Wrocław University of Science and Technology

Faculty of Information and Communication Technology

Field of study: **Algorithmic Computer Science (INA)**

Speciality: **Cryptography and Computer Security (CCS)**

Master Thesis

Selected cryptographic schemes implemented in the blockchain system

Karolina Bak

Thesis supervisor

DSc, PhD, Eng. Łukasz Krzywiecki

Keywords: blockchain, cryptography, digital signatures, smart contracts, distributed applications

Wrocław 2023

Abstract

During the last decade, we have observed the rise of the blockchain technology from simple distributed ledgers to powerful programmable networks. This master thesis aims to explore the computational capabilities of these systems to implement selected cryptographic schemes. The work covers the design and implementation of a distributed timestamping system based on a protocol combining Schnorr signatures and Pedersen commitments in which data is stored and validated using Ethereum network. The proposed system utilizes a smart contract to store protocol data and mediate the flow of information between server nodes which issue timestamps and client nodes who can submit data for timestamping and call the verification procedure on any recorded timestamp. We evaluate the efficiency of the implemented protocol based on gas fees, memory usage and execution time. The work discusses the advantages and limitations of Ethereum network observed during the development of the system, such as the public storage which on one hand provides full transparency and on the other prevents us from keeping any secret on the network.

Contents

List of Figures

Introduction

The main objective of this master thesis is to explore the potential of blockchain networks for implementing cryptographic schemes. The work covers the design and implementation of a system that issues and verifies attestations for submitted data where the attestations are in a form of a digital signature. The functional requirements for this system include:

- full or partial implementation of a cryptographic scheme in a smart contract;
- utilization of a blockchain network to distribute information.

The first chapter (1) is an introduction to the blockchain networks that focuses on the two popular systems, Bitcoin and Ethereum. The chapter explains the fundamental concepts necessary to understand this technology such as transactions or blocks and describes in detail the consensus mechanisms used in those networks.

The second chapter (2) thoroughly describes the protocol that will be adapted and implemented in the blockchain environment. The chapter presents the schemes and mathematical problems that are a part of the protocol. All stages of the protocol are closely examined and visualized using pseudocode and illustrations.

The third chapter (3) focuses on the design of a signature-based system. It features the description of the main parts of the system and illustrates its architecture. The chapter also presents the implementation details such as used programming languages or libraries and instructs how to set up the system.

The fourth chapter (4) presents the results of the efficiency tests performed on the implemented protocol. It focuses on the gas usage of the smart contract, the time needed to generate a single timestamp and the memory usage of the protocol during and after its execution.

The last chapter (5) discusses advantages and limitations of cryptography in blockchain environment observed during implementation and the tests. The chapter also includes ideas for further improvements of the proposed system.

Chapter 1

Blockchain Technology

This chapter introduces the concept of a blockchain through the description of the inner workings of the two most popular systems: Bitcoin, the pioneer of the blockchain technology, and Ethereum, a platform for building decentralized applications and organizations.

1.1 Bitcoin

The *blockchain* has been conceptualized by Satoshi Nakamoto in a Bitcoin whitepaper [6] that was published in October 2008. The main reason behind this system is to provide a way to make online payments that go directly from one party to another without the involvement of a financial institution. The author (or authors as the identity of Nakamoto is unknown) points out the flaws in the current centralized trust-based payment systems and expresses the need for a distributed cryptography-based ones that will protect the users from fraud - especially double-spending problem - by making the transactions computationally infeasible to reverse. The system proposed by Nakamoto is a peer-to-peer network that timestamps transactions by hashing them into an ongoing chain secured by a proof of work mechanism that prevents the changes to previous data without redoing the proof. The longest chain serves as a public ledger showcasing the sequence of transactions accepted by the majority of the computational power in the network. The next subsections describe the data structures present in the blockchain system, the consensus mechanism and how the currency supply is managed in more detail.

1.1.1 Transactions

There is no designated model to represent a coin in the Bitcoin blockchain. All information about the ownership of the currency is stored in the *transactions*, a structure that can be likened to a material forming the links in the chain. The transactions consist of two main parts:

- a list of `inputs`, which contain the hashes of previous transactions that provide the currency to distribute, the indexes of specific outputs from where the currency is taken, and the signature for the verification script;
- a list of `outputs`, which specify how the inputs are to be distributed by stating the exact amount of currency to transfer and providing the verification script that allows anyone who matches the criteria to claim it.

The output that has not yet been claimed by anyone is called *unspent transaction output* (**UXTO**). Only UXTOs can be used as an input to the transaction to prevent the double

spending. The network will automatically reject any transaction that violates this principle. The inputs and the outputs must contain the same amount of Bitcoin, otherwise the difference can be collected by the person that successfully writes the transaction into the blockchain. To avoid losing money, a user can send the extra currency to themselves and use it later in another transaction.

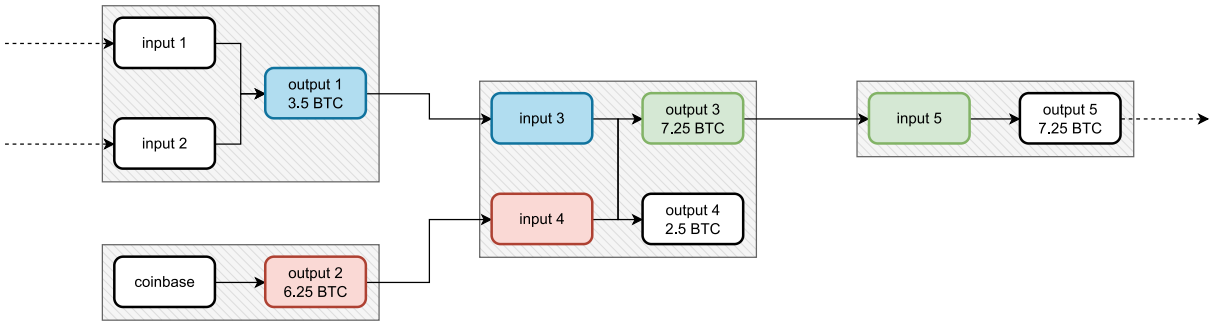

Figure 1.1: The transactions in Bitcoin

1.1.2 Blocks

Since there is no central trusted authority that checks the validity of transactions in Bitcoin, the users of the network are responsible for verifying them. To achieve that, all new transactions are gathered into a structure called the *block* which is akin to a link in the chain. The first block in the blockchain is called the *genesis block*. In Bitcoin genesis block consisted of a transaction awarding 50 BTC to the creator with the headline of The Times from 3rd January 2009 encoded in it. The block consists of the *header* and the list of transactions that are a part of it. The header is the core part which includes all of the necessary information for connecting the blocks and protecting the data inside from manipulation:

- a hash of the previous block's header;
- the root of the Merkle tree of all transactions in the block;
- the `timestamp` containing the time of creation;
- the `target` value shared by the network's users determining the difficulty of creating the block;
- a `nonce` value that proves the validity of the block.

Including the hash of the previous block in the subsequent block prevents data manipulation in the old records, because any change to the previous blocks would destroy the blockchain structure. Additionally, each block is protected by a security mechanism that makes it hard to create blocks in order to prevent the attacker from recreating a part of the chain to gain advantage over the honest users.

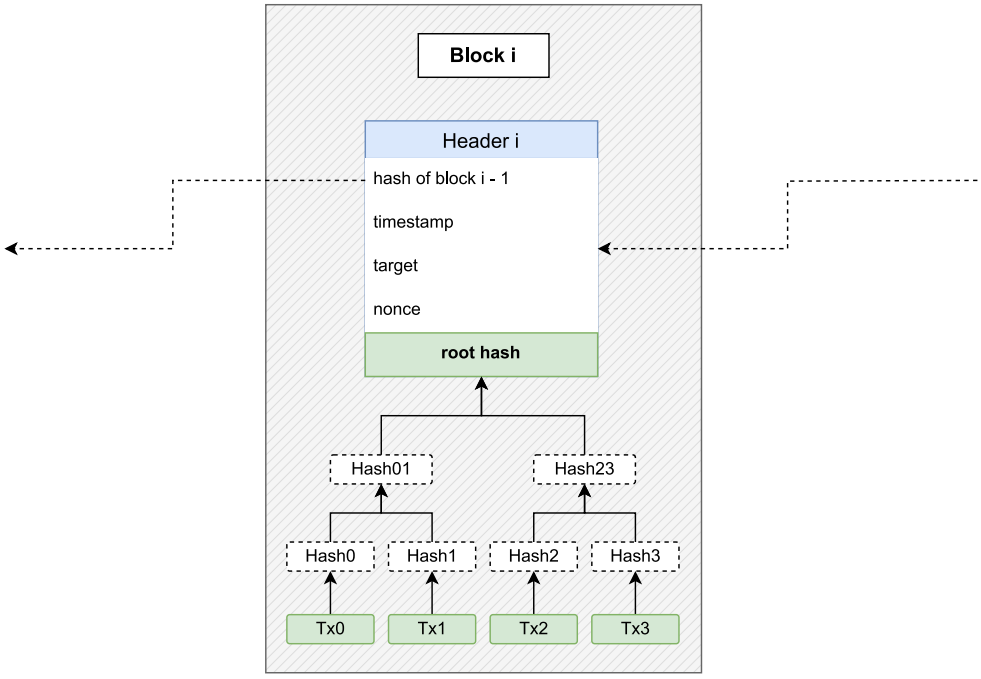

Figure 1.2: The structure of block in Bitcoin

1.1.3 Proof-of-Work

To ensure the existing chain is not tampered with, Bitcoin implements a **proof-of-work** system, which involves incrementing the nonce field in a block until a value is found that produces a hash of a block that has a fixed amount of zero bits at the beginning. Finding such a value is colloquially called *mining*. Once the work is done, the mined block can't be changed without redoing the proof. The average work required to find a value that satisfies this condition is exponential to the amount of zero bits required and the verification can be done by simply hashing the block. The proof's difficulty is determined by a moving average targeting a creation of one block every ten minutes. Proof-of-work also ensures that in majority decision making process one CPU is equal to one vote.

