World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:18, No:7, 2024

Implementation of a Virtual Testbed for Secure IoT Firmware Update Using Blockchain

Tarun Chand, Michael Jurczyk

***Abstract*—**With the increasing need and popularity of IoT devices and how integrated they are becoming in our daily lives and industries; these devices make for a very lucrative target for malicious actors. And since these devices have such limited resources, the implementation of robust security features is a tradeoff to be made for the actual functionality the device was intended for. This makes them an easy target with high returns. Several frameworks for the secure firmware update of these devices have been recently proposed in the literature. They focus on methods such as blockchains and distributed file systems to secure firmware updates, but do not go into the details of the actual implementation of these frameworks and the lower-level interactions among these methods used. This work integrates some of these security measures into one overall framework and details the actual lower-level implementation of this framework in a virtual dockerized testbed running on AWS.

***Keywords*—**Blockchain, Ethereum, Geth, IPFS, secure IoT-firmware update, virtual testbed development

I. Introduction

THE rapid gain in popularity of Internet of Things (IoT) devices has introduced new challenges, particularly in managing firmware updates securely and efficiently. Traditional methods often face issues related to trust, transparency, and decentralized control, and pose vulnerabilities, such as single points of failure and susceptibility to malicious interference. In response to these challenges, several methods of securely updating IoT device firmware have been recently proposed in the literature. These approaches, among others, use blockchains for immutable record keeping, proof of authority consensus to validate and authorize firmware updates, and decentralized storage for distributed and secure storage of firmware binaries. This work combines these mechanisms to establish a decentralized and secure framework for managing firmware updates in IoT devices. A virtual testbed is then developed that implements the proposed framework, utilizing private Ethereum blockchains implemented through Geth for secure record keeping and proof of authority, and a private Interplanetary File System (IPFS) for secure firmware storage and retrieval. The testbed consists of Docker containers deployed on Amazon Web Services (AWS). The main contribution of this work is the actual lower-level implementation of a framework in a virtual dockerized testbed running on AWS. It is also shown how individual security methods interact on a lower-level to implement the overall secure framework.

Tarun Chand and Michael Jurczyk (Dr.) are with the EECS Dept., College of Engineering, University of Missouri, Columbia, MO, United States (e-mail: tcp99@missouri.edu, jurczykm@missouri.edu).

II. Background and Related Works

A. Existing Methods and Frameworks

Several methods of securely updating IoT device firmware have been recently proposed in the literature [1]–[7]. For any secure firmware update, the process needs to ensure that only the official OEM device manufacturer's firmware will be installed, and that during the firmware delivery, the firmware binary has not been tampered with. This can be accomplished using blockchains for secure record keeping of firmware updates by the manufacturer [1]–[7]. In addition, blockchains provide proof of authority consensus algorithms [1], [2], where the validation is based on the reputation and authority of participants, often employed in private or consortium blockchains for enhanced governance.

To reliably perform firmware updates, the OEM device manufacturer needs to ensure that the firmware is accessible to customers' IoT devices. Centralized firmware storage by the manufacturer could result in a single-point-of-failure or an attack point to prevent timely firmware updates [3]. The use of a distributed storage service such as Interplanetary File System (IPFS) for firmware storage is therefore proposed in [3] to guarantee reliable firmware updates.

For firmware delivery, either a push or pull operation can be used [4]. In a push operation, the source of the update pushes the updates onto the IoT devices whenever they are ready. In a pull operation, the IoT device will ping the firmware source of the update and check to see if there are any updates to be made. This could be automated or could be triggered by the user/owner and works well for devices that are constantly connected, for example home routers. Networks with many nodes, like the ones seen in large scale sensor networks or home IoT devices, wireless methods for firmware upgrades are most suitable since connecting each device via cables and wires is not exactly scalable and convenient [5], [6].

After the firmware is delivered to the end device, verification needs to be performed to check if the update was not modified by malicious actors, for example a man-in-the-middle attack. The authors of [5] do have a verification process in place and is well integrated into their system since they are using blockchain to deliver their payload in the first place. Another paper [7] proposes an architecture where there is a separate runtime that works like a pseudo-operating system to isolate the memory of the system and is responsible for the verification of the firmware installed on the device.

