Secure smart contract data sharing for IoT Devices

Julio Vega Sanchez
Supervisor: Kaitai Liang
EEMCS, Delft University of Technology, The Netherlands
j.r.vegasanchez@student.tudelft.nl, kaitai.liang@tudelft.nl

Abstract

The increasing demand for sharing Internet of Things (IoT) increases demand for a secure way of sharing data. Smart contracts could provide this because of its distributed nature. Proxy re-encryption is an encryption (PRE) method that can be used to share information. In this paper, a secure data sharing scheme is presented that uses multi-hop PRE with HyperLedger Fabric (HF). The scheme requires only two parties to participate with the data owner serving as the proxy and lets users remain access after re-encrypting the cipher. The implementation is used as means for demonstration and analysis. It still requires further work for actual deployment but demonstrates that the scheme holds in terms of efficiency and scalability.

1 Introduction

The rise of the Internet of Things (IoT) increases the need for a safe, secure and fast way of sharing data online with fellow data consumers. Furthermore, distributed systems are gaining popularity over centralized systems. These systems have the characteristic that all information is not stored on a single location but over multiple nodes. Sharing encrypted data over a distributed system comes with high computational costs. It would require the data owner to download the encrypted file, decrypt it and re-encrypt it for the receiver [4]. Using proxy re-encryption (PRE) the costs can be reduced by only letting the data owner generate a re-encryption key without needing to perform any encryption methods on the file itself.

PRE is an encryption method in which a semi-trusted party re-encrypts the delegators ciphertext to the delegatee [4]. The new ciphertext can be decrypted using the delegatee's private key. In this process, the plaintext is not revealed and the proxy does not need to know the private key of either party. This method is of great value since it gives the ability to transfer data securely without both parties needing to new each other private-public key pair. Therefore it is suitable for sharing IoT data between different parties.

Improved methods of PRE incorporate the identity of the delegator and delegatee in the encryption process, also known as identity-based (IB) PRE [13]. It is a useful technique to identify all users that participate in the encryption process in a secure manner. A more advanced version uses conditional PRE (as demonstrated in [12]) to form conditional IB-PRE [14].

Schemes have been created for IoT data sharing purposes in which data owners share their information over a blockchain network using smart contracts and PRE [16] [15]. These approaches require a third party to act as a proxy and verify all users that participate in the network. An altered version of such a scheme has been proposed by [1] by using IB-PRE to verify each user.

Another method for sharing data online combines PRE with secret sharing [9] [7]. This method does not require the need for a proxy. The data user shares his/her public key with the data owner. The data owner will then generate the re-encryption key and split the key into smaller pieces. These pieces will be shared with consensus nodes that all re-encrypt a part of the original cipher text before combining it all into one newly generated cipher text that is put on the ledger.

This paper describes a new approach for IoT data sharing by sending encrypted keys over a distributed network using smart contracts. This is done by a method that incorporates symmetric encryption and PRE with Hyperledger Fabric (HF) that is efficient and scalable. It allows the data owner to share data with multiple data users without needing to generate new key pairs.

This paper is structured as follows. Chapter 2 discusses the methodology of this research. Chapter 3 evaluates the literature on smart contracts, PRE and HF. The decentralised IoT PRE data sharing method is described in Chapter 4. In Chapter 5, the implementation of the method is discussed using code examples. The setup of the test environment and adequate results are discussed in Chapter 6. Chapter 7 reflects on the ethical aspect of the research. In Chapter 9 the research is concluded.

2 Methodology

The first step in this research consisted of gaining a better understanding of the theoretical concepts used by HF as well as being able to set up a test network and deploy a smart contract. This step is necessary as a thorough understanding of the mechanics helps in further researching other techniques that will be implemented in HF. HF provides clear documentation on critical concepts and tutorials on how to set up a

network and deploy a smart contract [11]. The test network is also used in the final prototype.

In the second step existing papers on different PRE methods were studied. Since the release of the first paper on PRE in 1998 [4] many different versions of the encryption scheme have been developed to suit different use cases. An overall understanding of the differences in properties between the methods is necessary to create a scheme that is suitable for IoT data sharing.

After finishing the literature study a PRE-scheme suitable for IoT was created that can be implemented through an HF smart contract. It was first designed on a conceptual level before being translated into an implementation. During this part the implementation also required new techniques that had not yet been examined. Therefore, the exploration and inspection of suitable continued whilst implementing the proposed PRE-scheme.

Finally, the implementation was tested on performance in terms of speed and block size. The tests were performed with a test set of files with variable sizes. The results are used to formulate a proper conclusion on the efficiency and scalability of the proposed scheme for IoT data sharing.

3 Preliminaries

In this chapter, we will discuss important concepts that will be used later on in this paper.

