.2 and Subsection 4.1.3 respectively.

4.1.1 Key Exchange Procedure

We have been implemented two type of prototype with two different connections. First prototype (repository-node prototype) simulate protocol between vendor repository as client and vendor node as server, this prototype uses the Internet for the information exchange. Second prototype (gateway-device prototype) implemented client-server architecture between gateway and embedded device, the second prototype uses Bluetooth connection to exchange the information. Gateway takes the role of Bluetooth server and the embedded device takes role of Bluetooth client.

Both prototype uses Diffie-Hellman [21] key exchange procedure as describe in Algorithm 1,2,3. In initiation procedure of Algorithm 1, **vendor repository** in repository-node prototype uses $WIFI\_CLIENT\_SOCKET$ and **embedded device** in gateway-device prototype uses $BLUETOOTH\_CLIENT\_SOCKET$. Client sets some pre-defined variables used in key exchange procedure. After the setting is complete, client concatenates the variables as one string message and concatenates the it with its hashed-value. Client uses the $CLIENT\_SOCKET$ to send the whole message.

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:     $SET\ ID_v : "vendorA\_repo\_1"$
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:     $CALCULATE\ A = g^a mod(p)$
10:    $m = (sid||ID_v||g||p||A) \leftarrow message$
11:    $CLIENT\_SOCKET.SEND(m||SHA256(m))$
```

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 $BLUETOOTH\_SERVER\_SOCKET$. $SERVER\_SOCKET$ receives the message sent from client in initiation procedure, server fetch the concatenated message by using split function.

The split function results array, if the array length not match in required procedure (6), server will cut the session. If the array length match the required length, server put each array component into different variables.

The verification process checks if the hashed-value in the last array matches with the newly hashed variables.

Once the message is verified, server calculate the shared secret s and uses PBKDF2 [22], key derivation function that applies hash-based message authentication code (HMAC) to the input passphrase (s) along with a salt value (sid). The function repeats the process many times to produce a derived key, which can then 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.

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 $SHARED\_SECRET$.

Algorithm 2 Server Key Exchange Response

```
1: procedure KEY_EXCHANGE_PROCEDURE_SERVER
2:     $INIT\ SERVER\_SOCKET(HOST : 127.0.0.1, PORT : 8080)$
3:     $SET\ ID_c$ : "$id\_vendorA\_cothority\_1$"
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 $(ARRAY\_DATA.LENGTH()\ ! = 6)$ 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:            $m = (sid'||ID_c||B) \leftarrow message$
22:            $SERVER\_SOCKET.SEND(m||SHA256.H(m))$
23:    else
24:        $TERMINATE\_SESSION()$
```

The procedure after the client gets the message from Algorithm 2 is described in Algorithm 3. First step, client splits the response message from the server. If the array length match the required length (4), client put each array component into different variables. Client runs verification process checks if the hashed-value in the last array matches with the newly hashed variables. If the message verified, client calculate the shared secret uses the $SHARED\ NUMBER,\ B$. Last step, client uses PBKDF2 to produce same session key as 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:             $s = B'^{(a)} mod(p') \leftarrow shared\ secret$
15:             $k = PBKDF2(s, sid') \leftarrow session\ key$
16:     else
17:         $TERMINATE\_SESSION()$
```

4.1.2 AES Encryption Function

After client and server have the same session key, they can use the session key as AES symmetric key. Client and server can use the AES encryption architecture to send sensitive information. AES encryption function described in Algorithm 4. The encryptor to have a symmetric key and a salt. The encryptor creates an AES object with session key k as parameter. The encryptor uses session id sid as salt and generate nonce value IV. The function encrypts the plain text using the nonce IV, then concat the hex-value of the result, salt and 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

Using the session key k, the other party can decrypt the cipher text created by Algorithm 4. The AES decryption fucntion described in Algorithm 5, where the decryptor create the same AES object as encryptor using session key k. Decryptor split the received cipher text, then unhex it using $UNHEX()$ function. After separating the salt from the nonce (IV) and the actual cipher text, decryptor using the decryption function from the AES object. The decryption function needs nonce value (IV) and the cipher text, it 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 experiment, we use laptop, a home pc, and a raspberry pi with the following specification:

1. MSI GL62 laptop as vendor repository equipped with Intel i7 2,6GHz, 16 GB RAM, SSD 512MB.

2. Asus PC as vendor node and gateway equipped with Intel i5 2,3GHz, 12 GB RAM, HDD, and Bluetooth 4.0 dongle. This PC run 4 virtual device as active node and 1 virtual device as passive node (gateway function) to simulate the skipchain environment.