The successfully mined blocks are broadcasted across the network and accepted only if the transactions inside are valid and not spent. The approval is expressed by inserting the hash of the approved block in the header of the next block. Honest network users are always working on extending the longest chain as it is considered the correct one. If some nodes find the proof at the same time and they differ from each other, the users may receive different versions of the block. When that happens, the nodes will work on the first one received and keep the other one for backup. The conflict is resolved when the chain is extended further and the users switch to the longer branch.

To incentivize users to participate in the mining process, the network awards new currency and the transaction fees to the person that manages to find the proof-of-work in a special transaction called *coinbase*. The Bitcoin is designed to have a maximum supply of 21 million BTCs and this is regulated by the miner's rewards. At first the coinbase transaction awarded 50 BTCs, but every 210 000 blocks this value is halved and now the reward is at 6.25 BTCs. After reaching the maximum supply the network will cease to create new currency and the miners will only be awarded the transaction fees.

1.2 Ethereum

After the success of Bitcoin many other blockchains have emerged with different ideas how to utilize this type of network. One of such systems is Ethereum [1] formalized in 2014 by Vitalik Buterin with the intent to provide a platform for creating decentralized applications. To achieve that Ethereum implements a Turing-complete programming language which allows for sophisticated scripting capabilities in regard to ownership rules, transaction formats, various operations on data and storage. While Bitcoin is often described as a distributed ledger, Ethereum is more akin to a distributed transaction-based state machine where the state is an enormous data structure consisting of all user data, every published "program", balances and the state of the *Ethereum Virtual Machine* (**EVM**) which is the execution model of this blockchain. There are several implementations of EVM in various programming languages which adhere to the specification described in the yellowpaper [10] maintained currently by Nick Savers and other members of the community.

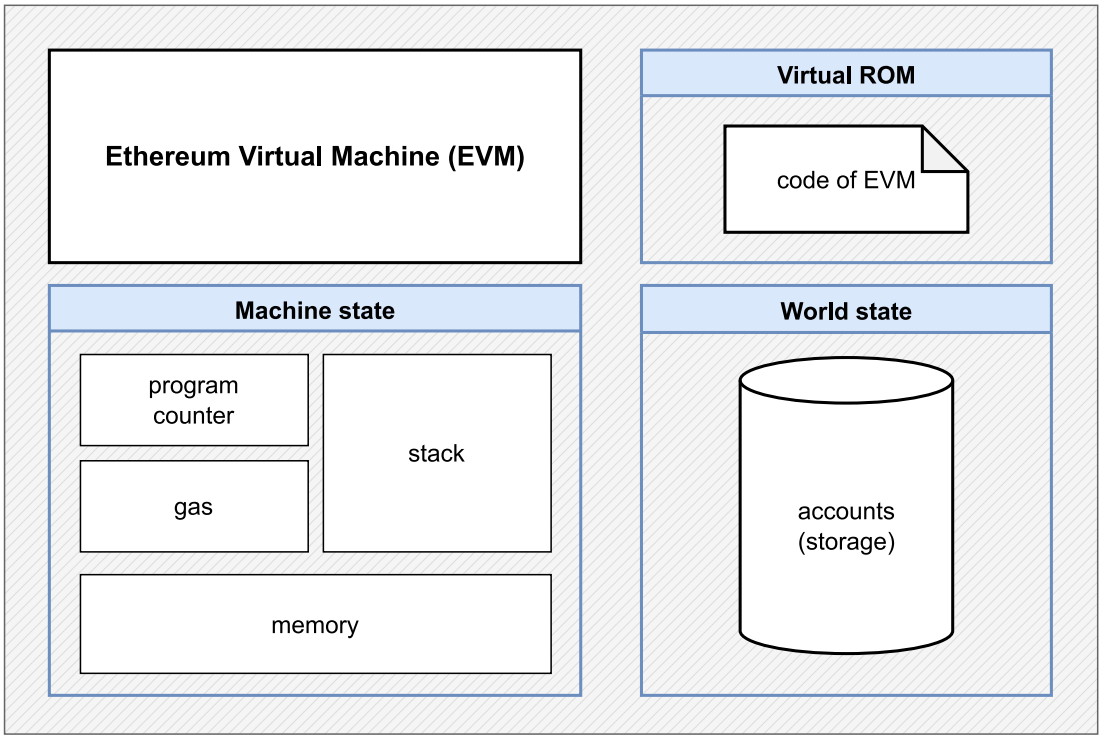

Figure 1.3: The structure of Ethereum Virtual Machine

1.2.1 Accounts

The Ethereum state consists mostly of objects called *accounts* which represent users, also referred to as *externally owned accounts*, and *contracts*. Each account is composed of four fields:

- `nonce`, a counter showing the number of transactions sent from this account used to mitigate replay attacks;
- `balance`, the amount of *ether*(**ETH**) — the currency of the blockchain that can be exchanged between users or used to pay for computational power — held by the account;

- `codeHash`, a hash of the code of the account on EVM that is stored in the state database (for externally owned accounts this value is equal to a hash of an empty string);
- `storageRoot`, a hash of the root of a Merkle Patricia trie [10, Appendix D] that encodes the storage contents of the account (empty by default).

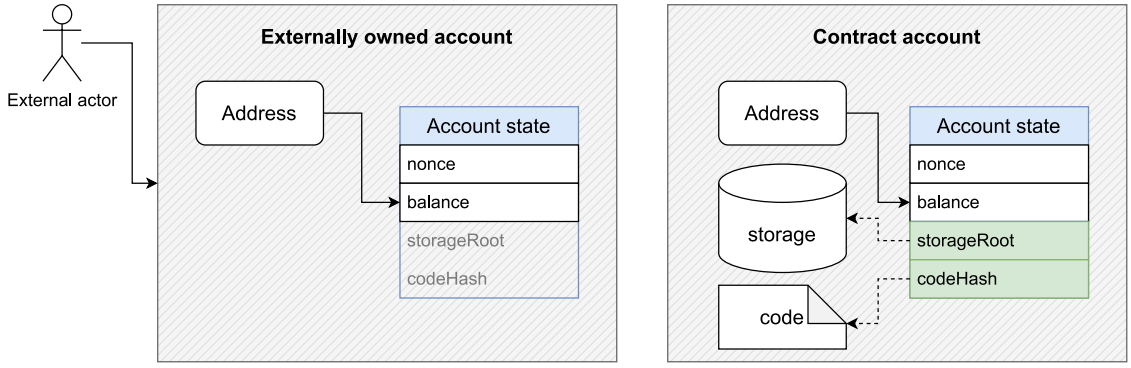

Figure 1.4: The account structure in Ethereum

Externally owned accounts are regular network users' accounts which can be created for free by generating a pair of cryptographic keys (public and private). Possession of the private key proves the ownership of the account and allows direct control over operations associated with it, such as interactions with other accounts and exchange of ETH or tokens.

Contracts are a special type of Ethereum account that acts as program which runs on the blockchain. They aren't directly controlled by users, as they don't have any cryptographic keys connected to them, instead they are deployed and run as programmed automatically when prompted. Contracts by default can't be deleted and interactions with them are irreversible. Users can deploy a contract to the network, however the cost for this operation is much higher than for a transfer, and the contract must be already compiled so it can be stored and interpreted by the EVM. The two main limitations of contracts are the size (max 24 KB) and the inability to send HTTP requests.

1.2.2 Transactions and Messages

In Ethereum, there are two data structures that are used for communication in the network. Externally owned accounts can send *transactions* which are cryptographically signed data packages that update the state of the blockchain. There are three main applications of transactions: to exchange resources between accounts, to put smart contracts into the network and to interact with the contracts that are present in the network. While originally there has been one set format of a transaction, after EIP-2718 [11] there is possibility for extra 128 formats. The original format, called a legacy transaction, consists of nine fields:

- `nonce`, the number of transactions sent by the sender;
- `gasPrice`, the price expressed in Wei (10^{-18} ETH) per unit of *gas* — a resource used in exchange for the network's computational power – that will be used during the execution of a transaction;
- `gasLimit`, the maximum amount of gas the execution of a transaction may use;

- `to`, the address of the recipient, empty if creating a contract;
- `value`, amount of ether to be transferred in Wei;
- `data`, an unlimited size byte array that contains arbitrary information (e.g. the code of a contract);
- `v`, `r` and `s`, components of an ECDSA signature of the sender.

Accounts communicate with each other via *messages* which consist of the same fields as transactions except for the signature components. This structure exists only in Ethereum execution environment and is not serialized in blocks like transactions are. Contract accounts are unable to send transactions due to lack of cryptographic keys but they can send messages if they are programmed to do so. This allows to establish relationships between contracts to create complex applications on the blockchain. The figure 1.5 shows all of the possible ways a message can be sent: either triggered by a transaction or by the EVM code.