3.1 Smart Contract

A smart contract is an extension of classic blockchain in which programs are included in the transaction to host a virtual contract [22]. It was first introduced by Ethereum. Whereas classic blockchain only offers peer-to-peer transactions of coins [17] smart contracts offer executable programs that can be run on several peers without human interaction.

The contract consists of a set of procedural rules and logic [21]. The contract is agreed upon by all parties before it is deployed on the blockchain. The execution of the contract is automated and signed by the participating parties if verified. The executed contract is then saved on the chain.

3.2 HyperLedger Fabric

HF [5] is an open-source platform for building distributed ledger solutions created under the guardianship of the Linux foundation. It has a modular architecture to deliver a high degree of confidentiality, flexibility, resiliency, and scalability.

HF runs smart contracts as programs, called chaincode [3]. The smart contract is defined as the transaction logic and the chaincode is the packaged logic that gets deployed on the network. Chaincode can be divided into two different, namely system- and application chaincode. The first one handles system-related transactions, e.g. policy configuration, while the latter one manages application states on the ledger.

A network in HF consists of several components that are important for setting up and running the network. A schematic representation can be seen in Figure 1. We will discuss the core concepts necessary for understanding the workings of HF as described in [10].

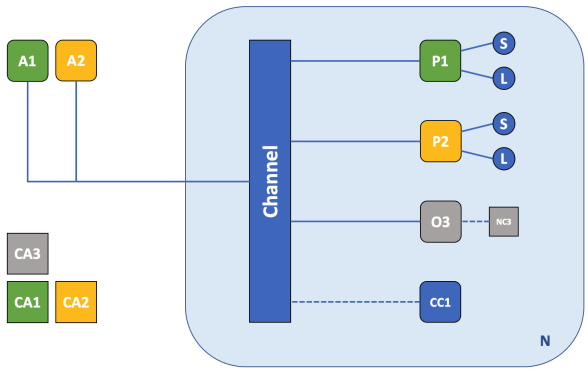

Figure 1: The HyperLedger Fabric network

In HF multiple organisations can join a network as a consortium in which policies and permissions are determined upon the creation of the network. If an organisation wants to join a channel they need to have a certificate for identification known as the certificate authority (CA1, CA2, CA3).

The network is created when the ordering service (O3) is set up. The service is started by an administrator in one of the organisations and is responsible for determining the order in which transactions take place as well as maintaining the consortia and associated policies. The administrator may allow another organisation to get administrator rights allowing them to use the ordering service. All administrator rights are held in network configuration (NC3).

In HF multiple organisations can join a network as a consortium in which policies and permissions are determined upon the creation of the network. The permissions are stored in the channel configuration (CC1). In the example, two organisations have joined the channel (the organisations with certificates (CA1 and CA2). They are connected to the channel as peer nodes (P1, P2) hosting a copy of the ledger (L) and definition of the smart contract (S) also known as chaincode. Organisations can interact with the network through applications (A1, A2) that live outside of the network. For the application to access the ledger it needs to go through the definition of the chaincode describing the common access patterns to the ledger.

3.3 Proxy re-encryption

PRE is an encryption method in which a third party (proxy) transforms the ciphertext under one key into a ciphertext under another key. The proxy does not need to know the secret keys of both parties or the original message. This method was first introduced by [4]. New variants have been created that improve the security issues of the original scheme. A study on these variants has been done by [20].

Typically any PRE scheme consists of multiple steps as illustrated in Figure 2. To illustrate the method we have the scenario where Alice wants to send a message to Bob. In the first step, Alice and Bob create their public-private key pair. Then Alice encrypts the message under her public key to generate a cipher. Bob is not able to decrypt the cipher with his key pair. Therefore, a third party known as the proxy requests the key pair of both Alice and Bob. With these pairs

he/she generates a re-encryption key that he/she will use to re-encrypt the cipher. The newly generated cipher can now be decrypted by Bob using his private key.

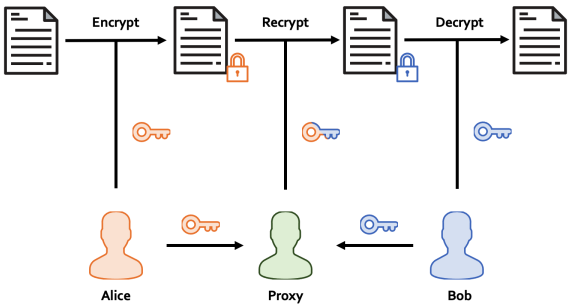

Figure 2: Schematic representation of PRE

To be able to compare different PRE schemes the properties of PRE have been described in [2] and are listed as follows:

1. *Unidirectional:* A PRE scheme is unidirectional if the proxy is only allowed to translate the delegators ciphertext into the delegates ciphertext. If the proxy may translate the ciphertext vice versa, we consider the scheme to be *bidirectional*.
2. *Non-interactive:* A non-interactive scheme require the delegatee to only share his public key pk_j for the generation of the re-encryption. The private key sk_j remains secret for all parties.
3. *Proxy invisible:* The delegatee will not be able to distinguish first-level encryption (computed under his public key) from a re-encryption of the ciphertext. In case both the delegator and delegatee do not have to be aware of the proxy we consider the scheme to be *transparant*.
4. *Original-access:* The delegatee can decrypt re-encrypted ciphertexts under his/her private key. This is also known as a *multi-hop* scheme.
5. *Key-optimal:* A users secret storage remains constant regardless of the number of delegates he/she accepts.
6. *Collusion-"safe":* In a collusion-safe PRE scheme, the proxy colluding with the delegatee can not recover the private key of the delegator. This protects the delegator against malicious proxy and participants.
7. *Temporary:* A PRE scheme with the temporary property entitles the delegator to revoke the delegated decryption rights.
8. *Non-transitive:* The decryption rights can not be re-delegated by the proxy. More formally, the proxy can not calculate $rk_{a\to c}$ from $rk_{a\to b}$ and $rk_{b\to c}$
9. *Non-transferable:* A non-transferable PRE scheme prevents the proxy to collude with delegates to re-delegate decryption rights. In other words, $rk_{a\to b}$, sk_b and pk_c can not produce $rk_{a\to c}$.

4 IoT PRE Data Sharing Method

4.1 Overview

The proposed method describes a new approach for sharing encrypted IoT data between different users. A visual representation can be seen in Figure 3. The scheme consists of four entities, namely the data owner (DO), the data users (DU) and the HF blockchain network.

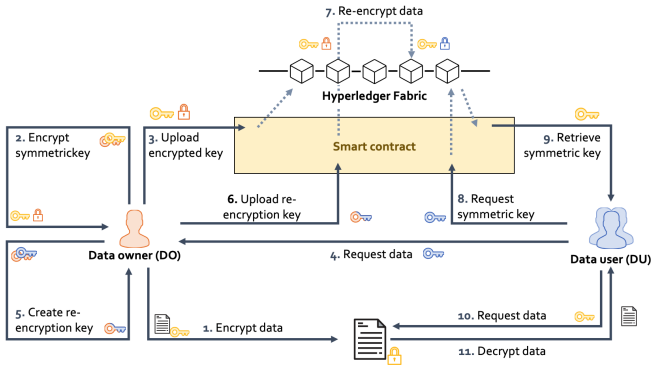

Figure 3: Schematic representation of data-sharing method

Data Owner: the DO is the entity that owns the IoT data and wants to be able to share this data with other users. He does this by encrypting his data under a symmetric key creating a cipher. The DO then generates a key pair that is used to encrypt the symmetric key under which the data will be encrypted. At this point, the DO is the only entity that can decrypt the key. The encrypted key is published to the blockchain through a smart contract for other users to access. If the DO receives a request from another user to access the data a re-encryption key can be generated using the DO's private key and the DU's public key. This way the DO serves as the proxy. The data can be re-encrypted using this key.

Date Users: The DU is the entity that wants to access the IoT data for personal use. The DU has access to the blockchain network and can read all published data containing the encrypted keys, though he can not use the data since it is encrypted. If the DU wants to access a particular piece of information he can request access to the DO. If the DO approves the DU can send his public key (as part of his asymmetric key pair) so the data can be re-encrypted. After this action, the DU can decrypt the symmetric key using his private key. The symmetric key can then be used to decrypt the cipher and generate the original data.

Hyperledger Fabric: HF is the platform that is used to host the smart contract needed for this method. All data is saved on the ledger of which a copy resides at every peer. This ensures the data can not be tampered with once it is put on the ledger. The blockchain network provides the methods to upload the symmetric key and encrypt it using the PRE scheme. Furthermore, the data can be re-encrypted by the DU and will be updated on the chain. In case someone tries to re-encrypt data with the wrong key, HF will cancel the transaction and the ledger will remain intact.

4.2 Encryption methods

The scheme combines PRE and symmetric encryption to create a scheme that is suitable for IoT sharing. For this scheme to be properly implemented it must hold several PRE properties. Firstly, it must be non-interactive so that the DO is allowed to generate a re-encryption key without the knowledge of the DU's private key (no third party needs to be involved). Secondly, the scheme must be multi-hop to ensure every user that once was authorised to decrypt the cipher can still do so after the cipher gets re-encrypted.