The authors of [2] introduce a secure firmware update

framework using the Hyperledger Fabric [8], with Hyper-ledger Composer as their UI frontend. Although the authors mention that the development was done on a Linux environment, the actual execution takes place inside a Docker container, which is a very portable implementation and easy to replicate. They have a very specific use case of developing a system for health-care, and their design reflects that.

B. Methods/Framework used for the proposed testbed

From these works, it can be concluded that there are efforts being made to incorporate blockchains into the firmware update process for IoT devices and. It is very much practical to do so as shown by the various implementations of the paper. The works of [5] and [6] come close to creating a secure system for update and verification wirelessly on a wide network, and [2] shows the implementation of a system with GDPR compliance using Hyperledger as its platform for blockchain implementation and the firmware being stored in a distributed storage as done by the authors of [3]. In our testbed, elements of these different papers have been selected and brought together as one complete end-to-end package that can be easily deployed to any system capable of running containers. Our virtual testbed utilizes private Ethereum blockchains implemented through Geth for secure record keeping, proof of authority, and firmware validation, a private Interplanetary File System (IPFS) for secure and distributed firmware storage and retrieval, and pull operation for firmware delivery. The testbed consists of Docker containers deployed on AWS.

III. Technologies Used in the Testbed

A. Blockchain

Blockchains use cryptographic signatures to link and secure a chain of records/blocks [9]. These blocks are then distributed among multiple computing systems to form a decentralized digital ledger. Each block in the blockchain contains a collection of transactions, along with a cryptographic hash of the previous block in the chain. This creates an immutable chain of blocks, where any modification to a block would require changing all subsequent blocks in the chain.

The Ethereum whitepaper [10] describes a blockchain as a state transition system. When a new transaction is submitted to the network, it is broadcast to all nodes in the network. Each node validates the transaction and adds it to a pool of unconfirmed transactions. Miners, who are nodes that perform computational work to secure the network and validate transactions, do so in accordance with the consensus mechanism dictated by the particular blockchain protocol being used. The Ethereum blockchain, for example, uses proof of stake while the Bitcoin blockchain uses proof of work.

Once the block is added to the chain, it is distributed to all nodes in the network, which validate the block's contents and cryptographic hash. If the block is valid, it is added to the local copy of the blockchain. Any nodes that do not agree with the block's contents can reject it and continue to validate transactions using the existing chain.

The use of cryptographic hashes, distributed consensus, and the consensus mechanism makes the blockchain highly resistant to tampering and hacking.

1) Ethereum

One of the existing blockchain platforms is the open-source Ethereum platform that is used, among others, for digital currency exchange and implementation of decentralized applications using smart contracts. Here is how it works according to the Ethereum whitepaper [10]:

Accounts are used to access the Ethereum blockchain. An account can either be externally owned by a person (secured by a private/public key pair), or a contract account implemented and secured through smart contracts. Each account has a unique 20-byte address, either derived from the public key of an externally owned account or from the smart contract of a contract account. For externally owned accounts, account information includes the current currency balance, and a nonce. The nonce is incremented for each transaction to ensure that each transaction is only processed once. In addition, in contract accounts, the byte-code of the smart contract and storage data is part of the account as well.

Ethereum also has its own cryptocurrency called Ether (ETH), which is used to pay for transactions on the network and incentivize validators. Ether can also be used to purchase and sell other digital assets, such as tokens that represent ownership in a decentralized application or a specific asset.

Ethereum accounts interact through "transactions". A transaction can either be the execution of a smart contract, or the transfer of some asset (such as Ether cryptocurrency) between accounts. Each transaction includes information such as sender and recipient addresses, the Ether amount to be transferred, and a "Gas Limit" (maximum resources the sender is willing to pay for) and "Gas Price" (maximum amount the sender is willing to pay for a resource). Gas Limit and Gas Price determine the price the sender is willing to pay for this transaction. After a transaction is submitted into the Ethereum network, validators validate the transaction, and once it is successfully validated, the transaction will be added to the Ethereum blockchain.