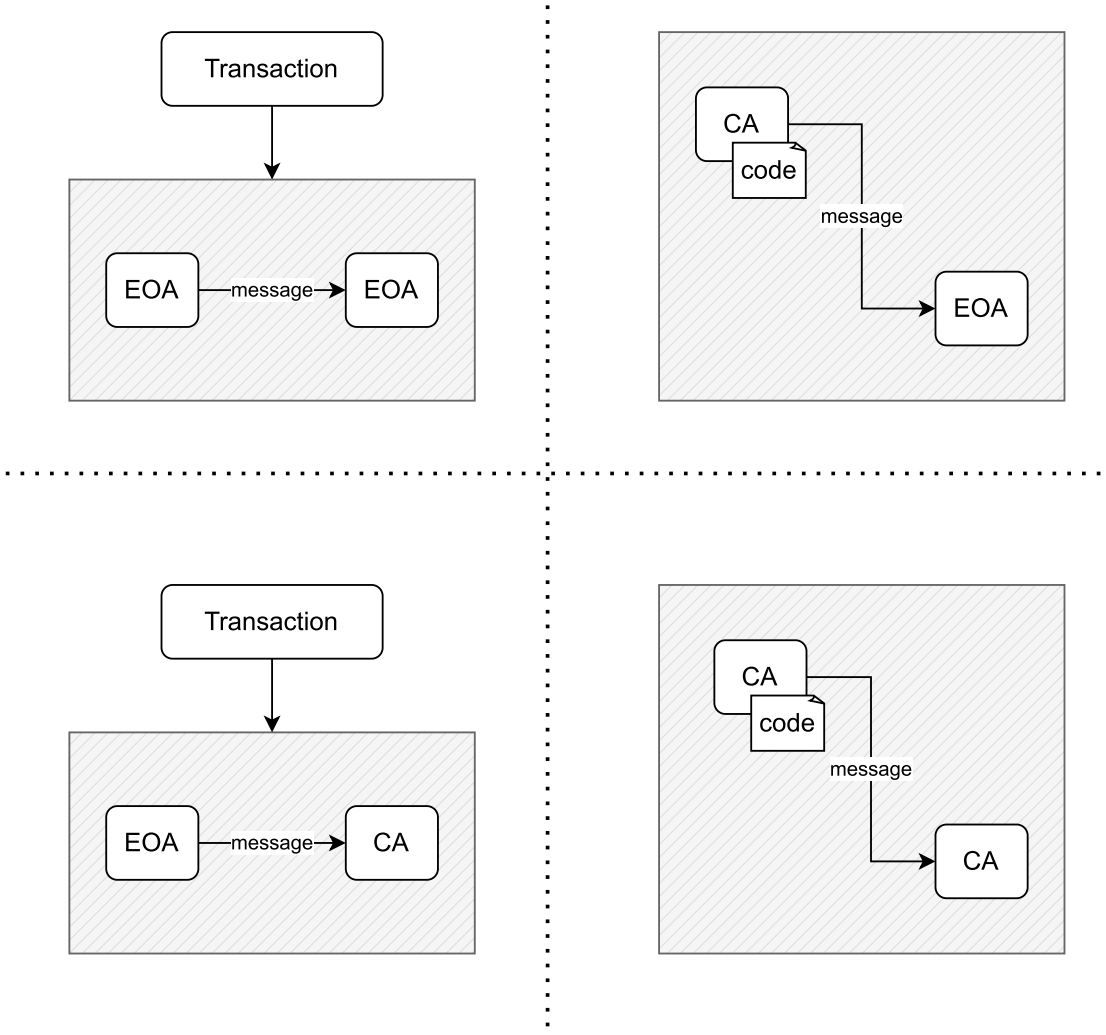

Figure 1.5: The cases in which messages are sent in Ethereum

1.2.3 Proof-of-Stake

Similarly to Bitcoin, Ethereum batches transactions in *blocks* which are linked together by including a hash of a previous block in the new one. Originally the network operated on a proof-of work system like Bitcoin. However, in 2022 Ethereum migrated to a new validation protocol called **proof-of-stake**. The protocol runs in accordance to the Gasper algorithm [2] which is a combination of the Casper-FFG (Casper the Friendly Finality Gadget) algorithm that oversees the voting process and finalization of blocks and the LMD-GHOST (Latest Message Driven Greediest Heaviest Observed SubTree) algorithm that selects the version of the chain that has greatest weight of attestations in its history where only the latest messages from the validating nodes are considered. The change substantially reduced the energy consumption of the network, lowered the strain on the hardware running the nodes and shortened the block creation time. The structure of a block and the way the network operates have significantly changed during the migration. The new blocks consist of five main parts:

- `slot`, a period of time when the block has been created;
- `proposer_index`, the node that has created the block;
- `parent_root`, the hash of the previous block;
- `state_root`, the root hash of the global state;
- `body`, a large structure consisting of all values necessary for the proof-of-stake protocol, the root hash of the global state after applying transactions, the id of the block, the timestamp, the amount of gas used when executing transactions, some additional data and the list of transactions.

Validators

In the proof-of-stake protocol only a group of nodes, *validators*, can propose and validate new blocks. To become a validator, a user must deposit 32 ETH to the special contract on Ethereum Mainnet and run three separate programs: an *execution client*, a *consensus client* and a *validator*. The deposit is a deterrent to dishonest activities, because it gets destroyed once the malicious actions are proven. After submitting the deposit, the candidate joins the queue limiting the rate of new validators and waits to be activated. Once the setup is completed, they start to receive new blocks that need to be verified and can get selected to be a creator of a block.

Block Proposals

In proof-of-stake the tempo of block creation is fixed unlike in proof-of-work where the tempo is affected by the proof's difficulty. The time is divided into *epochs* consisting of 32 *slots* which are 12-second time intervals. One validator is randomly selected for each slot to create a block, broadcast it to the network and update the global state. The protocol randomly assigns a team of validators for each slot that vote in favor or against newly proposed block. To verify the block, a validator needs to reexecute the transactions in the execution client to check if the state change is valid. Then, if the block is correct, the validator sends a vote in favor of the block to the network and adds it to their local database.

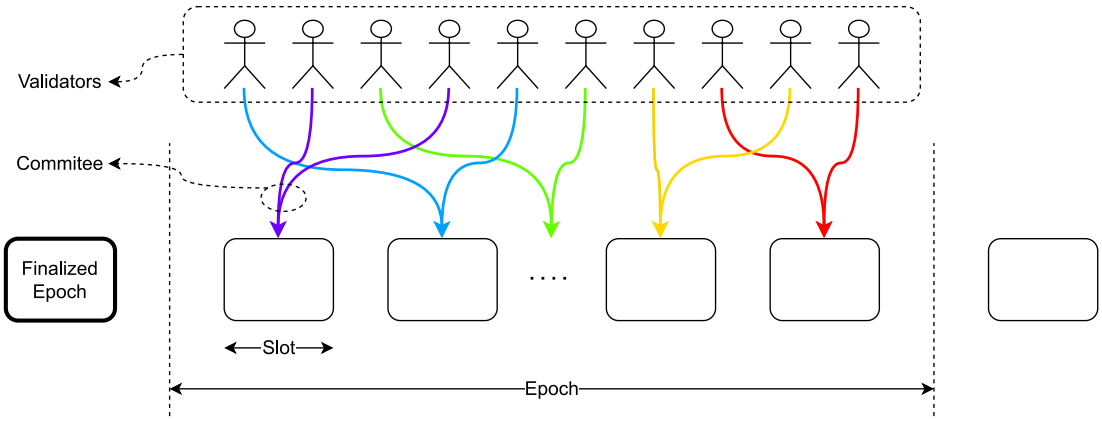

Figure 1.6: Voting for new blocks in Ethereum's proof-of-stake protocol

Finality

The transactions can be perceived as confirmed and almost irreversible when they are a part of a *finalized* block. Finality is a property of a block that means it can't be reverted without destroying a significant amount of ether. The finalization of an epoch is achieved with *checkpoint blocks*, which are the first blocks in an epoch. Validators vote for a pair of checkpoints they consider valid. If such a pair attracts two third of total staked ether, the more recent block becomes *justified* and the older upgrades to a *finalized* state. In other words, if validators agree on the two latest epochs, all epochs before them are considered finalized. To prevent blocking finality by having one third of staked ether, if the chain doesn't finalize for more than four epochs, the staked currency of validators against finalization is slowly destroyed until the necessary threshold is reached. This mechanism is called *inactivity leak*.

Rewards and Penalties

The main incentives to become a validator despite the high initial investment are the rewards for honest and diligent work. The users who vote consistently with the majority, propose new blocks and participate in committees are paid in ether based on their performance and the `base_reward` value which depends on the staked ether across all active validators. The `base_reward` is calculated as:

$$\text{base reward} = \text{effective balance} \times \left(\frac{\text{base reward factor}}{\text{base rewards per epoch} \times \sqrt{\sum \text{active balance}}} \right)$$

where `base_reward_factor` is equal to 64, `base_reward_per_epoch` is equal to 4 and the sum of `active_balance` is the total staked ether of all validators. The base reward is proportional to the validator's effective balance (ether staked by them) and inversely proportional to the number of validators on the network. That means the rewards for a single validator are smaller if there are many validators in the network but the overall amount of ether issued is greater. Validators can get an additional reward for fast block attestation equal to:

$$\text{base reward} \times \frac{1}{\text{delay}}$$

where the `delay` is the number of slots between the block's proposal and the vote, and for providing the evidence of malicious activity equal to $\frac{1}{512}$ of the penalty applied to the misbehaving node.

The validators are not penalized for inefficient work but they get smaller rewards if their performance drops. However, the nodes can be affected by the inactivity leak if they are inactive during the checkpoint vote, especially if they have significant amount of staked ether. If the validator exhibits dishonest behavior that is detected by someone, they are forcefully removed from the network and lose their staked ether. Such an action is called *slashing*. There are three activities that result in applying this punishment:

- proposing and signing two different blocks in the same slot;
- voting in favor of two candidates for the same block;
- voting in favor of a block ”surrounds” the other one.

Chapter 2

Stamp & Extend

This chapter describes in depth the protocol that will be adapted and implemented in the blockchain system. **Stamp&Extend** [5] is a time stamping scheme built upon *Schnorr signatures* [8] and *Pedersen commitments* [7]. The protocol utilizes a centralized *Time Stamping Authority* (**TSA**) to issue timestamps and provide transparent records allowing anyone to audit them. The TSA is deterred from manipulation of the data by the threat of the private key leakage provided by the commitments and the structure of the timestamp which allows to easily detect any modifications. The security of the scheme is based in particular on the hardness of the *Discrete Logarithm Problem* (**DLP**) in cyclic groups of prime order q such that $p = 2q + 1$ is still a prime. The problem can be defined as follows:

$$\text{given an element } y = g^x \text{ of a group } G \text{ where } x \in \mathbb{Z}_q, \text{ find } x.$$

The sections below describe the Stamp&Extend scheme using the modernized notation. Some modifications have been introduced to the verification algorithm to adapt to the changes in the way the Schnorr signatures are calculated.