The IoT data sharing scheme consists of several methods that are used for encrypting and decrypting files and keys. They are assessed in chronological order:

1. **Key Generation**$(U) \rightarrow (pk_{DO}, sk_{DO}, pk_{DU}, sk_{DU})$: Two key-pairs can be generated for the DO and DU. The algorithm relies on cryptographically secure random bytes that are generated at run-time. These random bytes are used to generate the private key sk. Then, the public key pk is calculated from the private key.
2. **Data Encryption**$(Data, Key) \rightarrow C_D$: The IoT data is encrypted by the DO using symmetric encryption. This method only needs one key Key to be shared among the users to encrypt and decrypt the data. The method used in this scheme is AES [8] (with a 256-bit key). It is the most suitable method because it requires little computing power and memory, and is fast [19]. The cipher C_D can be shared among users but will only be accessible for users that have access to the key.
3. **Key Encryption**$(Key, pk_{DO}) \rightarrow C_K$: The symmetric key is encrypted using an asymmetric method under the public key of the DO. This can be seen as the first encryption step of the PRE method. The algorithm generates a cipher C_K which will be uploaded to the HF network. After the encryption, the cipher can only be decrypted using the private key of the DO sk_{DO}.
4. **Re-Encryption Key Generation**$(sk_{DO}, pk_{DU}) \rightarrow rk_{DO \rightarrow DU}$: A re-encryption key is generated using the private key of the DO and the public key of the DU that requests the data. The generated key is uploaded to the HF network for further use.
5. **Re-Encryption**$(C_K, rk_{DO \rightarrow DU}) \rightarrow C'_K$: This algorithm transforms the original cipher C_K to an altered cipher C'_K under the re-encryption key. This step allows the DU to decrypt the newly created cipher under his/her private key. The original cipher in the HF network will be updated and written into a new block.
6. **Key Decryption**$(C'_K, sk_{DU}) \rightarrow Key$: The new cipher can be decrypted by the private key of the DU sk_{DU}. This gives them access to the symmetric key to decrypt the original data. Note that the cipher C'_K can still be decrypted by sk_{DO}.
7. **Data Decryption**$(C_D, Key) \rightarrow Data$: Since the DU now has access to the symmetric key he/she can decrypt the cipher C_D to generate the original data.

5 Implementation

The implementation of the scheme is done by invoking an HF smart contract through a dedicated application. The contract is deployed on the HF test network and the application interacts with the contract through this network. The methods described earlier are partially implemented in the contract and partially performed by the application.

The prototype makes use of a proxy re-encryption package provided by IronCoreLabs called Recrypt[1]. The library is based on [6] and holds the following PRE properties: unidirectional, non-interactive, non-transitive, collusion-safe and multi-hop. The library is built in Rust but for this implementation, a Node.js binding called Recrypt Node Binding[2] is used to run the actual code.

Some encryption methods make use of randomization which is not supported by deterministic smart contracts. Therefore, the prototype requires the user to perform certain tasks locally before querying the smart contract and therefore differs from the method described in Chapter 4.

Both the chaincode and application are written in Node.js and are publically available on Github[3]. The version used in the prototype of HyperLedger Fabric is v2.4.0, the version of Recrypt is v0.13.1 and the version of Recrypt Node Binding is v0.8.1.

An overview of the tasks performed by the application and chaincode can be seen in Figure 4. The following sections will discuss them separately using the figure as a reference.

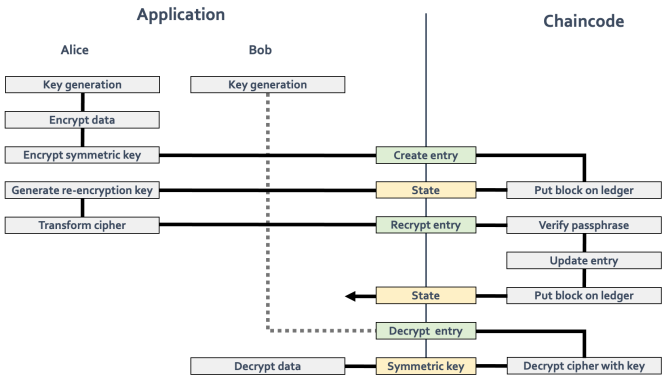

Figure 4: Communication between application and chaincode

5.1 Application

The methods that relate to key generation and encryption are performed locally through the application. This is because the methods rely on the generation of random numbers which can not be performed in the chaincode due to its deterministic nature. Therefore the application is responsible for generating keys and ciphers, whereas the chaincode is used for storing the information.

The key generation consists of generating two key pairs (one for Alice and one for Bob) and a symmetric key. The

[1] https://github.com/IronCoreLabs/recrypt-rs
[2] https://github.com/IronCoreLabs/recrypt-node-binding
[3] https://github.com/julio1998/proxyreencryption

key-pairs are generated using `Api256` provided by the Recrypt package. The symmetric key is generated using the Crypto package and has a size of 256 bits.