At the heart of Ethereum are smart contracts, which are self-executing contracts that are stored on the blockchain. Smart contracts are written in a high-level programming language called Solidity. They can be invoked by specific events and used to automate complex processes and transactions.

Ethereum's blockchain is composed of several key components:

- Blocks: as mentioned earlier, Ethereum's blockchain is made up of a series of blocks that contain transactions and other data. Each block is linked to the previous block in the chain, creating an immutable and tamper-proof record of all transactions on the network.
- Gas: Ethereum uses a concept called "gas" to regulate the cost and complexity of executing smart contracts. Gas is a measure of computational work, and users must pay a fee in Ether to execute smart contracts on the network. The more complex the contract, the more gas it requires, and the higher the fee.

- EVM: The Ethereum Virtual Machine (EVM) is a runtime environment that executes smart contracts on the Ethereum network. The EVM is responsible for validating transactions, executing smart contract code, and storing data on the blockchain.
- Nodes: Ethereum is a decentralized network, which means that it is run by a network of nodes, rather than a single central authority. Nodes are computers that run the Ethereum software and validate transactions on the network.

2) Geth

Geth [11] is one of the most popular clients for running a node on the Ethereum network. It is a command-line interface that enables users to interact with the Ethereum network, mine Ether, and execute smart contracts. Geth connects to the Ethereum network and downloads a copy of the blockchain. This copy of the blockchain is stored on the node running it, and it allows us to interact with the network.

In the words of Geth documentation [11], ”Geth is an Ethereum execution client meaning it handles transactions, deployment, and execution of smart contracts and contains an embedded computer known as the Ethereum Virtual Machine.”

Geth performs the following functions:

- Interacting with the network,
- Mining Ether,
- Executing smart contracts.

Geth can run in the following modes:

- Full node: This mode downloads the entire Ethereum blockchain and validates all transactions. Running a full node allows you to have a complete copy of the blockchain, which can be useful for developing decentralized applications.
- Light node: This mode downloads only the header of each block and is much faster than running a full node. However, light nodes cannot validate transactions, so they rely on other nodes on the network to do so.
- Fast sync node: This mode downloads the entire blockchain, but it does so more quickly than a full node by skipping over certain details that are not necessary for validation.

The default behavior when Geth is run is to start in Light node mode, which is what our project does as well.

3) Geth Private Network

Geth provides a number of tools and commands that allow for creation and managing of a private Ethereum network.

To create a private network, a genesis block needs to be setup, which is the first block in the new blockchain. The genesis block contains all of the initial configuration and settings for a new network, including the initial allocation of Ether, the network ID, and the consensus algorithm.

After this, Geth is used to initialize the new network and start mining blocks. Geth can also be used to add new nodes to the network, monitor the status of the network, and interact with smart contracts on the network.

To take part in the blockchain, each node needs to have an account represented by its public address and a private key. When an account is created using Geth, a password is needed to encrypt the private key and stored in a keystore, typically in the data directory along with the other blockchain data required for running a Geth node. One thing to note is, this Geth generated key pair is valid for any Ethereum network including the mainnet and other popular test networks and not just our private network.

The implementation of proof of authority consensus mechanism is done when the Geth clients are initialized and is handled by Geth itself. There is a genesis.json file that every client has to be initialized with before taking part in the blockchain network. The presence of ”clique” field in the genesis file implies that we are using the proof of authority consensus mechanism and the following field with ”extradata” field contains the list of accounts to be used as authoritative accounts that can mine new blocks.

4) Formation of Blockchain Network

The formation of the blockchain network in this project is going to peer to peer with a few nodes as authoritative Geth clients with the ability to mine new blocks. These nodes are going to be chosen at the time of deployment of the OEM server. Although the signers can be any node, the testbed is structured in such a way that the OEM manufacturer gets to decide who the signers are going to be before the blockchain is deployed. Once the genesis block is formed, the list of signers becomes immutable and any new peers that were not registered with the genesis block do not have the ability to mine new blocks.

