b
UNIVERSITÄT
BERN

# Secure Causal Atomic Broadcast

## Bachelor Thesis

Noah Schmid

from

Baltschieder VS, Switzerland

Faculty of Science, University of Bern

30. June 2021

Prof. Christian Cachin
Orestis Alpos, Giorgia Marson
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

# Abstract

As used in blockchain networks today, atomic broadcast protocols do not preserve causality, which can be exploited in front-running attacks to profit arbitrage. Such attacks can be prevented when the order of the transactions is fixed before the content becomes known. Secure causal atomic broadcast protocols use threshold encryption to guarantee a causal order among the delivered messages [1]. In this thesis, we implement a threshold encryption scheme and create a secure causal atomic broadcast protocol based on the HotStuff consensus library and evaluate its performance.

# Acknowledgements

Throughout the writing of this thesis, I have received a lot of support and assistance. I would like to thank Orestis Alpos for competently supervising my thesis, brainstorming sessions, and helping to troubleshoot my prototype implementation. I also wish to thank Giorgia Marson for providing me with the necessary background material. I want to acknowledge Professor Christian Cachin for inspiring my interest in distributed systems and providing his expertise in the field from which I have benefited a lot. Last but not least, I would like to thank Olivier Staehli and my brother Jonas for proofreading this thesis.

# Contents

# Chapter 1

# Introduction

In current blockchain technology, all transactions are visible to the network for a certain amount of time before being included in a block. This concept allows for a phenomenon called front-running. In a front-running attack, an attacker spots a transaction that is not yet committed in a block and takes advantage of it by putting a new transaction in first. As an example, a client wants to buy a significant amount of token $x$. An attacker can front-run this transaction by placing his buy order of token $x$ in front of the client's order. As soon as the network places the client transaction, causing the token's price to rise, the attacker can sell the previously bought tokens for a higher price. At the time of this writing, front-running bots on the Euthereum blockchain alone have made over $750M since January 1st, 2020 [2, 3].
Front-running attacks in decentralized exchanges can be prevented when the order of transactions is determined before the transaction content becomes known. This can be done by encrypting all the transactions before submitting to the network and then decrypting them once the transaction order is fixed. Such a protocol is called "secure causal atomic broadcast." However, standard encryption schemes are not suitable for this solution, as nothing stops a single party from decrypting the transaction content before the order becomes fixed. This shows the need for a threshold encryption scheme [4] that requires a certain number of parties to collaborate to decrypt a ciphertext [1]. In this thesis, we will explore the creation of such a secure causal atomic broadcast protocol.
The two main contributions of this thesis are:

- Developing a chosen-ciphertext secure threshold encryption scheme in C++
- Integrating the threshold scheme into a consensus protocol called HotStuff

Chapter 2 introduces some basic cryptographic and distributed computing concepts needed to understand the following chapters and presents related work. We will then discuss our design requirements, the different approaches we tried, and how we integrated the threshold scheme in the consensus protocol in Chapter 3. In the following Chapter 4, we will give technical insight into the core algorithms, and in Chapter 5, explain the integration into HotStuff in detail. Finally, in Chapter 6, we will evaluate the execution time of our threshold implementation and measure how much encryption slows down the HotStuff protocol.

# Chapter 2

# Background

## 2.1 Consensus and Atomic Broadcast Protocols

In decentralized systems, it has to be ensured that all parties agree on the system's current state. This problem is referred to as the consensus problem. There are many different consensus protocols; some more widely known protocols include proof-of-work and proof-of-stake, used in the Bitcoin and Cardano networks respectively [5, 6, 7].
In an atomic broadcast protocol, all non-faulty participants in a distributed system agree on producing the same set of messages in the same order. Because all participants have to agree on the order of the messages, consensus and atomic broadcast are equivalent problems. While the protocol does make sure that all recipients receive the same sequence of messages, it does not preserve input causality which requires the system not to deliver any messages that depend in a meaningful way on a yet undelivered message sent by an honest client [1]. Secure causal atomic broadcast extends atomic broadcast such that the messages arrive in an order that maintains input causality. This can be achieved by encrypting the message to broadcast with a threshold encryption scheme for which all parties share the decryption key. After delivering the ciphertext using an atomic broadcast protocol, the parties must collaborate in an additional decryption round to recover the message from the ciphertext [1, 5, 8].

## 2.2 HotStuff

HotStuff is a consensus protocol library with the prototype implementation of the HotStuff protocol written in C++. It provides Byzantine fault-tolerant state machine replication. Byzantine faults are a condition in a distributed system where components may fail without notice. A state machine replication protocol is a way of implementing fault-tolerant services by replicating servers containing an internal state machine. For each round of consensus, one of the replicas acts as a leader. Clients can submit commands to the network, which the leader then proposes to the other replicas. The replicas vote for proposed commands over three rounds. After a majority of replicas voted for a certain command, it is delivered to the individual state machines updating the state of the replicas. All this can be done in linear time [9, 10].
HotStuff is being used by Facebook in its Diem (formerly called Libra) project [11].

## 2.3 ElGamal Encryption

ElGamal encryption is an asymmetric encryption scheme proposed by Taher Elgamal in 1985 based on the Diffie-Hellman key exchange. It can be defined over any cyclic group $G$ of prime order $q$ with generator $g$. To generate a key pair, we choose our private key $x$ randomly in $[1, q-1]$ and derive the corresponding public key $y$ by calculating $y = g^x$. The group parameters $q, g$ have to be known to all

parties. The ciphertext of a plaintext message $m \in \{0,1\}^k$ under public key $y$ is the tuple

$$(c_1, c_2) = (g^r, m \oplus H(z)),\ z = y^r$$

where $r$ is randomly chosen in $[1, q-1]$ and $H : G \to \{0,1\}^k$ is a hash function. Decryption of a ciphertext $(c_1, c_2)$ works by calculating $m' = H(c_1^x) \oplus c_2$. One can see that $m' = m$ because $c_1^x = g^{rx} = y^r = z$ [12, 13].
The original ElGamal scheme did not make use of *xor* operations with the output of a hash function. Instead, it simply multiplied the message $m$ with the random key $y^r$ and decrypted using division [12]. We will refer to our variant using H as *hashed ElGamal*.