2.1 Existing schemes in the protocol

2.1.1 Schnorr Signatures

Let G be a group of a prime order q where the Discrete Logarithm Problem is hard. Let g be a generator of G. Let $\mathbb{Z}_q$ be a ring of integers modulo q.

Key generation

The private key of the signer is a number a chosen randomly from $\mathbb{Z}_q$. The corresponding public key A is an element of G equal to g^a.

Signing procedure

Signature creation requires the use of a cryptographic hash function H. A signer possessing the private key a can sign the message m using the algorithm 2.1.

Algorithm 2.1: SchnorrSign(a,m)

Data: a private key of a signer a, a message m
Result: a signature $\sigma = (X,s)$

```
1 x ← $Z_q                          // x is chosen randomly from Z_q
2 X ← g^x
3 h' ← H(X,m)                       // X and m are concatenated
4 s ← x + ah'
5 return σ = (X,s)
```

Verification procedure

A verifier can confirm that signature σ was created by a signer identified by the public key A for a message m with the algorithm 2.2. Verification requires the use of the same cryptographic hash function H as in the signing procedure.

Algorithm 2.2: SchnorrVerify(A,m,σ)

Data: a public key of a signer A, a message m, a signature σ
Result: *valid* or *not valid*

```
1 X,s ← σ
2 h' ← H(X,m)                       // X and m are concatenated
3 if g^s = XA^h' then
4 |   return valid
5 else
6 |   return not valid
```

2.1.2 Pedersen Commitments

Let G be a group of a prime order q where the Discrete Logarithm Problem is hard. Let g,h be generators of G such that no one knows $\log_g h$. Let $\mathbb{Z}_q$ be a ring of integers modulo q.
The committer creates a commitment c to k by choosing l at random from $\mathbb{Z}_q$ and calculating

$$c = g^k h^l$$

Such commitment can be opened at a later date by revealing k and l. The commitment reveals no information about k and cannot be opened to $k' \neq k$ without the knowledge of $\log_g h$.

2.2 Description of the protocol

2.2.1 Initialization

Before the protocol can run, all the necessary parameters need to be established. The first step is to choose a group G of a prime order q, where the Discrete Logarithm Problem is hard, and a secure hash function H. Next, the keys for the Time Stamping Authority and initial commitments are created by the algorithm 2.3.

Algorithm 2.3: SEKeyGen(G, g, h)

Data: a group G of a prime order q, generators g, h of a group G where $\log_g h$ is secret
Result: the private key a, the public key A, the first pair of commitment parameters (k_1, l_1), the first Pedersen commitment c_1

1 $a \leftarrow \$\{1, 2, \ldots, q-1\}$ // a is chosen randomly
2 $A \leftarrow g^a$
3 $(k_1, l_1) \leftarrow \$\{0, 1, 2, \ldots, q-1\}$ // k_1,l_1 are chosen randomly
4 $c_1 \leftarrow g^{k_1} h^{l_1}$

The protocol also employs the use of a certificate to confirm the public key A and initial commitment c_1, so that the initial parameters cannot be changed after initialization. The certificate is used as the first element of the timestamp chain HS_0 and the root of trust in the verification algorithm.

2.2.2 Timestamp creation

The protocol for its execution requires the storage of the following information from the TSA:

- $i-1$, the index of the last created timestamp;
- $P = [(k_i,l_i), \ldots, (k_{2i-1}l_{2i-1})]$, a private list of exponents used by Pedersen commitments;
- $C = [c_1, \ldots, c_{2i-1}]$, a public list of Pedersen commitments;
- $HS = [Cert, (T_1,H_1), \ldots, (T_{i-1},H_{i-1})]$, a public list of issued timestamps where the initial element HS_0 is the certificate generated during the initialization phase and the next elements are tuples HS_j for $j \in 1, \ldots, i-1$, where $T_j = (S_j, l_j, j)$ is the timestamp and H_j is the timestamped data.

The relationship between the timestamps and the commitments is visualized by the figure 2.1. The structure of the HS resembles a blockchain (blue lines) as the timestamps are linked together by the hashes of the previous elements. The chain of timestamps is also connected by the commitments embedded in them forming a secondary structure resembling a binary tree (dashed lines). With every new timestamp, two new elements c_{2i} and c_{2i+1} (circles) are added to the list of commitments C, and linked to the i-th timestamp. This approach forms $\lfloor \log_2 i \rfloor$ step paths from each timestamp to the root of trust (HS_0). This way, the timestamps can be verified without having to check every preceding element in HS. The private exponents k_i and l_i used in the creation of the i-th timestamp are discarded shortly after using them to save storage space because there is no further need for keeping them. The timestamps' creation process and their internal structure are described more thoroughly by the algorithm 2.4.

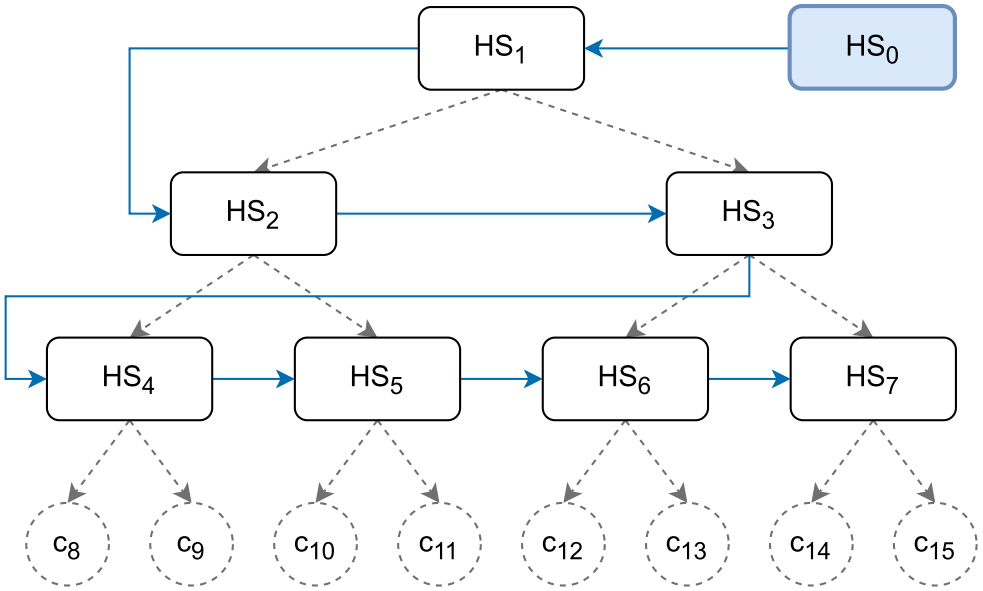

Figure 2.1: The connections between timestamps and commitments in the Stamp&Extend protocol

Algorithm 2.4: SECreateTimestamp($i-1$,C,P,HS,H_i)

Data: the index of a previous timestamp $i-1$, the list of Pedersen commitments C, the list of unused private exponents in commitments P, the list of issued timestamps HS, the data to be timestamped HS_i

Result: the index of new timestamp i, the updated lists C,P and HS

/* Generate two new commitments and extend P and C */

1 $k_{2i}, l_{2i}, k_{2i+1}, l_{2i+1} \leftarrow \$\{0, 1, \ldots, q-1\}$ // values are chosen randomly

2 $c_{2i} \leftarrow g^{k_{2i}} h^{l_{2i}}$

3 $c_{2i+1} \leftarrow g^{k_{2i+1}} h^{l_{2i+1}}$

4 P.append((k_{2i}, l_{2i}),(k_{2i+1}, l_{2i+1}))

5 C.append(c_{2i}, c_{2i+1})

/* Remove the exponents to be used in the new timestamp from P */

6 k,$l \leftarrow P$.pop(k_i,l_i)

/* Create a Schnorr Signature of the following value m using k in place of random x */

7 $m \leftarrow (H(HS_{i-1}), H_i, c_{2i}, c_{2i+1}, l, i)$ // $H()$ is a cryptographic hash function

8 $S_i \leftarrow$ SchnorrSign(a,m) // TSA knows a

/* Form a timestamp and add it to HS */

9 $T_i \leftarrow (S_i, l, i)$

10 HS.append((T_i, H_i))

11 **return** i

2.2.3 Timestamp verification

The verification can be performed by any party that has access to the list of commitments C and the chain of timestamps HS. The validity of a timestamp is verified by following the path from the HS_i to HS_0 which is a trusted value. Schnorr signatures inside the timestamps prove the continuity and the integrity of the data inside it. The signatures also contain a part of a corresponding commitment that can be reconstructed to check whether the secret exponent k

was used during the signing as an additional security measure. Any other value used instead of k is easily detectable. The verifier has all of the necessary information to reconstruct the message inside the signature and can check if the chain of timestamps has been tampered with by following the algorithm 2.5. The original verification contained a small error in the commitment reconstruction where c'_j was equal to $g^{e_j} y^{s_j} h^{l_j}$ which yields $g^{e_j + x(k_j - xe_j)} h^{l_j}$ instead of $g^{k_j} h^{l_j}$. This problem has been addressed in the fifth line of the algorithm 2.5.

Algorithm 2.5: SEVerifyTimestamp(i,C,HS)

Data: the index of a timestamp i, the list of Pedersen commitments C, the list of issued timestamps HS
Result: *valid* or *not valid*

/* Get the public key of TSA from HS_0 */
1 $A \leftarrow HS_0$
2 **for** $\alpha = 0, \ldots, \lfloor \log_2 i \rfloor$ **do**
3 $\quad j \leftarrow \lfloor \frac{i}{2^\alpha} \rfloor$
$\quad$ /* Reconstruct the message in S_j */
4 $\quad m \leftarrow (H(HS_{j-1}), H_j, c_{2j}, c_{2j+1}, l_j, j)$
5 $\quad$ **if** *not (SchnorrVerify*(A,m,S_j) == *valid) or not* ($c_j == X_j h^{l_j}$) **then**
6 $\quad\quad$ **return** *not valid*
7 **if** c_1 *is confirmed by* HS_0 **then**
8 $\quad$ **return** *valid*
9 **else**
10 $\quad$ **return** *not valid*

Chapter 3

Distributed Timestamping System

This chapter describes in detail the distributed system that was designed using the Stamp&Extend scheme as a foundation and includes the instructions on how to set up and use the prototype.