B. Decentralized Storage

Decentralized storage is a type of data storage system that does not rely on a central server or authority. Instead, it distributes data across a network of computers, making it more secure and reliable.

In a decentralized storage system, data is broken up into smaller pieces, encrypted, and then distributed across a network of nodes, which can be computers owned by individuals or organizations. Each node stores a small portion of the data, and no single node has access to the entire dataset.

When a user wants to retrieve the data, their client software contacts multiple nodes on the network and requests the pieces of data needed to reconstruct the original file. The nodes then transmit the requested data to the user’s client, which decrypts the data and assembles it into the original file.

Since the data is distributed across multiple nodes, there is no single point of failure or attack. This makes it much more difficult for a hacker or malicious actor to compromise the data or tamper with it.

Another advantage of decentralized storage is its scalability. Since the data is distributed across multiple nodes, it can easily be scaled up or down to meet changing demand. This makes it an ideal storage solution for applications that require large amounts of data storage, such as storage of firmware for IoT devices.

There are several decentralized storage systems currently in

use, including IPFS (InterPlanetary File System) [12], Storj [13], and Sia [14].

1) IPFS

InterPlanetary File System [12] is a decentralized file storage system that uses a distributed network of nodes to store and distribute files. Unlike traditional file storage systems that rely on centralized servers, IPFS allows files to be stored and accessed from multiple nodes.

The core concept behind IPFS is content addressing, which means that files are identified by a unique hash or fingerprint rather than by their location on a specific server. This allows files to be retrieved from any node on the IPFS network, rather than from a single server.

According to IPFS documentation, this is what the life cycle of data looks like:

1. *Content-addressable representation:* this includes chunking the file and hashing each chunk.
2. *Pinning:* this involves advertising and providing the data a node has.
3. *Retrieval*: this involves content routing, block fetching from the Merkle DAG, and verification.
4. *Deleting*: this is always a local operation for a node.

Here is how IPFS works:

- *Add content*: To add content to the IPFS network, we first create a file or folder on your local computer. You then use IPFS client software to add the content to the network, which generates a unique hash that identifies the content.
- *Distribute content*: The IPFS network is made up of a distributed network of nodes, which can be computers owned by individuals or organizations. When we add content to the IPFS network, the content is automatically distributed across multiple nodes.
- *Retrieve content*: To retrieve content from the IPFS network, we use the unique hash generated when the content was added. The client software then searches the IPFS network for nodes that have a copy of the content and retrieves the pieces needed to reconstruct the original file.
- *Update content*: If we need to update a file that has already been added to the IPFS network, we need to create a new version of the file, add it to the network, and calculate the new hash for the file. We then have to change the address in the smart contract with this new hash. It is difficult to make changes and update existing content once it has been added since data on the IPFS network is immutable.

2) IPFS Private Network

It is possible to host a private IPFS network. One of the key benefits of IPFS is that it is designed to be easily deployable and customizable, allowing individuals and organizations to set up their own IPFS nodes and networks.

To host a private IPFS network, these general steps need to be taken:

- *Install and Configure IPFS*: This involves setting up storage and networking parameters, as well as configuring access control and security settings. The important one being generation of a swarm key which can be used by other nodes to join the network which we can refer to as a swarm.
- *Add content*: Once the IPFS node is up and running, we can add content to the network by using IPFS client software to create and upload files or folders. This will generate a unique hash that identifies the content on the IPFS network.
- *Distribute content*: The distribution of the content is done by the IPFS clients present in the same swarm. The uploaded file propagates through the network ready to be retrieved by referencing the content id of the file.
- *Retrieve content*: To retrieve content from the private IPFS network, we can use IPFS client to search for the content by its unique hash.

IV. Testbed Design

A. Testbed Architecture

Fig. 1 block diagram shows the overall architecture of the system used in this project. The figure examines the high-level architecture of the system, including the major components, interactions, and data flows. The IoT device firmware upgrade process using blockchain and distributed storage involves several components working together seamlessly. At the heart of the OEM network are two main servers - a web server and a Geth signer server. The web server is responsible for handling requests from various users, such as OEM manufacturers and end-users, while the Geth signer server provides the necessary infrastructure for securely storing and retrieving firmware update metadata on the blockchain.