During the key generation, a signing key-pair suitable for EDDSA [] is also generated. This pair is required for most methods provided by the API (e.g. for signing ciphers or keys). Therefore, in future listings the signature keys are apparent. Since there is no sufficient useful implementation of these keys in the IoT PRE data sharing method (because blockchain creates a signature of each successful transaction) they will only be used as a requirement for the API.

To encrypt the data under the symmetric key (Listing 1) the fs package is used to create an input stream to the file path and write the encrypted data to a file. The algorithm first creates a `Cipher` object using the AES 256-bit algorithm in CBC mode with the symmetric key. Upon opening the input stream the algorithm creates a cipher of the data using the `Cipher` object and inserts it into a new buffer. The buffer is written to the output file. Upon closing the input stream any enciphered data is written to the output stream.

```
// Step: "Encrypt data"
function encrypt = (filePath, encryptFilePath, algorithm,
     symmetricKey, iv) => {
  let input = fs.createReadStream(filePath);
  let output = fs.createWriteStream(encryptFilePath);

  let cipher = Crypto.createCipheriv(algorithm,
     symmetricKey, iv);

  input.on('data', function(data) {
    let buf = Buffer.from(cipher.update(data, 'binary'));
    output.write(buf);
  });

  input.on('end', function() {
    try {
      let buf = Buffer.from(cipher.final('binary'), '
     binary');
      output.write(buf);
      output.end();
    } catch(e) {
      fs.unlink(output);
    }
  });
};
```

Listing 1: Encrypt data

Listing 2 displays all methods used in the application from the ”Encrypt data” step until the final step, as well as calling methods from the chaincode.

The encryption of the symmetric key is done using `Api256`. The API requires three arguments for encryption, the symmetric key to be encrypted, the public key of Alice and a private signing key. The API only allows for buffers with the size of 384 bytes to be encrypted. Therefore the symmetric key is sliced from a generated plaintext that meets the size requirements. The generated cipher is submitted as a transaction to the network.

The re-encryption key is generated using the private key of Alice and the public key of Bob. In the hypothetical scenario, Alice would be responsible for generating this key. The original cipher is then transformed using the re-encryption key and a transaction is submitted with the newly generated cipher.

In the final step of the process, Bob wants to decrypt the cipher using the retrieved symmetric key from the HF network. First, the key is sliced so that the 384-byte cipher is reduced to a 32-byte key. A similar approach as the one described in Listing 1 is used to decrypt the data. Two streams are created to read and write files. The `Cipher` object is now generated by the `CreateDecipherIV` function used to decipher encrypted data. Using the same approach Bob now has access to the original data.

```
// Step: "Encrypt data", "Encrypt symmetric key"
symmetricKey = Buffer.from(plaintext).slice(0, 32)
const encryptedKey = Api256.encrypt(plaintext, alice.
     publicKey, signingKeys.privateKey);

// Method: "Create Entry"
let result = await contract.submitTransaction('CreateEntry
     ',id, JSON.stringify(encryptedKey));

// Step: "Generate re-encryption key", "Transform cipher"
const recryptKey = Api256.generateTransformKey(alice.
     privateKey, bob.publicKey, signingKeys.privateKey);
const recryptVal = Api256.transform(encryptedKey,
     recryptKey, signingKeys.privateKey);

// Method: "Recrypt Entry"
result = await contract.submitTransaction('ReCryptEntry',
     id, JSON.stringify(recryptVal), JSON.stringify(
     signingKeys.privateKey));

// Method: "Decrypt Entry"
result = await contract.submitTransaction('DecryptEntry',
     id, JSON.stringify(bob.privateKey));

// Step: "Decrypt data"
let retrievedKey = Buffer.from(JSON.parse(result).
     plaintext.data).slice(0, 32);
await decrypt(encryptFilePath, decryptFilePath, algorithm,
      retrievedKey, iv);
```

Listing 2: Encrypt data

5.2 Chaincode

The contract saves entries with four parameters as demonstrated in Listing 3. The `ID` identifies the entry that is being saved and must be unique. The `Cipher` represents the encrypted data in the form of an `EncryptedValue` object provided by the API. It contains multiple fields necessary to be further processed by the API. The entry saves a passphrase in `Passphrase` which only allows the DO to change the state of the entry (e.g. with re-encryption).

```
const Entry = {
    ID : id,
    Cipher: cipher,
    Passphrase: passphrase
}
```

Listing 3: Entry

The DO or DU can interact with the smart contract using three methods and will be evaluated independently.