3.1 The design

The system consists of two types of nodes:

- *client nodes*, which can access the lists of timestamps and commitments, request a timestamp for specific data and verify the validity of the timestamp chain at any point;
- *server nodes*, which are responsible for issuing timestamps for submitted data and publishing them.

Both types of nodes are communicating through the Ethereum network by interacting with a special smart contract that serves as a ”gateway”.

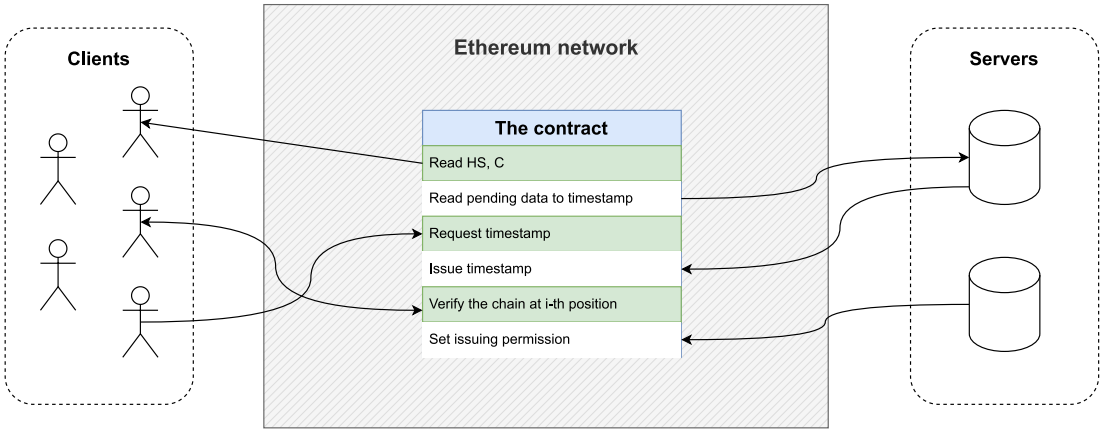

Figure 3.1: The architecture of the timestamping system

The system is designed this way to mitigate one major security risk that would compromise the security of the whole Stamp&Extend scheme. Even though the Ethereum Virtual Machine is fully capable of creating a timestamp, the storage of the contract accounts is public, so anyone could read the data that is saved there. There is practically no way to securely store any secret on the blockchain, so the timestamps need to be created outside Ethereum environment. However,

the other functionalities can be implemented using blockchain without compromising the security of the system.

The Stamp & Extend protocol requires the publication of timestamps and commitments and Ethereum is a perfect environment to do that. Decentralized nature of the network ensures that the access to the data is unrestricted for all and there is no downtime (unless every node is down which is extremely unlikely). The blockchain also acts like a version control system that preserves every change made in the data and all transactions connected to the contract, so every malfunction or malicious activity is visible to anyone browsing the network. Pairing that with the advanced computing capabilities of the EVM allows to create a lightweight program that can verify the correctness of the timestamp chain, accept the requests from various clients, receive and store the data issued by server nodes and notify the network of the changes.

3.1.1 Smart contract

The contract acts as a mediator between the clients and the server nodes. It combines several functionalities to efficiently process timestamping requests from clients and provides ways to audit the results of protocol execution. The contract keeps in its storage the following data:

- `HS`, a list of timestamps following the set format (look 3.1) issued so far;
- `C`, a list of created commitments;
- `pubKey`, the public key of server nodes;
- `H`, the second generator for commitment verification;
- `pendingTimeStamps`, a mapping between addresses and data waiting for a timestamp;
- `issuers`, a mapping of addresses allowed to publish a timestamp.

```
1    struct TimeStampEntry {
2        Curve.G1Point X;  // first part of Schnorr signature
3        uint256 s;        // second part of Schnorr signature
4        uint256 l;        // private exponent in the corresponding commitment
5        uint256 i;        // index of a timestamp
6        uint256 data;     // timestamped data
7    }
```

Listing 3.1: The structure of a timestamp in Solidity

The `pubKey`, the generator `H` and the first elements of `HS` and `C` are supplied during the contract deployment. The account that publishes the contract is also added as an approved account to `issuers`. The compiler automatically creates getters only to single values like the public key or a chosen element from the timestamp list. To make access to the data easier, the contract provides functions which allow to read whole `C`, `HS` and `pendingTimeStamps` at once.

The contract also includes a function responsible for registering the timestamp requests which temporarily stores the address of the requester and the data and emits an event informing of a new request which are monitored by the server nodes (look 3.2). There are no limitations in terms of who can request a timestamp. The only requirement is to not exceed only one pending request per address.

```
    function requestTimeStamp(uint256 data) external {
        require(
            pendingTimeStamps[msg.sender] == 0,
            "You already requested a timestamp!"
        );
        pendingTimeStamps[msg.sender] = data;
        pendingIndexes.push(msg.sender);
        emit TimeStampRequested(msg.sender, data);
    }
```

Listing 3.2: Gathering requests from clients

The server nodes are allowed to publish new timestamps and commitments created for requested data. The following listing 3.3 shows the function that updates the storage, emits an event about the new timestamp and removes the request from the log. The array indexing the stored addresses is copied to the memory to reduce gas fees because accessing the storage costs significantly more that accessing the memory.

```
    function issueTimeStamp(
        uint256[6] memory ti, uint256[4] memory comms, address requester
    ) external {
        require(issuers[msg.sender], "Only approved accounts allowed!");
        require(
            pendingTimeStamps[msg.sender] != 0,
            "This person didn't request a timestamp!"
        );
        // Extend HS
        TimeStampEntry memory HSi = TimeStampEntry(
            Curve.G1Point(ti[0], ti[1]), ti[2], ti[3], ti[4], ti[5]
        );
        HS.push(HSi);
        // Extend C with c_2i,c_2i+1
        C.push(Curve.G1Point(comms[0], comms[1]));
        C.push(Curve.G1Point(comms[2], comms[3]));
        // Let the requester know that timestamp has been issued
        emit TimeStampIssued(requester, HSi);
        // Delete address from pending record
        delete pendingTimeStamps[requester];
        address[] memory _requesters = pendingIndexes;
        for (uint256 i = 0; i < _requesters.length; i++) {
            if (_requesters[i] == requester) {
                pendingIndexes[i] = _requesters[_requesters.length - 1];
                pendingIndexes.pop();
                break;
            }
        }
    }
```

Listing 3.3: Publishing the timestamp

Finally, the contract features a function that can verify the chain of timestamps from a selected index to the root of trust HS_0 using logarithmic shortcuts provided by the commitments (look 3.4). The verification uses the same gas saving tricks as the previous function plus a custom `log2floor` function which calculates $\log_2 i$ rounded down to the nearest integer. After confirming that every timestamp on the path is valid, the function also checks if the root of trust is not malformed. The verification returns true only if every element on the path including the HS_0 has a valid Schnorr signature and the correct commitments were used during their creation.

```
    function verifyTimeStamp(uint256 i) external view returns (bool valid) {
        require(i >= 1, "Accepting only i >= 1.");
        TimeStampEntry[] memory _HS = HS;
        require(i < _HS.length, "We haven't issued this timestamp yet.");
        Curve.G1Point[] memory _C = C;
        uint256[2] memory pk = [pubKey.X, pubKey.Y];
        Curve.G1Point memory _H = H;

        for (uint256 alpha = 0; alpha <= log2floor(i); alpha++) {
            uint256 j = i / (2**alpha);
            // Hash j-1-th timestamp
            uint256 hs1 = uint256(
                keccak256(abi.encodePacked(_HS[j - 1]))
            );
            // Reconstruct message
            uint256 m = uint256(
                keccak256(
                    abi.encodePacked(
                        hs1, _HS[j].data, _C[2*j-1], _C[2*j], _HS[j].l, _HS[j].i
                    )
                )
            );
            uint256[2] memory x = [_HS[j].X.X, _HS[j].X.Y];
            bool proof = Schnorr.VerifyProof(pk, m, x, _HS[j].s);
            Curve.G1Point memory c_j = Curve.g1add(
                _HS[j].X, Curve.g1mul(_H, _HS[j].l)
            );
            if (!proof || c_j.X != _C[j - 1].X || c_j.Y != _C[j - 1].Y) {
                return false;
            }
        }
        uint256 cert_m = uint256(
            keccak256(abi.encodePacked(pk[0], pk[1], _C[0].X, _C[0].Y))
        );
        uint256[2] memory xx = [_HS[0].X.X, _HS[0].X.Y];
        return Schnorr.VerifyProof(pk, cert_m, xx, _HS[0].s);
    }
```

Listing 3.4: Verification of the timestamp chain

3.1.2 Clients

Client nodes can interact with the system by contacting the contract with transactions or message calls. They can embed the connection and interactions in their applications or use any tool that allows to send transactions or messages on Ethereum. The details of the client node design are not a part of this master thesis.