## 2.4 Secret Sharing

Secret sharing is used to share a secret key among a group of participants such that each participant receives a different share of the secret. When enough shares are combined (more than a certain threshold $k$), the original secret key can be reconstructed. A secret sharing method is safe if $k-1$ shares reveal no information about the secret key. Secret sharing schemes allow for more secure storage of data, as there is no single point of failure. In Threshold Cryptography, a secret sharing method is needed to create the key shares. There are many secret sharing methods available, but the one we use for the scheme in this thesis is called Shamir's Secret Sharing [14]. The following part explains how this method works.
We want to split a secret key $x$ into $n$ distinct key shares such that any $k$ shares can be combined to retrieve $x$. First, we choose a random polynomial of degree $k-1$ with $x = f(0)$. The key shares $x_i, i = 1, ..., n$ are generated by evaluating the resulting polynomial in different places: $x_i = f(i), i = 1, ..., n$. To retrieve the original secret key, we use Lagrange interpolation to reconstruct $f$ from $k$ key shares and evaluate $f(0) = x$. This can be done by calculating

$$f(0) = \sum_{i \in S} x_i \lambda_{0i}^S, \text{ for } \lambda_{0i}^S = \prod_{j \in S, j \neq i} \frac{j}{j-i},$$

where $S$ is the set of share indices. Figure 2.1 shows an example with the red circle being the secret key and the blue circles being key shares. It is known that given $k+1$ distinct points, there exists a unique polynomial of degree at most $k$ interpolating those points. This means that to reconstruct a polynomial of degree $k-1$, at least $k$ points are needed and $k-1$ shares give no information about the secret [14].

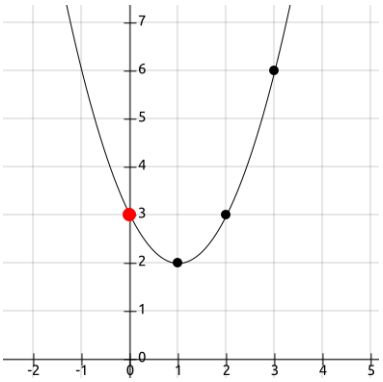


**Figure 2.1.** An example secret sharing for the polynomial $f(p) = p^2 - 2p + 3$ and private key $x = 3$. The resulting key shares are $2, 3$ and $6$.

## 2.5 Threshold Encryption

Threshold encryption schemes are based on traditional public key encryption schemes. However, instead of having one public/private key pair for each party, the private key $x$ is shared between $n$ parties using

a secret sharing method. Encryption works using the public key $y$, but $k < n$ parties have to collaborate to recover the plain text using their private key share $x_i$. Opposed to secret sharing, the private key is not reassembled during this process, and the parties never reveal their private keys. The parameter $k$ is called the threshold parameter, and the whole scheme is called a $k$ out of $n$ or short $(k, n)$ threshold encryption scheme. The decryption process works in two steps. First, each party generates a decryption share from the ciphertext using its secret share. Then $k$ such shares are combined to recover the plain text. For a threshold scheme to be secure, $k - 1$ shares should give no information about the plain text [13].
A threshold encryption scheme based on hashed ElGamal encryption would work as follows. First, the private ElGamal key $x$ is shared among $n$ parties using Shamir's Secret Sharing with a threshold of $k$. To create a decryption share from a ciphertext $(c_1, c_2)$, each party $P_i$ calculates $s_i = c_1^{x_i}$. After having received $k$ decryption shares $s_i$, one can combine the shares to reconstruct the original message $m$ using

$$m' = H(\prod_{i \in S} s_i^{\lambda_{0,i}^S}) \oplus c_2.$$

We can verify that $m' = m$ because

$$\prod_{i \in S} s_i^{\lambda_{0,i}^S} = \prod_{i \in S} c_1^{x_i \lambda_{0,i}^S} = c_1^{\sum_{i \in S} x_i \lambda_{0,i}^S} = c_1^x = y^r = z \text{ [13].}$$

## 2.6 DLIES

The Discrete Logarithm Integrated Encryption Scheme (in short: DLIES) is a hybrid encryption scheme. It is based on an asymmetric encryption scheme but uses symmetric encryption to encrypt the actual message. Again, we operate on a cyclic group $G$ with prime order $q$ and generator $g$ with $x$ being the private and $y = g^x$ the public key. When encrypting a message $m$ using the public key $y$, a random value $r \in [1, q - 1]$ is chosen and $u = g^r$ is calculated. In ElGamal, $z = y^r$ would be used for encryption. Instead, in DLIES, a new symmetric key $k = KDF(z)$ is generated using a key derivation function (KDF). The message $m$ is encrypted using an authenticated symmetric encryption scheme with key $k$, which gives us $c = ENC_{sym}(k; m)$. The ciphertext consists of $(u, c)$.
To decrypt from a ciphertext $(u, c)$ using the private key $x$, first $z$ is reconstructed using $u^x = y^{rx}$. Then $z$ is fed to the KDF to obtain the symmetric key [15].

# Chapter 3

# Design

To create a secure causal atomic broadcast protocol, a threshold encryption scheme is needed. We were searching for a chosen-ciphertext secure threshold encryption scheme written in C++ based on published work. After searching the Internet for fitting libraries, we have soon realized that none of the available solutions would fit our needs. Therefore, we decided to implement our own scheme. This chapter presents the design requirements and gives a high-level description of the threshold scheme. We also show a high-level overview of how we integrate the scheme into HotStuff.