3. Raspberry Pi 3 as embedded device, equipped with CPU: ARM Cortex 1,4GHz, 1GB SRAM, Integrated Bluetooth 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 code still under hard development, we could not find usable API. In result, we use python library, **subprocess**, to access the skipchain via terminal.

We conduct two experiments, first we do protocol experiment between vendor repository and vendor node over the Internet connection. Average running time for the first protocol experiment is one second, the process of putting the data to the skipchain takes 90% of the running time. Second, we conduct protocol experiment between gateway and embedded device over Bluetooth connection. Average running time for the second protocol is one second. It takes 50 microseconds to send 10 megabytes of firmware data over the Bluetooth connection.

Chapter 5 Security Analysis & Discussion

In this chapter, a 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. In last section, Section 5.3 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. [18]**	**Lee et al. [18]**	**Boudguiga et al. [18]**
Firmware update method	Push-method	Push-method	Poll-method	Poll-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 use push-method to update the 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 [18], [19] use poll-method, where an embedded device need 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 publish new firmware update. Push-method also more energy and network efficient than poll-method, because in poll-method, power-constrained and connection-limited embedded device need to constantly request the availability of the new firmware version whether it is exist or not.

Our framework achitecture adopts skipchain technology, which enables smart contract creation. Skipchain as permissioned blockchain platform protocols 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 tackle the energy required by proof-of-work consensus mechanism with drawback of no anonimity within the permissioned blockchain network.

Using the skipchain technology, forward link, our proposed framework also provide peer-to-peer block verification process. It means, 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 is exist in the valid skipchain network without storing the whole blockchain data.

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. [18]**	**Lee et al. [18]**	**Boudguiga et al. [18]**
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

The comparison of the security mechanism between proposed framework and the other related works shown in Table 5.2. Proposed framework is designed to protect firmware update process from major cyber-attack, respectively: firmware modification attack, impersonation attack, man-in-the-middle attack, replay attack and isolation attack. On the other hand, Lee et al. [18] might not prevent firmware modification and impersonation attack. Furthermore, only our proposed framework that provide security mechanism 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 Security against attack within information exchange over insecure connection

The following theorems are applied on both firmware verification protocol and peer-to-peer verification protocol. **Vendor repository** roles in firmware verification protocol and **embedded 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 $s = B^a mod(p)$. On the other hand, server uses the shared public key A from client and its private key b, to calculate the same shared secret $s = A^b mod(p)$. 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 need to solve the Diffie-Hellman problem to get the current shared secret s and use the same key derivation function $KDF()$ as 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 target of the adversary to impersonates to, a vendor repository, a vendor node, or a 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 embedded device that there is new fake firmware update. The forward link in skipchain enables embedded 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 certain 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*