3.1.3 Server node

The server node's primary function is to respond to the requests recorded by the smart contract. To effectively perform this task, the node scans new blocks in the network for events about an incoming request (look 3.5). To make sure that a pause in operation doesn't ignore missed requests, the server reads the request log from the contract and responds to them before the asynchronous scanning loop begins.

```
    # Checking the backlog in the contract
    pending_data = contract.functions.getPendingData().call()
    if (len(pending_data[0]) >= 1):
        print(f"You missed some requests: {pending_data}")
        for i in range(len(pending_data[0])):
            issue_timestamp(
                stamp_extend, pending_data[0][i], pending_data[1][i])

    # Monitor incoming events
    event_filter = contract.events.TimeStampRequested.create_filter(
        fromBlock='latest'
    )
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(
            asyncio.gather(
                log_loop(event_filter, 2, stamp_extend)
            )
        )
    finally:
        # close loop to free up system resources
        loop.close()
```

Listing 3.5: Scanning for new requests

The server generates timestamps in accordance to the algorithm defined in the Stamp&Extend protocol. Before the data is published it is split into separate integer values, to ensure the contract reads them correctly. To issue a transaction the server needs access to the private key and the address of the account authorized to issue a transaction. The process of building and sending a transaction is presented in the code listing 3.6.

```
    def issue_timestamp(se: StampExtendProtocol, requester, data) -> None:
        T_i, c2i, c2i1 = se.create_timestamp(data)
        t_i = [T_i["X"][0], T_i["X"][1], T_i["s"],
               T_i["l"], T_i["i"], T_i["data"]]
        comms = [c2i[0], c2i[1], c2i1[0], c2i1[1]]
        # Build transaction
        timestamp_tx = contract.functions.issueTimeStamp(
            t_i, comms, requester
        ).build_transaction(
            {
                'from': acc_address,
                'nonce': w3.eth.get_transaction_count(acc_address),
            }
        )
        # Sign tx with PK
        tx_create = w3.eth.account.sign_transaction(timestamp_tx, acc_pk)

        # Send tx and wait for receipt
        tx_hash = w3.eth.send_raw_transaction(tx_create.rawTransaction)
        tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)

        print(f'Tx successful with hash: { tx_receipt.transactionHash.hex() }')
```

Listing 3.6: Publishing a timestamp

The node also requires some form of a safe storage for the protocol data. For the configuration where there are many servers the information needs to be synchronized between them to ensure

the continuity and correctness of the issued timestamps. For a simple configuration with only one server the local encrypted files or a database suffice. The design of the node prototype considers only the one server configuration and includes a module which handles the serialization of data, AES encryption in GCM mode and read/write operations. The module uses a password provided before execution to derive an encryption key for the protocol data.

3.2 Use cases

There are two major situations the system deals with: issuing the timestamps and providing the means to verify the published timestamps. These cases are described below with the help of sequence diagrams.

3.2.1 Timestamp publication

The creation of a timestamp requires the cooperation between the clients, the contract and the server node. The process is initialized by the client who sends a transaction to the contract connected to the function `requestTimeStamp` (look 3.2). Then the contract emits a special event `TimeStampRequested` which contains the address of the requesting party and the submitted data. The server node is monitoring latest blocks for this type of event and after it detects one it responds to it with the `issue_timestamp` function (look 3.6) where it creates a timestamp according to the Stamp&Extend protocol and sends a transaction that contains the timestamp, generated future commitments and the address of the requester to the contract corresponding to the `issueTimeStamp` function (look 3.3). After contract finishes updating data it emits a `TimeStampIssued` event which contains the address of party that requested the timestamp and the contents of the timestamp. The sequences finishes when the client receives the event addressed to them and extracts the timestamp from it.

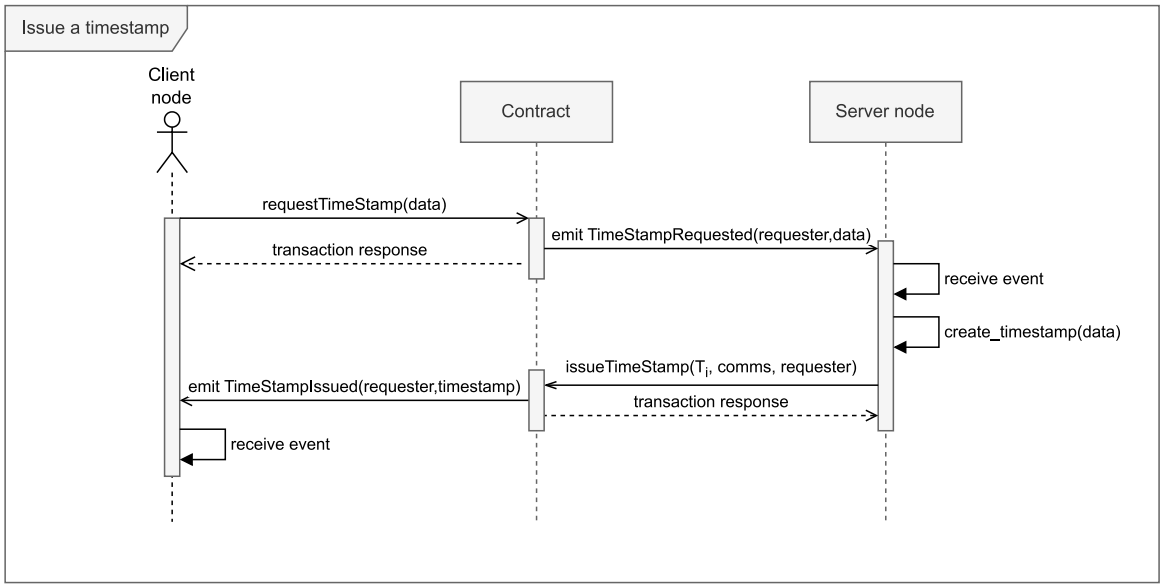

Figure 3.2: The sequence of timestamp publication

3.2.2 Timestamp verification

The verification of a timestamp requires only the interaction between a client node and the contract. The sequence begins when the client sends a message call to the contract connected to the `verifyTimeStamp` function (look 3.4) with the index of the timestamp to be verified. Then the contract checks if the path to the first issued timestamp can be confirmed (each timestamp on the path needs to be correct). Finally, the contract validates the root of trust, the 0-th element of HS. Then if everything is correct, the contract responds with `true` to the client, otherwise `false`.

Alternatively, the client can read the list of timestamps, the list of commitments, the public key and the second generator from the contract and run the verification algorithm themselves. The only requirement is to use the same elliptic curve, hashing procedure and data encoding as the contract.

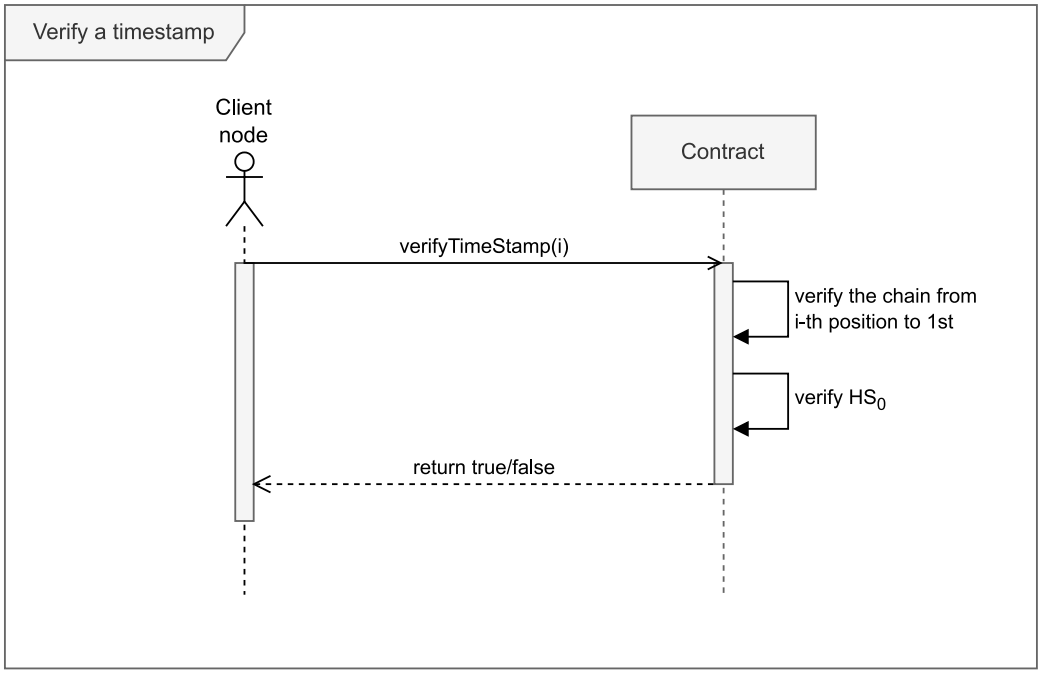

Figure 3.3: The sequence of timestamp verification

3.3 Implementation

The prototype of the system consists of a smart contract and one simple server node. The client node implementation is not necessary for the operation of system as users can interact with the contract using any tool that composes transactions. The smart contract was deployed on the **Sepolia** chain, which is a separate Ethereum environment that mimics the operation of the Ethereum Mainnet. Sepolia was created in October 2021 as a proof-of-authority network and later moved to the proof-of-stake consensus mechanism. Sepolia is the recommended environment for application development [3] because of permissioned validator set consisting of Ethereum's client and testing teams, fast synchronization time, and smaller state size.

The code of the contract was written in **Solidity** (0.8.18) [9] which is one of the programming languages for implementing smart contracts in Ethereum. Solidity is a high-level, object-oriented, statically typed language with syntax inspired by C++ and JavaScript. As the version number

indicates, Solidity is developing rapidly and there are many changes between each minor edition. One of the most recent major additions to the language was the overflow check for mathematical operation which required the use of `SafeMath` library before.

The contract was developed using **Remix IDE**, a web-based development environment for creating and deploying Ethereum-based smart contracts. It requires no setup and provides many features like access to various Solidity compilers, transaction debugger, unit tests generator and many virtual chains where the contract can be tested before deployment. Remix IDE allows to interact with deployed contracts with intuitive user interface that simplifies sending transactions or message calls and displays the responses in a human-readable way.