## 3.1 Threshold Encryption Scheme

Our aim was to implement a threshold scheme that is secure against chosen-ciphertext attacks (CCA). In a chosen-ciphertext attack, an attacker gathers information by observing the decryptions of specially chosen ciphertexts. The attacker can then try to recover the secret key or any information about the plaintext out of the gathered information [16]. We started with the CCA secure *TDH2* scheme proposed by Shoup and Gennaro [17].
The main idea of that scheme is based on hashed ElGamal encryption combined with Shamir's Secret Sharing. Encryption works just as in hashed ElGamal, where first a random key $z = y^r$ is generated and fed to a hash function and then a $xor$ operation is performed between the random key $z$ and the plaintext $m$. Combining the decryption shares gives us the random key $z$ back, which we then hash and *xor* with the ciphertext to recover the plaintext. *TDH2* protects itself against chosen-ciphertext attacks using non-interactive zero-knowledge proofs that are even more complex than the encryption/decryption process itself. Those proofs ensure that the client who generated the ciphertext knows the decryption. This prohibits chosen-ciphertext attacks, as an adversary can not extract any new information from the decryption when he already knows the plaintext. Generating purposefully crafted ciphertexts to gather information would result in invalid proofs. Before creating a decryption share, the decryption algorithm first checks the zero-knowledge proofs. If they do not hold, it will not produce a share. There are also proofs to verify the validity of the decryption shares [17].
However, this scheme in its raw form does not work for bigger messages than the output of the hash function (most hash functions have an output of 64 bytes or less). Asymmetric encryption schemes are also known to be slow for big messages. Our goal was to support encrypting files of arbitrary size with our solution running as fast as possible. Therefore, we switched to a hybrid encryption scheme called DLIES [15]. The random key $z$ is fed to a key derivation function that leaves us with a symmetric key. We then encrypt the message with an authenticated symmetric encryption scheme using the derived symmetric key. The issue with this scheme is that the zero-knowledge proofs have to be calculated over the whole ciphertext (which could be very big).
The scheme we implemented encrypts the message using an authenticated symmetric encryption scheme with a random symmetric key and encrypts the key using the original *TDH2* scheme. This way, the zero-knowledge proofs only have to be calculated on the ciphertext of the symmetric key, which is relatively short (32 Bytes). The encrypted message consists of two separate parts: a decryption header

and a ciphertext. The decryption header holds all the information needed to create decryption shares and validate the zero-knowledge proofs, and the ciphertext is the actual encrypted message. This way, only the header (which is relatively short) has to be sent to the parties generating a share. The public key includes a verification key used to check the validity of decryption shares and the private key contains the public key. There are separate methods to verify the header and the decryption shares. Encryption and verification need a public key, while key generation, share generation, and share combination need private keys. There are also methods to serialize our keys with the private key being encrypted with a password after serializing.
We also wanted to be able to process messages that are too big to be held in memory. To do this, we added a way to process messages in blocks of a multiple of 8 bytes.

## 3.2 Integration into HotStuff

In the HotStuff GitHub repository that contains the prototype used in the paper, there is a demo application showing the library in action [18]. The demo consists of four replicas deployed locally on a single machine and one or more clients that send commands to the replicas. Our goal was not to change anything in the existing protocol but to extend it by building around the consensus logic to support working with encrypted commands. We wanted it to be possible to switch encryption on and off using a configuration file. Additionally, as little code as possible should be added to the demo code. Most of it should be inside the library code.
The HotStuff prototype implementation does not work with real commands/transactions but rather with clients that constantly send dummy commands. During the whole consensus part, the commands are represented by a 256-bit hash. When a command is decided, it is not delivered to a state machine as it would in a real-world scenario. Instead, a confirmation message is created (HotStuff calls it a finality) that contains the hash of the command that was decided. This message then gets sent to the client that issued the command. The client compares the hash in the finality message to an internal buffer containing hashes of all the commands it has sent. If there is a match, a message gets displayed indicating that the command was decided [18].
Our approach works in the following four steps.

1. The client first calculates the hash of the unencrypted command and stores it locally. Then it encrypts the command using the public key before sending it to the replicas.

2. Upon receiving a new encrypted command, the replicas need to store the encrypted command to be able to decrypt it later when the consensus part is finished.

3. When the command is decided, a confirmation message is created and stored. The replicas now create decryption shares and broadcast them to the other replicas.

4. As soon as enough shares have been received, the replicas search the set containing all the received encrypted commands to find the one with a matching hash and combine the shares to retrieve the original command. After the decryption is finished, they calculate the hash of the decrypted command and replace the hash in the corresponding finality. Finally, the confirmation message is sent back to the client.

The hashes of the received finality and the submitted command should now match, so that the client knows that its submitted command has been decided.
Algorithm 1 shows the pseudocode of this whole process.

**Algorithm 1** Replica command handling

1: let $shares$ be a list containing tuples of the form ($hash$, $\{share1, share2, ...\}$) indexed by $hash$
2: let $ciphertexts$ be a list containing triples of the form ($hash$, $header$, $ciphertext$) indexed by $hash$
3:
4: **upon** receiving a new encrypted command CMD **do**
5: $ciphertexts$.insert((Hash(CMD), CMD.$header$, CMD.$ciphertext$))
6:
7: **upon** having decided a command Hash(CMD) **do**
8: $c \leftarrow ciphertexts$.find(Hash(CMD))
9: $share \leftarrow$ create_share($c.header$)
10: $share.hash \leftarrow$Hash(CMD)
11: broadcast_share($share$)
12: $entry \leftarrow shares$.find(Hash(CMD))
13: **if** $entry = NIL$ **then**
14: $shares$.insert((Hash(CMD, $\{share\}$))
15: **else**
16: $entry.shares \leftarrow entry.shares \cup \{share\}$
17:
18: **upon** receiving a decryption share $share$ **do**
19: $entry \leftarrow shares$.find($share.hash$)
20: **if** $entry = NIL$ **then**
21: $shares$.insert((Hash(CMD, $\{share\}$))
22: **else**
23: $entry.shares \leftarrow entry.shares \cup \{share\}$
24: **if** $|entry.shares| > k - 1$ **then**
25: CMD $\leftarrow$ combine_shares($entry.shares$, $share.header$, $share.ciphertext$)
26: send_to_client(Hash(CMD))

# Chapter 4

# Threshold Encryption Implementation

This chapter will first take a quick look at Botan, the cryptography library we used for our implementation. We will then show how our scheme can be used and take a more detailed look at how the most important methods in the threshold scheme were implemented.