When an OEM manufacturer wants to release a new firmware version for their devices, they use the web server to upload the updated firmware. This firmware is then stored on a distributed storage system, specifically InterPlanetary File System (IPFS), which is also running on the same server as the web server. The IPFS system assigns a unique Content ID (CID) to the firmware, which acts as its digital fingerprint. This CID is then passed on to the Geth signer server, where it is used to create a new transaction on the blockchain. The Geth signer is responsible for ensuring that only authorized parties can update the firmware on the devices. The authorized party being the OEM manufacturer that is in possession of the private key for the account mentioned in the genesis.json file. Only those accounts mentioned in the genesis.json are able to mine new blocks.

One of the key reasons IPFS is chosen instead of storing the firmware binaries directly on the blockchain is that IPFS is better suited for storing larger files. Storing large files directly on the blockchain can lead to bloat and increased storage costs. Geth is not particularly optimized to sync large blocks across the blockchain network and hence not a good choice to store large blobs of data directly on the chain. By using IPFS, we can keep the blockchain lean and focused on storing metadata, while still providing a secure and decentralized storage solution for the firmware blobs themselves.

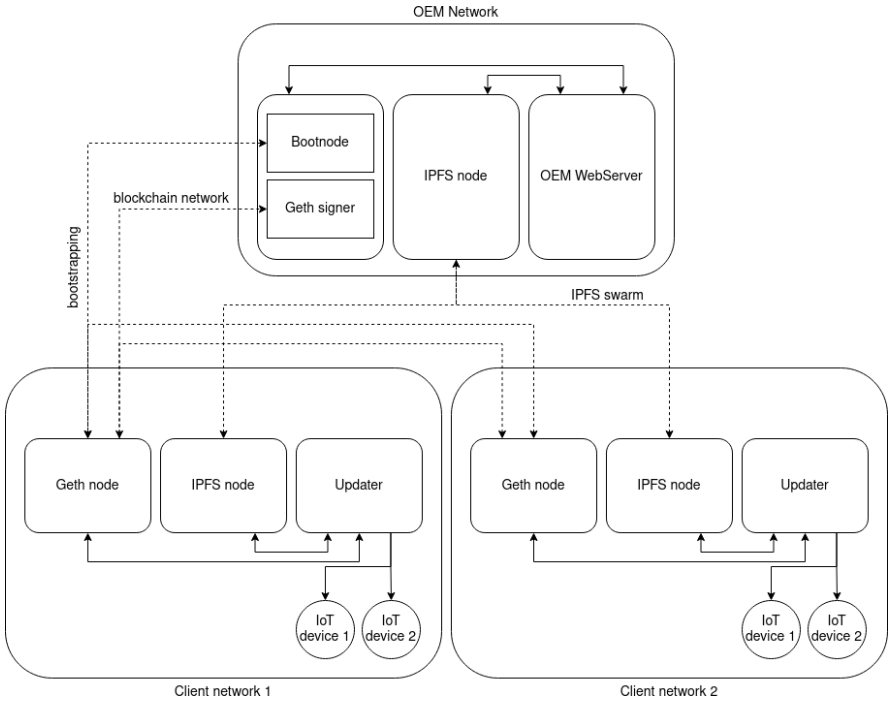

Fig. 1 Testbed Architecture

On the other side of the system, there is a second web server that runs on the end-user's updater device. This web server provides a user interface for the end-user to initiate firmware updates on their device. When the end-user requests an update, the web server communicates with the local Geth node (which is not a signer) to retrieve the latest firmware version from the IPFS node. The web server downloads the firmware from the IPFS node and applies the update to the IoT device.