The `CreateEntry` method (Listing 4) is used to create a new entry and save it on the ledger. It accepts an identifier `id` that must be unique, the `cipher` to save and a `passPhrase` which will be used to authenticate the user if the entry were to be updated.

```
async CreateEntry(ctx, id, cipher, passphrase) {
    const exists = await this.EntryExists(ctx, id);
    if (exists) {
        throw new Error('The entry ${id} already exists');
    }

    const entry =  {
        ID: id,
        Cipher: cipher,
        Passphrase: passphrase
    };

    // Step: "Put block on ledger"
    await ctx.stub.putState(id, Buffer.from(stringify(
     entry)));

    // Return: "State"
    return JSON.stringify(entry);
}
```

Listing 4: Create entry

The `ReCryptEntry` method (Listing 5) is used to re-encrypt an existing cipher. The cipher is selected by providing the `id` after the algorithm has checked its existence. The `transformedCipher` is generated before being sent to the network using the re-encryption key $rk_{DO \rightarrow DU}$. The `passPhrase` is used to identify the user. If the DO keeps his/her passPhrase the algorithm can ensure that the DO is calling the method. If the passphrase is correct, the algorithm will update the state of the entry and push it on the ledger.

```
async ReCryptEntry(ctx, id, transformedCipher, passPhrase)
     {
    const exists = await this.EntryExists(ctx, id);
    if (!exists) {
        throw new Error(`The entry ${id} does not exist`);
    }

    const entry = JSON.parse(await ctx.stub.getState(id));

    // Step: "Verify passphrase"
    if (entry.Passphrase.normalize() === passPhrase.
     normalize()) {
        const updatedEntry = {
            ID : id,
            Cipher: transformedCipher,
            Passphrase: passPhrase
        };

        // Return: "State"
        return ctx.stub.putState(id, Buffer.from(stringify
     (updatedEntry)));
    } else {
        throw new Error(`The new cipher did not match the
     hash of ${id} or the given signature was invalid.`);
    }
}
```

Listing 5: Recrypt Entry

The `DecryptEntry` method (Listing 6) is used to decrypt the metadata. The method looks up the cipher for the given `id` and will try to decrypt it under the giving `decryptionKey` using the API. If successful, the network will return the decrypted cipher to the user.

```
async DecryptEntry(ctx, id, key) {
    const exists = await this.EntryExists(ctx, id);
    if (!exists) {
        throw new Error(`The entry ${id} does not exist`);
    }

    const Api256 = new Recrypt.Api256();

    let cipher = this.EncryptedValueFromString(JSON.parse(
     await ctx.stub.getState(id)).Cipher);

    let ret = {
        message: 'Decryption was successful',
        plaintext: ''
    };

    try {
        // Step: "Decrypt cipher with key"
        ret.plaintext = Api256.decrypt(cipher, this.
     PrivateKeyFromString(key));
    } catch (e) {
        console.log(e);
        ret.message = e.message;
    }
    // Return: "Symmetric key"
    return JSON.stringify(ret);
}
```

Listing 6: Decrypt entry

6 Evaluation

For the performance analysis, the speed of the different methods was calculated on different file sizes as well as the difference in speed when performing multi-hop encryption on the cipher.

The experiments were performed on MacOS using the Node v16.13.1 interpreter. The computer has a 3,5 GHz Dual-Core Intel Core i7 processor and 8GB of RAM.

6.1 Performance analysis

The average time for encryption and decryption of the data increases with files of bigger size as seen in Figure 5. In the test setup, encryption includes the steps of encrypting the data (in this case the file), encrypting the symmetric key and creating an entry on the HF network. The decryption only consists of decrypting an entry on the network. The increasing time relates to the AES algorithm requiring more time to encrypt and decrypt a file if its size increases.

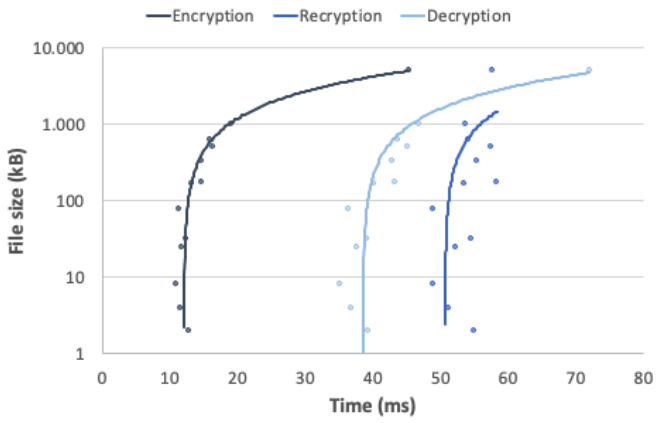

Figure 5: Speed performance of steps in implementation compared to different file sizes

Recryption does not show any significant changes in terms of speed. The recryption step includes generating the re-encryption key, transforming the cipher and recrypting the

entry on the network. The severe changes in terms of speed relate to the cipher containing only a symmetric key. The time for the algorithm to complete this step is not dependent on the file size but on a key of a fixed size of 256 bits.

The result was measured by performing an experiment 10000 times for 13 files of different sizes. The experiment consisted of the encrypting, recrypting and decrypting step in chronological order. The time per step is calculated by measuring the time before starting the first operation and after finishing the last operations. The averages per file are displayed in the graph and a trend line is calculated over the separate variables.