The prototype of a server node was written in **Python** (3.11.3). The server connects to Ethereum network using **web3** library through an authenticated API connection provided by **Alchemy** which is a platform with a set of web3 development tools to build and scale distributed application on various blockchains. Some of those tools include: webhooks notifying of activity in a distributed app, logs of all requests made by through an API key, transaction monitoring and powerful methods to extract information from the blockchain. Alchemy also dispenses currency for Ethereum Testnets (0.5 ETH a day can be requested for free) which helps developers to test contracts and applications connected to them.

The Stamp&Extend protocol was implemented with the help of a **solcrypto** library [4] which contains definitions of several operations on `secp256k1` and `alt_bn128` elliptic curves. The point multiplication corresponds to an exponentiation operation in the protocol and the point addition to the protocol's multiplication. The hashes of values are calculated using the `keccak256` hash function and then casted to an integer. The code of the library was updated to conform to the syntax of Solidity 0.8.18 and Python 3.11.3. Also the implementation of Schnorr signature was changed to match the notation used in the protocol (look 3.8). In the server node an additional modification was introduced, the ephemeral key is not randomly generated during signing, instead a secret exponent of a corresponding Pedersen commitment is used (look 3.7). The library files were attached to the smart contract and to the server node. To save gas necessary to deploy the contract, unused files from the library were discarded.

```
    def schnorr_create(secret, message, k=randsn(), point=None):
        assert isinstance(secret, long)
        assert isinstance(message, long)
        assert isinstance(k, long)
        A = multiply(point, secret) if point else sbmul(secret)
        X = multiply(point, k) if point else sbmul(k)
        h = hashs(X[0].n, X[1].n, message)
        s = addmodn(k, mulmodn(secret, h))
        return A, X, s
```

Listing 3.7: Schnorr signature creation algorithm used in the server node

```
    function CalcProof(
        uint256[2] memory pubkey,
        uint256 message,
        uint256[2] memory X
    ) internal view returns (uint256[2] memory s) {
        uint256 h = uint256(keccak256(abi.encodePacked(X[0], X[1], message)));
        Curve.G1Point memory A = Curve.G1Point(pubkey[0], pubkey[1]);
        Curve.G1Point memory sG = Curve.g1add(
            Curve.G1Point(X[0], X[1]),
            Curve.g1mul(A, h)
        );
```

```
12          s[0] = sG.X;
13          s[1] = sG.Y;
14      }
15
16      function VerifyProof(
17          uint256[2] memory pubkey,
18          uint256 message,
19          uint256[2] memory X,
20          uint256 s
21      ) internal view returns (bool) {
22          Curve.G1Point memory sG = Curve.g1mul(Curve.P1(), s);
23          uint256[2] memory proof = CalcProof(pubkey, message, X);
24          return sG.X == proof[0] && sG.Y == proof[1];
25      }
```

Listing 3.8: Schnorr signature verification algorithm used in the contract

3.4 Deployment

The full operation of the system requires setting up at least one server node that will respond to the timestamping requests. The server node prototype creates at random the private keys and roots of trust required by the protocol during the first execution, so it is best to run the server node and extract the initial data before attempting to deploy the contract.

The contract can be published to Ethereum using a special transaction that contains the compiled code with the initial parameters. The easiest way to that is to use Remix IDE. Once the Solidity source files are uploaded, they can be compiled using any compiler that matches the version 0.8.18. The list of compilers can be found in the *Solidity compiler* module.

Beside compilation there is also one important thing to do in this step which is to extract the *application binary interface* (**ABI**) of the contract. ABI is a JSON-formatted data that specifies how the communication between a high-level language like Python or JavaScript and EVM occurs. Providing this data is required to communicate with a deployed contract through a dedicated application. The option to copy the ABI is present on the bottom of the *Solidity compiler* module in Remix IDE.

The compiled contract can be deployed in the chosen environment with the *Deploy & run transactions* module. From there one can connect to a cryptocurrency wallet (Injected Provider) or to the other Ethereum node (External Http Provider) and use their account to publish the contract. If the transaction is successful the address of a contract account is returned. This is also a crucial information necessary for future interactions with the contract.

After deployment the contract can be interacted with using message calls and transactions directed at the address of its account. Furthermore, it is important to update the connection parameters in the server nodes, so they can begin responding to the requests. The *Deploy & run transactions* module in Remix IDE remembers the contracts that have been published during the current session and generates an interface to easily interact with them. In the subsequent sessions the interface is still available after referencing the address of the contract account.

Chapter 4

Efficiency Evaluation

4.1 Contract gas usage

The efficiency evaluation of the smart contract has been performed on Sepolia Testnet due to the existence of services that distribute the currency for this chain for free. The contact has been deployed on address `0xBC5B10a20149d9A473a43Ab386a436223Da12220`. Most of the operations are implemented as a `view` functions which means that unless they are called from other contract by a transaction they don't consume gas, because they don't cause a state change. There are three main operations that require the use of a transaction in the Stamp&Extend contract:

- the deployment,
- the request for a timestamp,
- the publication of a timestamp.

The most expensive operation is the deployment. To put the contract on the network the EVM has to burn 1836439 gas which costs 0.0045911 ETH. A bit less expensive is the publication of a timestamp that burns around 266588 gas which costs 0.000000013888 ETH. The least expensive is the request operation that uses approximately 89582 gas. The discrepancy between the price for a publication and other operations is caused by the different methods of sending the transactions. The deployment and requests were made through the cryptocurrency wallet MetaMask which used a priority fee for a unit of gas equal to 2.5 Gwei. However, the transactions publishing the timestamps were sent directly by the server node and the network assigned a priority fee of 0.000052082 Gwei, so even this operation used more gas than the request it still was 16125 times cheaper.

The high gas cost of these operations is mostly associated with access to the storage of the contract and emitting events. The fees for interacting with the storage are as follows:

- 20000 gas for initializing variable (double for arrays because they also store the length);
- 2900 gas for modifying the variable;
- 2000 gas for the first access to a variable in a function;
- 100 gas for each subsequent access to a variable in a function.

The emission of events costs 375 gas per topic and 8 gas for each emitted byte. Moreover, the transactions add an additional fee of 21000 gas to the cost of a function. Combining those values with the necessity of permanent storage of the protocol data we get quite expensive functions.

4.2 Timestamp creation and verification

The experiments have been performed on Asus ROG Strix G512LV with Intel Core i7-10870H CPU and 16 GB of DDR4 RAM in Arch Linux in the Windows Subsystem for Linux environment on Windows 11 operating system. The performance of the `create_timestamp` and `verify_chain` functions have been measured on a samples of 100, 1000 and 10000 randomly generated integers. The tests have been implemented in a Python script `benchmark.py` which is attached to this master thesis. The script uses the simplified version of the protocol without reading and saving data to hard drive because we focus only on the efficiency of the protocol.

4.2.1 Execution time

Most of the results of time measurements of a single timestamp creation fall between 0.092 and 0.105 seconds and the mean time necessary for the execution was 0.0975 seconds. There is a noticeable spike in the test for 10000 sample (look 4.2) but it can be attributed to the malfunctions in the operating system rather than an underlying issue in the protocol (PC entered a sleep mode briefly). The results indicate that the time of creating a single timestamp is not affected by its position in the chain.

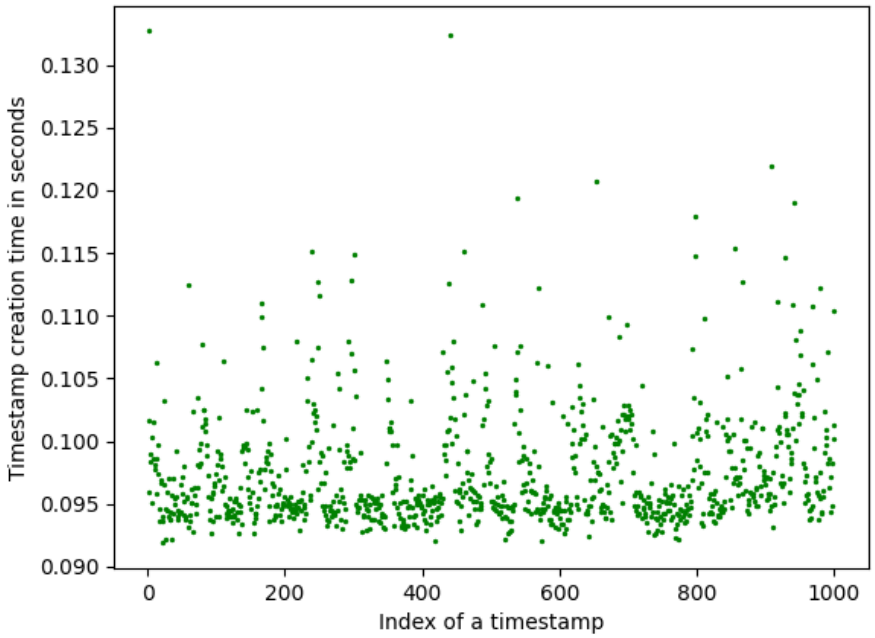

Figure 4.1: Time of a single timestamp creation based on its position in the HS in 1000 sample test

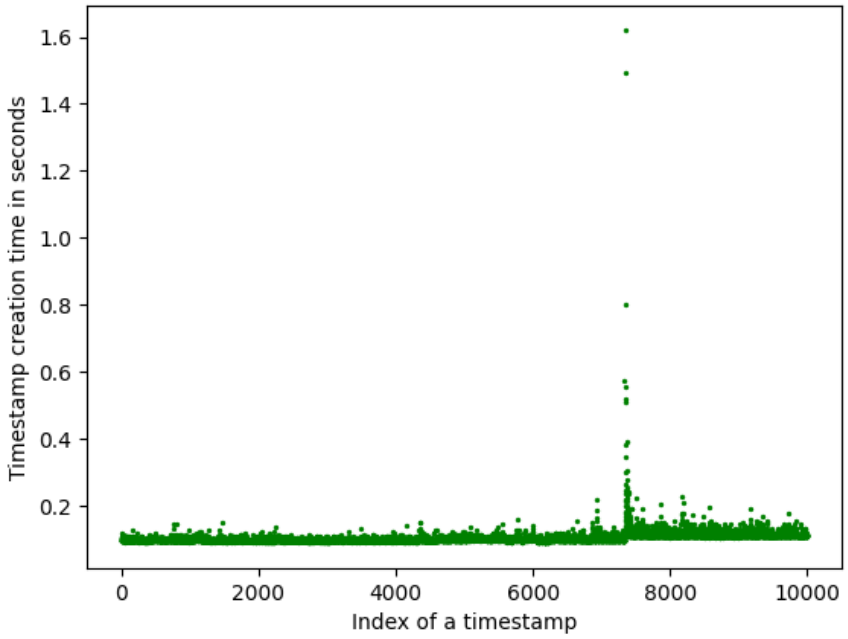

Figure 4.2: Time of a single timestamp creation based on its position in the HS in 10000 sample test

Additionally, the execution time of the timestamp publication sequence (3.2) was measured. The results are presented step-by-step below.