Ethereum was selected as the preferred platform for implementing the blockchain, as opposed to more customizable options like Hyperledger Fabric. This decision was based on practical considerations and the specific scope of the project. The rationale for choosing Ethereum is outlined below:

- *Rapid prototyping*: Ethereum's ecosystem offers valuable tools such as the Remix IDE and Ganache blockchain. These tools enable quick prototyping of smart contracts, allowing exploration of ideas and concepts in the early phase of development.
- *Incremental development*: Ethereum supports incremental development by allowing the system to be developed in stages. Functional placeholder tools like Ganache blockchain make it possible to develop individual components separately. This is in contrast to Hyperledger Fabric, where the conventional approach often involves developing the entire system in a single iteration.
- *Availability of resources*: Ethereum is widely adopted as a development platform, resulting in a lot of resources and a healthy online community for troubleshooting.
- *Developer experience*: The use of Remix IDE in Ethereum simplifies the debugging process, contributing to a more user-friendly and efficient developer experience.

B. Data Flow

The sequence diagram in Fig. 2 shows a high-level overview of how data will flow through the system:

- The manufacturer releases a new firmware version for a device.
- The manufacturer uploads the firmware binary to IPFS and gets a CID that represents the firmware in return.
- The manufacturer records the CID of the file in the blockchain with the help of the deployed smart contract.
- An end-user initiates a firmware update on their device.
- The client's updater device contacts the blockchain to retrieve the content id of the latest firmware for a given device.
- The updater device downloads the firmware binary from IPFS using the CID recorded in the smart contract.
- The updater device installs the firmware update onto the IoT device in the local network.

C. Pseudocode

In order to provide a clear and structured understanding of the IoT device firmware upgrade system's inner workings, the following section on pseudocode is presented: Let us take a look at the pseudocode for each component, beginning with the OEM server's firmware upload process.

```
# Function to Upload Firmware

Function Upload(firmwareFile, deviceName):
    connect_to_ethereum()    # Connect to the Ethereum node
    get_account() # Get the Ethereum account address
    get_contract_abi()   # Fetch the contract ABI
    get_contract_address()     # Fetch the contract address
    connect_to_ipfs()    #Connect to the local IPFS node

    # Upload the firmware:
    ipfs_hash = upload_firmware_to_ipfs(firmwareFile)
```

```
record_transaction_on_blockchain(ipfs_hash, deviceName)
return { 'status' : transaction_status, 'hash' : ipfs_hash }
```

Note that proof of authority mechanism is being handled by the geth signer node and is configured using the genesis.json file when the blockchain is initialized.