The decryption time and block size are related to the number of iterations in which a multi-hop re-encryption of the cipher is performed as demonstrated in Figure 6. Multi-hop re-encryption means that the cipher is re-encrypted under the private key of the delegatee and a public key of a new user (i.e. a cipher encrypted under $rk_{a\to b}$ is re-encrypted under $rk_{b\to c}$). The results can be justified by the way the cipher is generated. The cipher consists of an array of transformation blocks that adds a block every time a multi-hop re-encryption is performed, increasing the size. The decryption takes longer because every block needs to be processed.

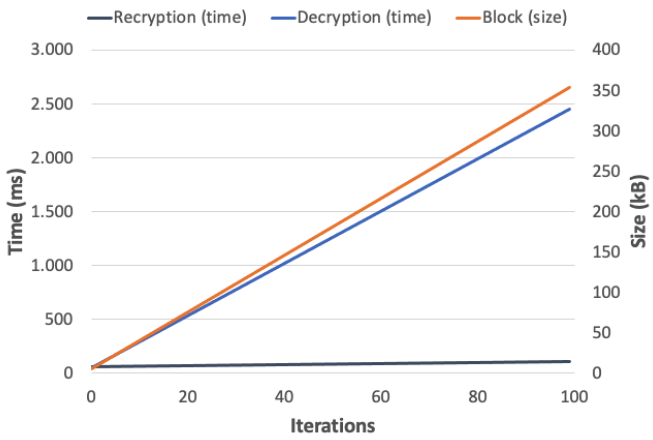

Figure 6: Speed and size performance after increasing multi-hop iterations

The experiment displays what might happen when performing multi-hop operations on the cipher. The prototype does not incorporate this method since the new cipher can not be decrypted under the original key. However, the graph does give an indication of what consequences occur in terms of speed and size if the prototype were to be refined.

6.2 Security analysis

The PRE algorithm used in the implementation has security level IND-CCA [6] which is high enough for sharing secure data with peers. The real issue lies in the security of access control for altering an entry. In the implementation, the access control is dependent on a single passphrase that has no minimum requirements in terms of size and character types. Therefore, it is easy for a malicious user to perform a brute-force attack on a single entry to be able to change the state of that entry.

7 Responsible Research

In this chapter, the ethical aspects of the research are discussed. The first section will talk about the integrity of the research followed by a discussion on the reproducibility of the implementation and results.

7.1 Integrity

In this paper, existing techniques are combined into a scheme that is suitable for IoT data sharing. During the research, many papers have been read that helped create the scheme and implementation discussed in Chapters 5 and 6. To prevent plagiarism and illegitimate ownership of intellectual property all papers that have contributed to this research have been referenced and are briefly discussed if necessary.

All code used in the implementation and test cases can be verified for legitimacy since everything is publically available on GitHub. Furthermore, cases in which a package or plug-in might reduce the speed or other performance of the prototype are discussed in the paper and need not be discovered in the code.

7.2 Reproducibility

To be able to reproduce this research several steps have been taken. Firstly, all code written for the prototype is publicly available on GitHub. Anyone can rerun the test cases in a different environment to confirm the results as displayed in Chapter 7.

Second, all plug-ins and packages that have been used in the implementation have been listed including the version number. Also, the operating system and hardware on which the tests are run are denoted. Therefore, if anyone runs the tests in the same environment they should produce similar results.

8 Conclusions

The aim of this research is to provide a secure smart contract to provide data sharing for IoT devices. The sharing method that uses PRE demonstrates the possibilities of distributing IoT securely over the HF network. Though the implementation does not fulfil the security requirements (mainly due to the insecure passphrase) the data-sharing scheme does provide on a conceptual level a secure system in which a DO can share information with other users.

The method is efficient and scalable since it remains fast and low in storage when working with larger files. The scheme is designed in such a way that file sizes do not have a significant on the speed of the network or the size of the blocks since the only information is stored in the fixed sizes symmetric key. If in a later version of this prototype a multi-hop solution is implemented, the speed and size would also not suffer since they increase linearly at a slow rate.

8.1 Future work

For future work, there are many aspects that can be improved for the prototype to be a fully functional implementation that can be deployed for public use. The first improvement is finding a more suitable package for PRE or creating the methods from scratch. The issue for the implementation is that many methods provided by the package use additional variables that do not have a suitable usage in the overall scheme. This would reduce unnecessary storage for unused variables. A PRE that has the property 'original-access' would make the prototype usable for sharing the IoT to multiple users without original users losing access. Lastly, a PRE scheme that does not increase the size of the cipher (demonstrated in [14]) when working with networks on which a high amount of peers participate.