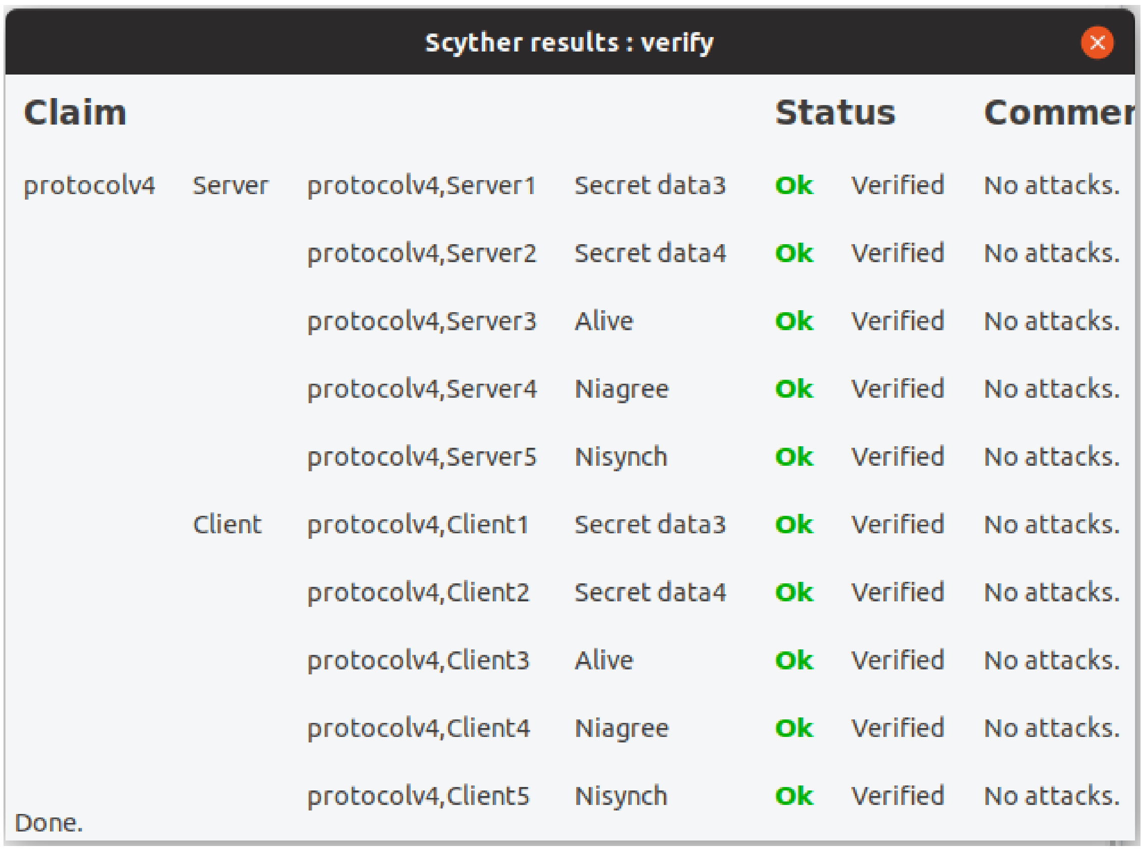
Proof. Based on the proof given for Theorem 1, proposed protocol provides mutual authentication between legitimate client and legitimate server. In addition, adversary can not use the session key k to decrypt and get the important metadata and the firmware binary during the process. Thus, proposed protocol 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 embedded 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, embedded device can detect the firmware binary is invalid. Embedded device with pre-installed vendor's public key can verify the signed-firmware-binary. □

5.2 Formal 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 client and vendor node acts as server. In peer-to-peer verification protocol, embedded device acts as client and gateway acts as 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 liveliness 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 [23], 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 syncronisation means the send event from the first run *send_1* followed by corresponding read *recv_1* and so on till the send-read event in the end of the protocol. The claim used in the SPDL code is:

$\#server\ role$

$\boldsymbol{claim(Server, Alive)}$

$\#client\ role$

$\boldsymbol{claim(Client, Alive)}$

5.3 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 use 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 protect 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 metadata 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, enchance 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 enchance 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] K. Salah and M. Ahmad Khan, "Iot security: Review, blockchain solutions, and open challenges," *Future Generation Computer Systems*, 11 2017.

[6] 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].

[7] 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.

[8] 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.

[9] P. Point, "Proofpoint uncovers internet of things (iot) cyberattack." `https://www.proofpoint.com/us/proofpoint-uncovers-internet-things-iot-cyberattack`, 2014.

[10] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system." `https://bitcoin.org/bitcoin.pdf`.

[11] B. Ford, "How do you know it's on the blockchain? with a skipchain." `https://bford.github.io/2017/08/01/skipchain/`.

[12] 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.

[13] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation.," in *NDSS* [13].

[14] 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.

[15] A. Back, “Hashcash - a denial of service counter-measure.” `http://www.hashcash.org/papers/hashcash.pdf`, 2002.

[16] William Pugh, “Concurrent maintenance of skip lists.” `https://drum.lib.umd.edu/handle/1903/542`, 1989.

[17] Wikipedia, “Skip list — Wikipedia, the free encyclopedia.” [Online; accessed 2-December-2018].

[18] 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.

[19] 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.

[20] 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.

[21] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Trans. Inf. Theor.*, vol. 22, pp. 644–654, Sept. 2006.

[22] Wikipedia, “Pbkdf2.” [Online; accessed 20-December-2018].

[23] J. F. Cremers, C and Mauw, Sjouke and Vink, Erik, “Dening authentication in a trace model,” 07 2004.