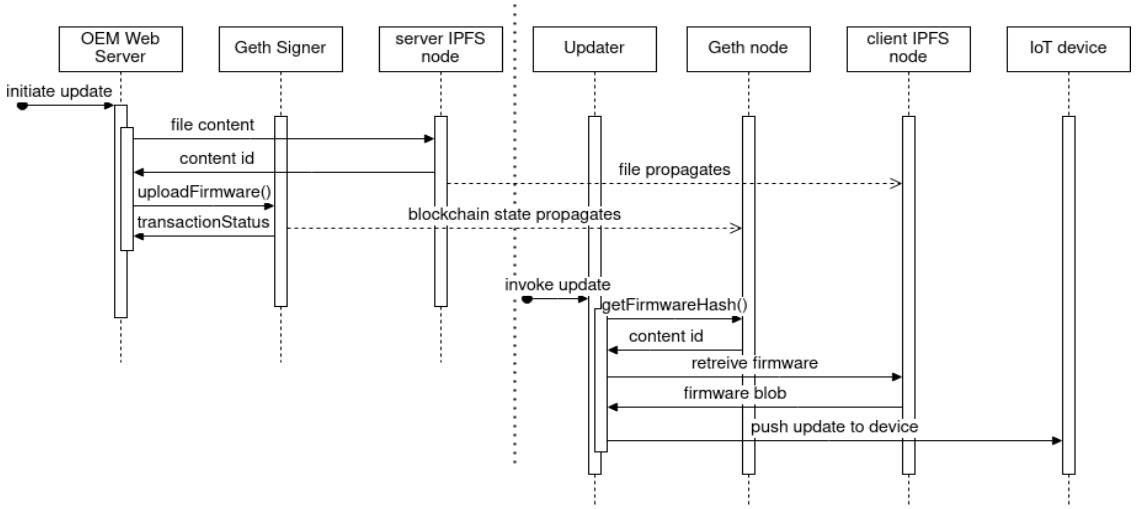

Fig. 2 Data Flow Diagram

And the following is how the end users' device would apply the firmware update from the web UI.

```
# Function to Fetch Latest Firmware

Function Update(device):
  connect_to_ethereum() # Connect to the Ethereum node
  get_contract_abi()# Fetch the contract ABI
  get_contract_address()  #Fetch the contract address

  latest_firmware_hash =
        fetch_latest_firmware_hash_from_blockchain(device)
  firmware_data =
        fetch_firmware_data_from_ipfs(latest_firmware_hash)

  apply_firmware_update(firmware_data)
```

As for the smart contract, the smart contract does the basic function of storing the latest firmware hash for each device and providing an interface for outside to store and retrieve those hashes.

```
Contract FirmwareUpdate {
    mapping(string => string) deviceToFirmwareMapping;

    function getLatestFirmwareHash (string device) view
    returns(string){
        return deviceToFirmwareMapping[device];
    }
    function storeFirmware(string ipfsHash, string device) public
    payable{
    deviceToFirmwareMapping[device] = ipfsHash;
    }
}
```

V. Testbed Development

Fig. 3 summarizes how the testbed has been structured using several Docker containers on the server and client sides:

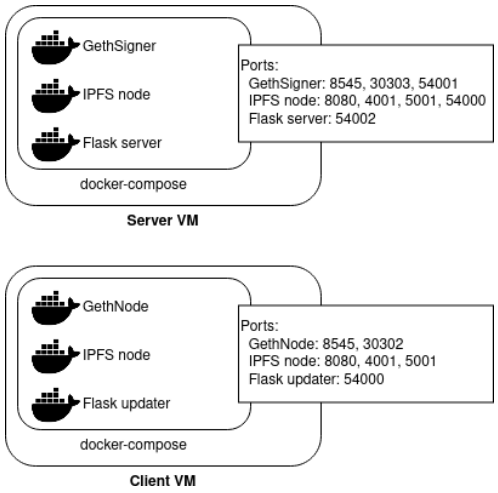

Fig. 3 Testbed structure

There is a heavy usage of Docker in this project. All the components are packaged up into Docker containers and spun up using docker-compose scripts. This is done in order to have a reproducible environment and stability in case there are version conflicts of packages being used in this project in the future.

All these Docker containers are being run in their respective Amazon Web Services EC2 virtual machines as Ubuntu 22.04 as their base operating system. The development process can be split into two distinct components: the server side and the client side.

A. Server Side

The server side of the development consists of three Docker containers:

Geth Signer: The two most important processes going to be running inside this container are the Geth client and a bootstrap node (a stripped down version of the Geth client responsible only for peer discovery). When the blockchain is initialized, it generates the genesis block, and configures the chain to follow a proof of authority consensus mechanism and specifies the account that has the authority to sign the blocks when they are mined. After it the blockchain is created and deployed, it acts as a web server serving details specific to this particular container like contract address and abi or the enode address for the bootnode. For the implementation of this web server, Flask is used since it is lightweight and fits perfectly for this kind of use case.

IPFS node: The IPFS node container initializes and sets up the IPFS node, generates a random swarm key and starts the file server. Only nodes with the swarm key are able to join this IPFS network. Finally it starts the IPFS daemon for the distributed storage. It also makes use of the Flask library to act as a web server. It serves up information about this particular container like the swarm key and the bootstrap address needed for the peer to peer connection with this particular node.

Web Server: This container serves as the interface between the blockchain, the distributed storage, and the developers of the device firmware. It provides the frontend for uploading new firmware for those devices.

B. Client Side

The client side also consists of three Docker containers:

Geth Node: The implementation for the client side Geth node that is not signing is straightforward. The entry point of this container is the creation of a new account for the user. After the account is generated, the details of the bootnode are retrieved from the server, Geth is initialized and a peer to peer connection is established with the bootnode present on the server side.