## 4.1 Botan

Botan is a C++ cryptography library released under the Simplified BSD license. It supports public key cryptography, ciphers, hashes, MACs, checksums, and other useful schemes such as key derivation functions. It also supports threshold secret sharing, but the approach Botan uses was not compatible with our design, so we had to implement our own secret sharing using Shamir's method. We tried to keep the structure and conventions of Botan in our implementation. Listing 4.1 shows a minimal ElGamal encryption/decryption demo in Botan [19].

```
std::string pt = "This is a plaintext message.";
Botan::secure_vector<uint8_t> message(pt.data(), pt.data() + pt.length());
std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);
std::unique_ptr<Botan::DL_Group> group(new Botan::DL_Group("modp/ietf/2048"));

// generate private/public keypair
Botan::ElGamal_PrivateKey private_key(*rng.get(), *group.get());
Botan::ElGamal_PublicKey public_key(private_key);

// encrypt with public key
Botan::PK_Encryptor_EME enc(public_key,*rng.get(), "EME1(SHA-256)");
std::vector<uint8_t> cipher = enc.encrypt(message,*rng.get());
std::cout << "encrypted message: " << Botan::hex_encode(cipher) << "\n\n";

// decrypt with private key
Botan::PK_Decryptor_EME dec(private_key,*rng.get(), "EME1(SHA-256)");
std::cout << "decrypted message: " << Botan::hex_encode(dec.decrypt(cipher));
```

**Listing 4.1.** ElGamal encryption demo using Botan

We can specify a custom random number generator and group to be used for decryption. The plaintext message is stored inside a `secure_vector`, which is an extension of the standard `std::vector` container. A `secure_vector` zeroes out all used bytes in memory when it is deleted.

## 4.2 Usage of our Threshold Implementation

Before dissecting the important parts of the implementation, we first demonstrate how the scheme is being used. Our goal was to make the usage as simple as possible for the end-user but still have some room for configuration. Using a three out of four threshold encryption scheme to create a similar demo as above would look like the following:

```
std::string pt = "This is a plaintext message.";
Botan::secure_vector<uint8_t> message(pt.data(), pt.data() + pt.length());
std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);
std::unique_ptr<Botan::DL_Group> group(new Botan::DL_Group("modp/ietf/2048"));
uint8_t label[20] = "this is a label"; // label can be anything

// generate private/public keypair
std::vector<TDH2::TDH2_PrivateKey> privateKeys =
    TDH2::TDH2_PrivateKey::generate_keys(3, 4, *rng.get(), *group.get());
TDH2::TDH2_PublicKey publicKey(privateKeys[0]);

// encrypt using public key
std::vector<uint8_t> header = publicKey.encrypt(message, label, *rng.get());
std::cout << "encrypted message: " << Botan::hex_encode(message) << "\n\n";

// create k decryption shares
std::vector<std::vector<uint8_t>> dec_shares;
for(int i = 0; i < 3; ++i) {
  dec_shares.push_back(privateKeys.at(i).create_share(header, *rng.get()));
}

// combine shares to recover message
privateKeys[0].combine_shares(header, dec_shares, message);
std::cout << "decrypted message: " << Botan::hex_encode(message) << "\n";
```

**Listing 4.2.** TDH2 encryption demo

Of course, it does not have to be first $k$ private keys to be used. Instead, they can be selected arbitrarily. For the sake of simplicity, we did not do this in the example above.
Instead of inputting the message as a whole, encryption/decryption can also happen in blocks of a multiple of 8 Bytes with the last block being of arbitrary size. To do this, one would use the `TDH2_Encryptor` and `TDH2_Decryptor` classes. After instantiating a new `TDH2_Encryptor` with the public key, a new encryption process can be started by calling `begin` with the chosen label which produces the decryption header. We can subsequently feed block by block to the encryptor using calls to an `update` method. The last block (which can be of arbitrary length) is fed to a `finish` method. Decryption works the same way, except that `TDH2_Decryptor` is instantiated with the private key and the shares and header instead of the label have to be supplied to the `begin` method. We will not describe these two classes in detail, as the core logic stays the same.

## 4.3 Key Generation

The whole scheme uses a cyclic group $G$ that is known to all parties. It is specified using the domain parameters $p$, $q$ and $g$. The first parameter $p$ is a large prime, such that $p - 1$ is divisible by the smaller prime $q$. Practical sizes may be a $p$ of 2048 bits and a $q$ of 256 or 512 bits. The last parameter $g$ is an element of $\mathbb{Z}_p^*$ with order $q$. Those domain parameters create a cyclic group $G$ of prime order $q$ with generator $g$ [20]. We assume that the domain parameters are encoded in $G$ to simplify the notation.
This first algorithm *KeyGen* takes as an input the desired group $G$, the threshold $k$ and the number of private keys $n$ and outputs a second generator $\bar{g}$, the public key $y$, $n$ private key shares $x_i$ and the verification key $H$ consisting of $n$ values $h_i$. In the actual implementation, a random number generator must be specified for most methods which we will leave out to simplify the algorithms.

**Algorithm 2** Key Generation

**procedure** KEYGEN($k, n, G$)
  **if** $k > n$ **then**
    **Output** 'Invalid threshold parameter'
  $\bar{g} \leftarrow$ choose random element in $G$
  $x \leftarrow$ choose random integer in $[2, q-1]$
  $y \leftarrow g^x$
  $f \leftarrow$ choose random polynomial of degree $k-1$ in $\mathbb{Z}_q$ such that $f(0) = x$
  **for** $i = 1, ..., n$ **do**
    $x_i \leftarrow f(i)$
    $h_i \leftarrow g^{x_i}$
  $H \leftarrow \{h_1, ..., h_n\}$
  $X \leftarrow \{x_1, ..., x_n\}$
  **Output** $\bar{g}, y, X, H$

We first have to ensure that the threshold $k$ is less than the total number of key shares $n$. After generating the keys and choosing a random element $\bar{g} \in G$, the secret $x$ is split into $n$ key shares using Shamir's Secret Sharing and the verification key $H$ is calculated. The group $G$ is stored inside the key objects, so from now on, we do not have to feed it as an input to the algorithms. The objects $\bar{g}$, $y$ and $H$ are stored inside the `TDH2_PublicKey` class, and the individual key shares $x_i$ are stored in instances of `TDH2_PrivateKey` which extends the `TDH2_PublicKey` class. The alternative generator $\bar{g}$ is needed for the zero-knowledge proofs and the verification key $H$ will later be used to verify the decryption shares.