- The time between the client's transaction and a first block creation is between 1 and 11 seconds.
- The time between the block creation and receiving the `TimeStampRequested` event by the server node is between 1 and 2 seconds.
- The time between receiving the event by the server node and sending the transaction with a timestamp is approximately 2 seconds.
- The time between the server's transaction and a second block creation is between 1 and 11 seconds.
- The time between receiving the `TimeStampRequested` event by the client is estimated as 2 seconds.

The time of processing the transaction by the chain usually falls in the middle of block creation time (12 seconds) which is around 7 seconds. Using this approximation the estimated time of issuing a timestamp from the request to receiving `TimeStampRequested` event is 18 seconds.

The time required to verify the chain of timestamps in the Python implementation of the verification algorithm for $i = 1000$ is equal to 0.5065 seconds and for $i = 10000$ is equal to 0.85868 seconds. The complexity of the verification algorithm is related to the path length which gives $O(\log n)$.

4.2.2 Memory usage

The memory profiling of `create_timestamp` and `verify_chain` has been performed on a sample reduced to 100 integers due to a significant reduction of performance while the monitoring is active. A python library `memory_profiler` has been used to capture the memory required by the function in 0.1 second long intervals. The memory usage during creating a timestamp has stayed consistently near 120.5 MB throughout the test which is demonstrated in the first part figure 4.3. The analysis of the recorded data has revealed minimal fluctuations in memory usage which were approximately 0.1 MB. After finishing 100 timestamps the memory usage jumped to 148.6 MB and during the verification it increased by 0.3 MB.

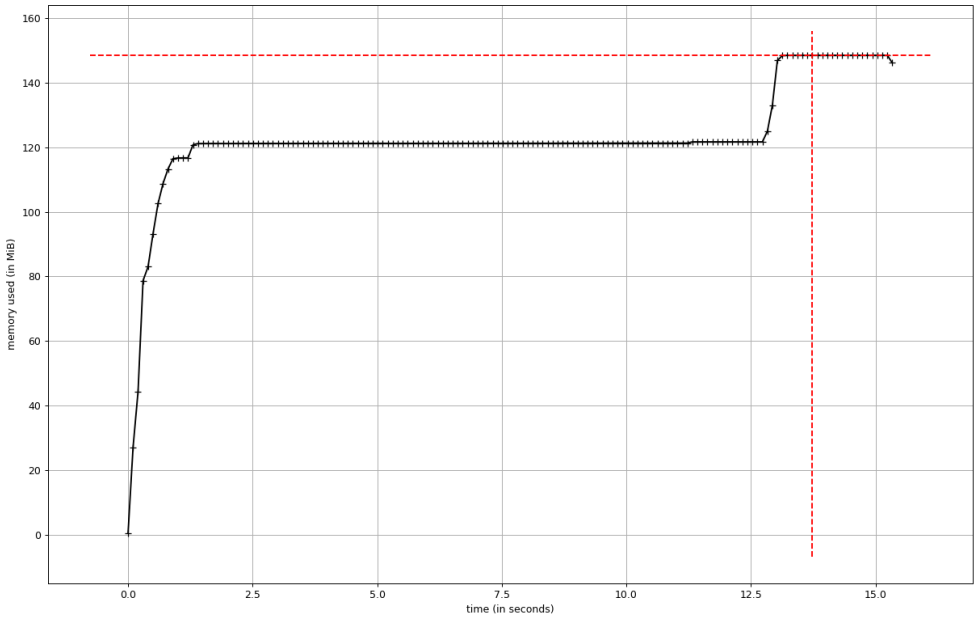

Figure 4.3: Memory usage of timestamp creation and verification

4.2.3 Estimated storage size

The protocol after creating 10000 timestamps has in its memory:

- 0.08123 MB of unused private exponents (10001 elements),
- 0.56259 MB of generated Pedersen commitments (20001 elements),
- 0.28133 MB of generated timestamps (10001 elements).

That gives us around 8 bytes for a pair of future exponents, 56 bytes for a pair of commitments and 28 bytes for the actual timestamp, a total of 92 bytes of data per single created timestamp. The size of a commitment is bigger than expected. One commitment consists of two integers in the encoded form, so a pair of commitments is four integers with approximate size of 14 bytes. The observations revealed that the randomly generated exponents are usually significantly smaller than the coordinates of the point which explains the difference in size.

The protocol data is serialized and encrypted before writing it to a local storage. The generated files have the following sizes:

- 1.61779 MB for list `P`,
- 3.20585 MB for list `C`,
- 4.19577 MB for list `HS`.

It is an expected increase caused by the conversion of the data to JSON format prior to encryption. The record of timestamps exceeds the record of commitments in size which can be explained by the difference in the length of string encoding of those structures.

The estimation of storage size used in the contract has also been performed. The contract after deployment uses 13 slots of storage memory. Each timestamp request requires 3 slots for keeping the log of data waiting for a timestamp which is cleared after the timestamp is issued. Extending the timestamp chain increases the used storage by 10 slots. So after 10000 issued timestamps the contract requires 100013 slots of memory. The memory slots in Ethereum are 32 bytes each and there are 2^{256} available slots for each account, so 10000 timestamps plus 20000 commitments would take only 3.2 MB of storage memory and the possibility of running out of space is negligible.

Chapter 5

Conclusions

5.1 Cryptography and blockchain

The blockchain is still a young technology and there is a lot to improve in terms of smart contracts. Solidity is rapidly developing and has not reached version 1.0.0 yet. There is a great potential to implement selected cryptographic operations, such as signature verification, but there are also some major limitations against the secret-based cryptography in smart contracts.

5.1.1 Advantages

The greatest advantage of blockchain is the full transparency of the data that is present in the network. The full history of changes is visible, state by state, and the consensus mechanism guarantees that it is not tampered with.

The easy access to the network is another major advantage. Creating an Ethereum account is free and there are a lot of services that allow to browse the network. Moreover, the transactions or message calls are not computationally expensive and can run from most devices.

5.1.2 Limitations

The biggest disadvantage of blockchains in terms of implementing cryptographic schemes is the public storage which makes it impossible to store any secret data like private keys. Also, the storage of a contract account in Ethereum is an array of size 2^{256} of 32-byte slots. That limits the size of a single value to a maximum of 256 bits which would make it hard to store or operate on data for schemes that for example use very large integers.

There is also the problem that once you upload a smart contract you can't change its code. If any error or vulnerability is found in the implementation of a scheme there is no way to fix it or to stop the contract. This situation can be circumvented by creating a contract which has exchangeable backend implemented in the other contract or a emergency shutdown switch but it would still leave dead contracts in the world state.

One of the limiting factors is the price of the ether cryptocurrency which is necessary for fueling EVM. Burning ether prevents denial of service attacks and excessive usage of computational power but it also discourages users who don't have heaps of money from interacting with the network.

5.2 The system

The designed timestamping system has fulfilled every functional requirement undertaken by this master thesis. Unfortunately, the properties of the Stamp&Extend scheme made it impossible to implement fully on Ethereum blockchain in a secure way. There are a few ways the design can be improved:

- implement a fail-safe mechanism that disables some function in case there is a major problem with the contract;
- create a prototype of a server node for multi-server configuration of the system;
- add reverting to the `issueTimeStamp` function if the new timestamp verifies as `false`;
- add handling of reverted transactions in a server node;
- lower the polling rate in event scanning loop;

Bibliography

[1] V. Buterin. Ethereum: A Next Generation Smart Contract & Decentralized Application Platform. 2014.

[2] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang. Combining GHOST and Casper, 2020.

[3] Ethereum community. Ethereum development documentation. Accessed May 2023: https://ethereum.org/en/developers/docs/.

[4] HarryR, kilic, drewstone, seresistvanandras. SolCrypto, 2019. Accessed May 2023: https://github.com/HarryR/solcrypto.

[5] L. Krzywiecki, P. Kubiak, and M. Kutyłowski. Stamp & Extend – Instant but Undeniable Timestamping based on Lazy Trees. Cryptology ePrint Archive, Paper 2013/730, 2013. https://eprint.iacr.org/2013/730.

[6] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 2008.

[7] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[8] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 01 1991.

[9] The Solidity Authors. Solidity, 2023. Accessed June 2023: https://docs.soliditylang.org/en/v0.8.18/.

[10] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. 2022. Accessed May 2023: https://ethereum.github.io/yellowpaper/paper.pdf.

[11] M. Zoltu (@MicahZoltu). EIP-2718: Typed Transaction Envelope. *Ethereum Improvement Proposals*, (2718), 2020. Accessed May 2023: https://eips.ethereum.org/EIPS/eip-2718.

Appendix A

DVD contents

The DVD attached to this master thesis includes the following directories:

- `benchmark_results` which contains the results of the tests performed on the server node;
- `contract` that contains the source code of the smart contract, modified `solcrypto` library and the ABI of the compiled contract;
- `tsa` which contains the source code of the server node and the Python implementation of `solcrypto`;
- `diagrams` that contains the illustrations used in the master thesis.

The DVD also includes a copy of this master thesis in a PDF file.