IPFS Node: The implementation of the IPFS client is also straightforward. The entry point configures the IPFS client and unlike the server side, no swarm key is generated. Instead, it reaches out to the server side of the IPFS client to retrieve the swarm key so that it can let this particular node join the swarm. Finally, the IPFS daemon is started and it becomes a part of our IPFS network.

Updater: The frontend allows the end user to select the device and check for updates. If a new update is detected, it will start the download process.

VI. Scalability and Implementation Issues

The firmware update framework is highly scalable, as both the blockchain and the IPFS file system are decentralized and distributed. In the firmware update testbed, each client is implemented using a separate AWS EC2 t3-small virtual machine running Linux. AWS pricing sets a fixed hourly price per virtual machine used. Thus, the cost running the testbed on AWS is directly proportional to the number of clients simulated. Depending on the simulation budget available, this might limit the maximum number of simulated clients.

During the testbed implementation phase, one main implementation issue was encountered. Server and Clients was run in different AWS instances with their own IP addresses. Every time the testbed is started up, the server is issued a new IP address and the clients need to find this address when starting up. It was decided, for simulation cost reasons, not to use AWS elastic IP addresses for the server, but to use an EFS file system shared among the server and clients' instances. When the server starts up, it will store its IP address in the file system, and clients can then retrieve it when they start up.

VII. Conclusion

In this paper, recently proposed methods and frameworks for the secure firmware update of IoT devices were surveyed. These methods rely on blockchains and distributed file systems to securely and reliably update IoT firmware. These mechanisms were then combined to establish a decentralized, secure, and reliable framework for these firmware updates. Then, the data flow among various components and the actual testbed architecture were defined. Finally, the lower-level implementation of this framework in a virtual dockerized testbed running on AWS was described. The testbed utilizes private Ethereum blockchains implemented through Geth for secure record keeping and proof of authority, and a private Interplanetary File System (IPFS) for secure firmware storage and retrieval.

References

[1] M. A. Uddin, A. Stranieri, I. Gondal, and V. Balasubramanian, "A survey on the adoption of blockchain in IoT: Challenges and solutions," Blockchain: Research and Applications, p. 100006, June 2021.
[2] M. Antwi, A. Adnane, F. Ahmad, R. Hussain, M. H. Rehman, and C. A. Kerrache, "The case of hyperledger fabric as a blockchain so- lution for healthcare applications," Blockchain: Research and Applications, p. 100012, March 2021.
[3] W.-J. Tsaur, J.-C. Chang, and C.-L. Chen, "A highly secure IoT firmware update mechanism using blockchain," Sensors, p. 530, 2022.
[4] R. Bielawski, R. Gaynier, D. Ma, S. Lauzon, and A. Weimerskirch, „Cybersecurity of Firmware Updates (Report No. DOT HS 812 807)," National Highway Traffic Safety Administration, Oct. 2020.
[5] X. He, S. Alqahtani, R. Gamble and M. Papa, "Securing Over-The-Air IoT Firmware Updates using Blockchain," Proceedings of the International Conference on Omni-Layer Intelligent Systems (COINS'19), Crete, Greece, pp. 164-171, 2019.
[6] N. S. Mtetwa, N. Sibeko, P. Tarwireyi and A. M. Abu-Mahfouz, "OTA Firmware Updates for LoRaWAN Using Blockchain," 2020 2nd International Multidisciplinary Information Technology and Engineering Conference (IMITEC), pp. 1-8, 2020.
[7] J. Clemens, R. Pal, and B. Sherrell, "Runtime state verification on resource-constrained platforms," MILCOM 2018 IEEE Military Communications Conference (MILCOM), pp. 1–6, 2018
[8] Hyperledger Fabric: https://www.hyperledger.org/
[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Decentralized business review, p. 21260 , 2008
[10] V. Buterin, "A next-generation smart contract and decentralized application platform," White Paper, 2014
[11] Ethereum - Geth Documentation: https://geth.ethereum.org/docs/
[12] IPFS Official Website: https://www.ipfs.com/
[13] Storj Official Website: https://www.storj.io/
[14] Sia Official Website: https://sia.tech/