## 4.4 Encryption

The encryption method is part of the `TDH2_PublicKey` class. It takes as an input a message $m$ and a label $L$ and outputs the ciphertext and a separate decryption header that contains all the information needed to generate the decryption shares. In the actual implementation, it only outputs the header and encrypts the message in place by overwriting the plaintext message. That way we can make sure the plaintext does not get leaked.

**Algorithm 3** Encryption

**procedure** ENCRYPT($m, L$)
  $k \leftarrow$ choose random symmetric key
  $c \leftarrow Enc_{Sym}(m, k)$
  $r \leftarrow$ choose random element in $[2, q-1]$
  $z \leftarrow y^r$
  $c_k \leftarrow k \oplus z$
  $s \leftarrow$ choose random element in $[2, q-1]$
  $u \leftarrow g^r$
  $w \leftarrow g^s$
  $\bar{u} \leftarrow \bar{g}^r$
  $\bar{w} \leftarrow \bar{g}^s$
  $e \leftarrow H_e(c_k, L, u, w, \bar{u}, \bar{w})$
  $f \leftarrow s + re$
  **Output** $(c_k, L, u, \bar{u}, e, f),\ c$

We first generate a new random symmetric key $k$ and use it to encrypt $m$ using an authenticated symmetric encryption scheme. In our implementation we used the scheme named "ChaCha20Poly1305".

The symmetric key $k$ is encrypted using a $xor$ operation with a random key $z = g^{rx} \in G$. Then we calculate the parameters for the zero-knowledge proofs and output the decryption header and ciphertext. The header consists of the encrypted symmetric key $c_k$, the label $L$, the value $u$ used for creating decryption shares (comparable to the value $c_1$ in ElGamal encryption) and the parameters $\bar{u}, e, f$ used for the zero-knowledge proofs. In line 11, we use a special hash function $H_e : \{0,1\}^* \times \{0,1\}^* \times G^4 \to \mathbb{Z}_q$ to calculate the value $e$. According to Shoup and Gennaro, this hash function makes the zero-knowledge proofs non-interactive [17]. It is based on a standard $b$-bit hash function $H : \{0,1\}^* \to \{0,1\}^b$. In our implementation we used SHA-256.

**Algorithm 4** Hash function

1: **procedure** $\mathrm{H}_e(m_1, m_2, g_1, g_2, g_3, g_4)$
2: $\quad h \leftarrow H(m_1) \parallel H(m_2) \parallel H(g_1) \parallel H(g_2) \parallel H(g_3) \parallel H(g_4)$
3: $\quad$ **if** $b < 2 \cdot \log_2 q$ **then**
4: $\quad\quad g \leftarrow h$
5: $\quad\quad$ **for** $i$ from 1 to $\lceil (\log_2 q - b/2)/b \rceil$ **do**
6: $\quad\quad\quad h \leftarrow h \parallel H(g \parallel i)$
7: $\quad r \leftarrow h \mod q$

It takes as inputs two binary strings $m_1$ and $m_2$ and four group elements $g_1, ..., g_4 \in G$ interpreted as binary strings and outputs a value in $\mathbb{Z}_q$. It calculates the SHA-256 hashes of the individual parameters, concatenates the outputs, and saves it in $h$. If $b$ is smaller than twice the bit size of $q$, we need to expand the hash to ensure a uniformly distributed output. As long as the bit size of $h$ is less than twice the bit size of $q$, we concatenate the current hash with an index $i$ that is incremented each round and concatenate the output with the hash of the previous round. Finally, we reduce the result modulo $q$. This procedure is a folklore method for extending the length of a hash function [21]. It makes sure that the output distribution is "as good as uniform" as described by Shoup [22].

## 4.5 Decryption Share Generation

The decryption header can now be used to generate decryption shares using the `create_share` method in the `TDH2_PrivateKey` class. Before we generate a decryption share, we first have to make sure that the ciphertext has been properly formed from a plaintext. Therefore we need a verification algorithm that verifies the zero-knowledge proofs in the header.

**Algorithm 5** Header Verification

1: **procedure** VERIFYHEADER$((c_k, L, u, \bar{u}, e, f))$
2: $\quad w \leftarrow g^f / u^e$
3: $\quad \bar{w} \leftarrow \bar{g}^f / \bar{u}^e$
4: $\quad$ **if** $e = H_e(c_k, L, u, w, \bar{u}, \bar{w})$ **then**
5: $\quad\quad$ **Output** True
6: $\quad$ **else**
7: $\quad\quad$ **Output** False

The main idea is based on a Schnorr's signature scheme with public key $u$ and private key $r$ (used in the encryption algorithm) [23]. The pair $(e, f)$ forms a Schnorr signature and proves that who created the header knows the "private key" $r$. If one knows $r$, then one can already decrypt the message and therefore know the plaintext. This means it is impossible to generate decryption headers without knowing the decrypted message, which makes chosen-ciphertext attacks impossible. There are other security measures in place, but we refer to the paper by Shoup and Gennaro for a detailed explanation [17]. The

values $g$ and $\bar{g}$ are stored inside the public key, so we do not have to pass them to the algorithm. Algorithm 6 shows how the shares are generated:

**Algorithm 6** Share Generation

1: **procedure** CREATESHARE($(c_k, L, u, \bar{u}, e, f), x_i$)
2: **if** VerifyHeader($(c, L, u, w, \bar{u}, \bar{w})$ = True **then**
3: $u_i \leftarrow u^{x_i}$
4: $s_i \leftarrow$ choose random element in $[2, q-1]$
5: $\hat{u}_i \leftarrow u^{s_i}$
6: $\hat{h}_i \leftarrow g^{s_i}$
7: $e_i \leftarrow H_{e_i}(u_i, \hat{u}_i, \hat{h}_i)$
8: $f_i \leftarrow s_i + x_i e_i$
9: **Output** $(i, 1, u_i, e_i, f_i)$
10: **else**
11: **Output** $(i, 0)$

Each share starts with an index $i$ and a flag indicating whether the header verification was successful. If it was not, it will set the flag to 0 and output the share. If the header verification was successful, it will first calculate a partial result $u_i = u^{x_i}$ and then calculate the parameters for the non-interactive proofs. The goal of those proofs is to prove that $u_i = u^{x_i}$, which is needed to ensure that the share combination yields a useful result. In line 7, we need another hash function $H_{e_i} : G^3 \rightarrow \mathbb{Z}_q$ that is implemented analog to $H_e$. The difference is that it only takes three input parameters.