The second improvement is to incorporate the PRE methods in the actual chaincode. Since asymmetric encryption relies on randomness this can not be implemented in a deterministic smart contract. A future version of the scheme should possibilities of using randomness in a determinstic contract so encryption can take place in the contract reducing the workload of the user and making a more coherent communication channel. This would also solve security issues since the private key of the data owner can be used as a means of identification.

The final improvement that can be explored is the use of a distributed file storage service provider such as IPFS [18]. In the prototype, files are saved locally which would mean in practice that the data owner must share the encrypted files manually with potential data users. The storage provider solves this by providing a single channel over which all DO's can share information with DU's that is easily accessible. The service can be structured in such a way that it serves as a copy of the ledger, where the service stores the encrypted files and the ledger contains all the keys.

References

[1] Kwame Opuni-Boachie Obour Agyekum, Qi Xia, Emmanuel Boateng Sifah, Christian Nii Aflah Cobblah, Hu Xia, and Jianbin Gao. A proxy re-encryption approach to secure data sharing in the internet of things based on blockchain. *IEEE Systems Journal*, 2021.

[2] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):1–30, 2006.

[3] Vipin Bharathan, Mic Bowman, and Sally et al. Cole. Hyperledger architecture, volume ii smart contracts. Technical report.

[4] Matt Blaze, Gerrit Bleumer, and Martin Straus. Divertible protocols and atomic proxy cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 127–144. Springer, 1998.

[5] Tamas Blummer, M Sean, and C Cachin. An introduction to hyperledger. Technical report, 2018.

[6] Yi Cai and Xudong Liu. A multi-use cca-secure proxy re-encryption scheme. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 39–44. IEEE, 2014.

[7] Yingwen Chen, Bowen Hu, Hujie Yu, Zhimin Duan, and Junxin Huang. A threshold proxy re-encryption scheme for secure iot data sharing based on blockchain. *Electronics*, 10(19):2359, 2021.

[8] Joan Daemen and Vincent Rijmen. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197, 2001.

[9] Michael Egorov, David Nuñez, and MacLane Wilkison. Nucypher: A proxy re-encryption network to empower privacy in decentralized systems. *NuCypher whitepaper*, 2018.

[10] HyperLedger Fabric. Blockchain network, 2020.

[11] HyperLedger Fabric. Using the fabric test network, 2020.

[12] Kai He, Xueqiao Liu, Huaqiang Yuan, Wenhong Wei, and Kaitai Liang. Hierarchical conditional proxy re-encryption: A new insight of fine-grained secure data sharing. In *International Conference on Information Security Practice and Experience*, pages 118–135. Springer, 2017.

[13] Kaitai Liang, Cheng-Kang Chu, Xiao Tan, Duncan S Wong, Chunming Tang, and Jianying Zhou. Chosen-ciphertext secure multi-hop identity-based conditional proxy re-encryption with constant-size ciphertexts. *Theoretical Computer Science*, 539:87–105, 2014.

[14] Kaitai Liang, Willy Susilo, Joseph K Liu, and Duncan S Wong. Efficient and fully cca secure conditional proxy re-encryption from hierarchical identity-based encryption. *The Computer Journal*, 58(10):2778–2792, 2015.

[15] Ahsan Manzoor, An Braeken, Salil S Kanhere, Mika Ylianttila, and Madhsanka Liyanage. Proxy re-encryption enabled secure and anonymous iot data sharing platform based on blockchain. *Journal of Network and Computer Applications*, 176:102917, 2021.

[16] Ahsan Manzoor, Madhsanka Liyanage, An Braeke, Salil S Kanhere, and Mika Ylianttila. Blockchain based proxy re-encryption scheme for secure iot data sharing. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 99–103. IEEE, 2019.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[18] Nishara Nizamuddin, Haya R Hasan, and Khaled Salah. Ipfs-blockchain-based authenticity of online publications. In *International Conference on Blockchain*, pages 199–212. Springer, 2018.

[19] Zahraa Ch Oleiwi, Wasan A Alawsi, Wisam Ch Alisawi, Ali S Alfoudi, and Liwa H Alfarhani. Overview and

performance analysis of encryption algorithms. In *Journal of Physics: Conference Series*, volume 1664, page 012051. IOP Publishing, 2020.

[20] Zhiguang Qin, Hu Xiong, Shikun Wu, and Jennifer Batamuliza. A survey of proxy re-encryption for secure data sharing in cloud computing. *IEEE Transactions on Services Computing*, 2016.

[21] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

[22] Hiroki Watanabe, Shigeru Fujimura, Atsushi Nakadaira, Yasuhiko Miyazaki, Akihito Akutsu, and Jay Kishigami. Blockchain contract: Securing a blockchain applied to smart contracts. In *2016 IEEE international conference on consumer electronics (ICCE)*, pages 467–468. IEEE, 2016.