## 4.6 Share Combination

Before combining the shares, we want to make sure that the header is valid and each of the shares is correct. The share verification can be done as follows:

**Algorithm 7** Share Verification

1: **procedure** VERIFYSHARE($s_i, (c_k, L, u, \bar{u}, e, f), H$)
2: Parse $s_i$ as $(i, flag, \cdots)$
3: **if** $flag = 0$ **then**
4: **Output** False
5: Parse $s_i$ as $(i, flag, u_i, e_i, f_i)$
6: $\hat{u}_i \leftarrow u^{f_i} / u_i^{e_i}$
7: $\hat{h}_i \leftarrow g^{f_i} / h_i^{e_i}$
8: **if** $e_i = H_{e_i}(u_i, \hat{u}_i, \hat{h}_i)$ **then**
9: **Output** True
10: **else**
11: **Output** False

This method looks almost identical to the header verification method except that it first checks whether the flag inside the share is set to zero. Finally, if we have enough valid shares, we can combine the decryption shares to reconstruct the symmetric key. The method `combine_shares` inside the `TDH2_PrivateKey` class is the biggest method of them all.

**Algorithm 8** Share Combination

**procedure** COMBINESHARES($S, (c_k, L, u, \bar{u}, e, f), c$)
  **if** $k > |S|$ **then**
    **Output** 'not enough shares to reconstruct message'
  **if** VerifyHeader($(c_k, L, u, \bar{u}, e, f)$) = $False$ **then**
    **Output** 'invalid header'
  **for** $s_i \in S$ **do**
    **if** VerifyShare($s_i, (c_k, L, u, \bar{u}, e, f)$) = $False$ **then**
      **Output** 'invalid share'
  $z \leftarrow 1$
  **for** $s_i \in S$ **do**
    $l \leftarrow \prod_{j \in S, j \neq i} \frac{j}{j-i}$
    $z \leftarrow u_i^l \cdot z$
  $k \leftarrow c_k \oplus z$
  $m \leftarrow Dec_{Sym}(c, k)$
  **Output** $m$

It first checks whether the decryption header is valid. If it is, it will then verify each share, and only if all of them are valid will it then combine the shares using Lagrange interpolation in the exponent to retrieve the random key $z$. Then, a $xor$ operation between $z$ and the ciphertext $c$ is performed to retrieve the symmetric key $k$. Now this key $k$ can be used to decrypt the ciphertext $c$ using the authenticated symmetric encryption scheme "ChaCha20Poly1305". If the decryption fails because of an integrity violation, an empty string is returned from the function.

# Chapter 5

# Integration into HotStuff

This chapter will examine how we extended the HotStuff library [18] to support a secure causal atomic broadcast protocol. We will only focus on the core logic and show as little code as possible.

## 5.1 Configuration File

HotStuff provides separate configuration files for the client and each replica to distribute the keys and set some global settings. The client configuration file `hotstuff.conf` specifies, among other things, the addresses of the replicas and their public keys. We added two new values to this file. The value `use-tdh2` determines whether the commands should be encrypted and `tdh2-public-key` specifies the *TDH2* public key. The files `hotstuff-sec[id].conf` (where `[id]` is a value from 0 to $n-1$) contain the ids and private keys of the corresponding replicas. We added a value `tdh2-privkey` holding the encoded *TDH2* private key and a value `tdh2-pwd` specifying the password used to decrypt the private key. The keys were generated outside the HotStuff code using a Python script.

## 5.2 Client Modifications

First of all, we need to modify the client to send encrypted commands. To do this, we looked at the source code of the demo client (located in the file `hotstuff_client.cpp`). The `try_send` method in the file creates a `CommandDummy` object, which is a placeholder for a real command. This object contains a command identifier `cid` (which remains static for the demo) and a counter `n` that is incremented for each new command. The method `try_send` then calculates the hash of the newly created command, wraps the object inside a `MsgEncReqCmd` message, and sends it to the replicas. The hash, as well as the command, is stored locally inside the `waiting` map as the client waits for a confirmation. The replicas will later send back a confirmation containing the hash of the command that was decided. The client then searches for the received hash in the map and displays a message if there is a match.
Our first approach was to modify the `serialize` method of the `Command` base class (which every command such as CommandDummy extends) such that it encrypts the data after serialization. However, this leaves us with a few problems. First of all, serializing an object only makes sense if we can deserialize again from serialized data. Changing the serialization process such that it also encrypts the data would make deserialization impossible. We would also need to pass the public key to each created command, which contradicts responsibility-driven design principles. By creating a new command type that holds the decryption header and ciphertext, we can circumvent both problems. We are still able to implement new command types in the future without having to worry about encryption.
After calculating the hash of the `CommandDummy` object, we now serialize the command and encrypt the result using the *TDH2* public key. We created a new command class `EncryptedCommand` that holds the decryption header and ciphertext and a new message class `MsgEncReqCmd` that holds the `EncryptedCommand` object. After sending the `MsgEncReqCmd` message, we store the hash of the

unencrypted command together with the `CommandDummy` object in the `waiting` map. Listing 5.1 shows that all of this can be done in just a few lines of code.

```
/* ... */

auto cmd = new CommandDummy(cid, cnt++);
auto hash = cmd->get_hash();

// serialize command and store inside a secure vector
std::vector<uint8_t> cmd_data(cmd->to_bytes());
Botan::secure_vector<uint8_t> data(cmd_data.data(),
    cmd_data.data() + cmd_data.size());

uint8_t label[20] = "this is a label"; // placeholder, could be anything
std::vector<uint8_t> header =
    tdh2_public_key->encrypt(data, label, *rng.get());

auto enc_cmd = new hotstuff::EncryptedCommand(header, Botan::unlock(data));
MsgEncReqCmd msg(*enc_cmd);

// send message to all replicas
for (auto &p: conns) mn->send_msg(msg, p.second);

/* ... */

waiting.insert(std::make_pair(hash, Request(cmd)));
```

**Listing 5.1.** *try_send* method implementation

## 5.3 Replica Modifications

The goal was again to change as little as possible in the replica code and move the decryption part to the parent class (inside `hoststuff.cpp`). We tried mirroring the original message flow by creating new versions of existing methods. In the demo replica code (`hoststuff_app.cpp`), there is a handler (`client_request_cmd_handler`) that is called when the replica receives a new command. Inside this handler, the received command is deserialized and added to a waiting list for consensus using the function `exec_command`, which takes as an additional input a callback function. This function is executed as soon as the command is decided.
We created a new handler `client_request_enc_cmd_handler` that is called every time a `MsgEncReqCmd` message containing an encrypted command is received. This new handler is similar to the existing one. The biggest difference is that we do not use the method `exec_command`, instead we use a new method `exec_encrypted_command`. This method takes as inputs the encrypted command, the address of the client and a callback that is executed once the command is decrypted. In the callback, we simply add the resulting confirmation message to a response queue. The replica will then send the response messages inside the queue back to the client on a different thread.
The method `exec_encrypted_command` is implemented as follows:

```
const auto &cmd_hash = cmd->get_hash();
encrypted.insert(std::make_pair(cmd_hash, cmd));
decrypted_callback.insert(std::make_pair(cmd_hash, callback));

cmd_pending.enqueue(std::make_pair(cmd_hash, [this, addr](Finality fin) {
    auto it = encrypted.find(fin.cmd_hash);
    if(it != encrypted.end()) {
        EncryptedCommand cmd(*it->second);
        std::vector<uint8_t> share =
            tdh2_priv_key->create_share(cmd.header, *rng.get());
        hotstuff::DecryptShare s(share, fin);
        do_broadcast_share(s);
}));
```

**Listing 5.2.** *exec_encrypted_command* implementation

The first thing it does is store the encrypted command along with its hash inside the unordered map `encrypted`. We also store the callback to be able to call it once we are finished inside the map `decrypted_callback`. We then add the hash of the encrypted command to the `cmd_pending` queue, which triggers the consensus logic. We pass in a function that handles the share generation and broadcasts the share. This function is called once the consensus round is finished and the command is decided. The decryption share object `s` also stores the finality `fin`. The method `do_broadcast_share` sends the share to all other replicas and stores the share `s` inside the map `shares`.

Now, decryption shares are created and sent after consensus for a command is reached, but nothing happens once a decryption share is received. We have to register another handler that is called when a replica receives decryption shares.
Once we receive a new decryption share, we search through the map `shares` to find existing shares for a given command. This map consists of a hash and a vector of shares. If we find no matching entry, we add a new one holding only the just received share. Else we add the received share to the list of shares with a matching hash. As soon as we have received enough shares to decrypt, we will start the decryption process by calling the `combine_shares` method of the replica's private key. We then calculate the hash of the decrypted command and replace the hash in the corresponding finality. Now we search for the matching callback function inside `decrypted_callback` that was registered inside `exec_encrypted_command` when the encrypted command was received. If we find it, we the function and pass in the finality with the updated hash which adds the message to the response queue.

# Chapter 6

# Validation

In this chapter, we first look at the performance of the threshold encryption implementation and examine how changing the different parameters impacts the execution time. We will subsequently look at how much the threshold scheme slows down the HotStuff protocol. All benchmarking was done on a Lenovo ThinkPad L390 Yoga with an Intel i7-8565U CPU and 32GB of memory running Ubuntu 20.04.2 LTS.

## 6.1 Standalone Threshold Encryption Benchmark

We tested encrypting files of different sizes using threshold encryption with three different groups: All groups had a modulus $p$ with a bit size of 2048, but different orders $q$. We used two randomly generated groups with a $q$ of 256 and 512 bits and the 2048 bit Diffie-Hellman MODP group defined in RFC3526 with a $q$ of 2047 bit which was already implemented in Botan [24]. We used 100 key shares with a threshold of 67. For the share generation benchmark, we looked at the average time it took to generate one decryption share over all 67 shares. Some of the results are shown in the two tables below (in ms).

| operation | 256 bit $q$ | 512 bit $q$ | MODP |
|---|---|---|---|
| **key generation** | 308 | 504 | 2546 |
| **encryption** | 2 | 4 | 24 |
| **share generation** | 6 | 11.9 | 52.2 |
| **decryption** | 411 | 803 | 5484 |

**Table 6.1.** Time for the different *TDH2* operations on a file of 1 kB (times in ms)

| operation | 256 bit $q$ | 512 bit $q$ | MODP |
|---|---|---|---|
| **key generation** | 329 | 498 | 2706 |
| **encryption** | 346 | 330 | 374 |
| **share generation** | 5.9 | 10.5 | 48.1 |
| **decryption** | 726 | 1098 | 6018 |

**Table 6.2.** Time for the different *TDH2* operations on a file of 320 MB (times in ms)

A few things can be observed from the provided results. A bigger file size did not seem to slow down the share generation process. This was to be expected, as the share generation only uses the header and not the message itself. The file size also did not make that much of an impact on the encryption and decryption run time. What seemed to be the most significant factor is the size of the parameter $q$. Going from a group with a 256 bit $q$ to the MODP group with a $q$ of 2047 bit roughly slowed decryption down by a factor of ten.

The decryption process took the most time to complete because it needs to verify the header and shares first before decrypting. This is where most computing time is spent because the modular exponentiation operations used for the zero-knowledge proofs are pretty costly in practice, and each share has to be verified separately.
We also wanted to find out how much different values for the threshold parameter $k$ would impact the performance of our scheme. Therefore we ran the same tests as above but with a threshold of seven instead of 67.

| operation | 256 bit $q$ | 512 bit $q$ | MODP |
|---|---|---|---|
| **key generation** | 125 | 174 | 543 |
| **encryption** | 2 | 4 | 22 |
| **share generation** | 5.3 | 9.6 | 42.3 |
| **decryption** | 47 | 69 | 232 |

**Table 6.3.** Time for the different *TDH2* operations on a file of 1 kB with $k = 7$ (times in ms)

| operation | 256 bit $q$ | 512 bit $q$ | MODP |
|---|---|---|---|
| **key generation** | 132 | 184 | 595 |
| **encryption** | 330 | 330 | 351 |
| **share generation** | 5.1 | 9.7 | 44.1 |
| **decryption** | 374 | 389 | 576 |

**Table 6.4.** Time for the different *TDH2* operations on a file of 320 MB with $k = 7$ (times in ms)

Decreasing the threshold by a factor of ten greatly reduced the decryption time - for example using the MODP group with a file size of 420 MB even by a factor of ten. Key generation also ran a lot faster as the degree of the polynomials for the secret sharing process decreased, which made evaluating the polynomials faster.

## 6.2 HotStuff Integration Benchmark

The HotStuff demo client application displays the elapsed time between sending a command and getting the finality message for that particular command. We tested running four replicas and a client on different processes on a single machine, so there were no network delays. Before adding anything to the library, the average elapsed time was 4 ms. After adding threshold encryption with $n = 4, k = 3$, a $p$ of 2048 bit and a $q$ of 256 bit to the protocol, the average elapsed time was 201.2 ms. This seems like a heavy increase in time compared to the original run time, but as previously stated, this does not take into account message transmission delays. If we were to test it in a more realistic scenario, it would probably cause a much smaller relative time increase. As such a deployment was out of scope for this thesis, we will leave it up to further work.
However, the first command sent by the client usually had an elapsed time of around 60 ms. All received decryption shares are processed on the same thread, which means newly decided commands have to wait until the last decided command has been decrypted.

# Chapter 7

# Conclusion

In this thesis, we have implemented a chosen-ciphertext secure threshold encryption scheme and integrated it into HotStuff to create a secure causal atomic broadcast protocol. We tested our threshold scheme in isolation, which gave us reasonably fast results for domain parameters $p$ of 2048 bit and $q$ of 256 bit using a small threshold $k = 7$. Encrypting a 320 MB file took 330 ms, creating a decryption share 5.1 ms on average, and combining the shares to retrieve the original file took 374 ms. Big file sizes did not seem to impact the execution time too much. However, increasing the bit size of $q$ from the underlying group or increasing the threshold parameter $k$ caused a significant increase in execution time.

We integrated our scheme into HotStuff and measured the overhead it introduced. On a deployment where four replicas run on different processes on a single machine, integrating the scheme into HotStuff increased the average elapsed time between submitting a command and receiving the confirmation from 4 ms to 201.2 ms. This testing scenario is of course not very realistic, as the network transmission delays in a real deployment would cause a much smaller relative time increase caused by the threshold encryption scheme.

The implemented protocol is just a basic prototype and has much room for optimizations. Combining the decryption shares on a different thread would surely speed up the application a lot, as the first command sent by the client usually had an elapsed time of around 60 ms in our tests. At the moment, commands can not be decrypted in parallel, which causes them to wait for a free decryption slot.
Further work may include measuring the performance in a more realistic setting, allowing the commands to be decrypted in parallel (on separate threads) to improve performance, and adding verification logic after decrypting the command to see whether the decrypted command is valid. Also, the *TDH2* scheme in its current form does not scale well for large thresholds. Further research could be done to find an alternative way to verify the validity of decryption shares, which is currently the implemented scheme's bottleneck.

# Bibliography


[1] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 524–541, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[2] Dr. Raffael Huber. Arbitrage and frontrunning in defi. Bitcoinsuisse, available online, `https://www.bitcoinsuisse.com/research/decrypt/arbitrage-and-frontrunning-in-defi`, 2021.

[3] Flashbots. MEV Explore. Flashbots, available online, `https://explore.flashbots.net/`, 2021.

[4] Yvo G. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[5] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. `http://bitcoin.org/bitcoin.pdf`.

[7] Cardano Foundation. Cardano — Home. Cardano , available online, `https://cardano.org/`, 2021.

[8] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3):986–1009, 1994.

[9] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain, 2019.

[10] Michael Reiter and Ken Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16:986–1009, 05 1994.

[11] Contributor Network. A technical perspective on Facebook's LibraBFT Consensus algorithm. The Block, available online, `https://www.theblockcrypto.com/post/28194/a-technical-perspective-on-facebooks-librabft-consensus-algorithm`, 2019.

[12] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, page 10–18, Berlin, Heidelberg, 1985. Springer-Verlag.

[13] Jovana Mićić Christian Cachin. Distributed cryptography. Whitepaper, 2020.

[14] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[15] Wikipedia contributors. Integrated Encryption Scheme - Wikipedia. Wikipedia, available online, `https://en.wikipedia.org/wiki/Integrated_Encryption_Scheme`.

[16] Wikipedia contributors. Chosen-ciphertext attack - Wikipedia. Wikipedia, available online, https://en.wikipedia.org/wiki/Chosen-ciphertext_attack.

[17] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In Kaisa Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 1–16, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[18] hot-stuff. hot-stuff/libhotstuff: A general-purpose BFT state machine replication library with modularity and simplicity, suitable for building hybrid consensus cryptocurrencies. Available online, https://github.com/hot-stuff/libhotstuff.

[19] Jack Lloyd. Botan: Crypto and TLS for Modern C++. Available online, https://botan.randombit.net/.

[20] Nigel P. Smart. *Cryptography Made Simple*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[21] fgrieu. How to realize a hash function H:0,1* × G × 0,1* -¿ Zq - Cryptography Stack Exchange. StackExchange, available online, https://crypto.stackexchange.com/questions/83775/how-to-realize-a-hash-function-h0-1-%C3%97-g-%C3%97-0-1-zq.

[22] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, USA, 2005.

[23] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[24] M. Kojo T. Kivinen. rfc3526. Available online, https://datatracker.ietf.org/doc/html/rfc3526.