
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# MOBILE CRYPTOCURRENCY WALLET BASED ON ZK-SNARKS AND SMART CONTRACTS

**MOBILNÁ PEŇAŽENKA NA KRYPTOMENY ZALOŽENÁ NA ZK-SNARK TECHNOLÓGIACH A SMART KONTRAKTOCH.**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** — **Bc. SAMUEL SLÁVKA**
**AUTOR PRÁCE**

**SUPERVISOR** — **Ing. IVAN HOMOLIAK, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2022**

**Brno University of Technology**
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

# Master's Thesis Specification


Student: **Slávka Samuel, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Application Development
Title: **Mobile Cryptocurrency Wallet Based on zk-SNARKs and Smart Contracts**
Category: Security
Assignment:

1. Study principles of thin mobile clients and their examples. Acquaint yourself with blockchains and smart contracts.
2. Study principles of zk-SNARKs and their variants.
3. Propose a client-server framework for mobile wallet that will utilize zk-SNARKs as a storage optimization technique.
4. Implement proposed framework using Android/IOS for client and arbitrary programming language for server. The framework should support at least 3 cryptocurrencies.
5. Evaluate the cost and performance of the framework.
6. Propose extensions and discuss limitations of the framework.

Recommended literature:

- Westerkamp, Martin, and Jacob Eberhardt. "zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays." *Contract* 1.2 (2020): 3.
- Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." *Ethereum project yellow paper* 151.2014 (2014): 1-32.
- Homoliak, Ivan, et al. "The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses." *IEEE Communications Surveys & Tutorials* 23.1 (2020): 341-390.

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Homoliak Ivan, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: November 3, 2021

# Abstract

The goal of this thesis is to propose and implement a framework for cryptocurrency wallets. The framework optimizes light client storage and bandwidth requirements in mobile devices. We propose a side-chain mechanism that validates blockchain header chains and creates zero-knowledge proofs. Furthermore, the framework stores the results of proof verification inside an Ethereum Smart contract. The Smart contract supports fork handling and storing header chains for multiple different blockchains. Light mobile clients using this framework can update their local header chain from checkpoints created by the proof verifications results stored in the Smart Contract. This thesis includes an implementation of a mobile wallet using this framework for synchronization with multiple blockchains.



# Abstrakt

Cieľom tejto diplomovej práce je navrhnúť a implementovať prostredie pre kryptomenové peňaženky, ktoré je optimalizované pre požiadavky na úložisko a prenosovú rýchlosť v mobilných zariadeniach. S využitím zero-knowledge dôkazov navrhujeme mechanizmy side-chain výpočtu, ktoré overujú reťazce hlavičiek blokov a uchovávajú dôkazy o ich overeniach v blockchaine. Ľahkí mobilní klienti, využívajúci toto prostredie, už nebudú nútení sťahovať a neustále aktualizovať svoj reťazec hlavičiek, ale môžu používať záchytné body uložené v Smart kontrakte. Táto práca taktiež zahŕňa implementácie mobilnej peňaženky, ktorá používa implementované prostredie pre synchronizáciu s viacerými blockchainami.



# Keywords

Side-chain, Zero-knowledge, Proof, Blockchain, Wallet, Optimisation

# Klíčová slova

Side-chain, Zero-knowledge, Dôkaz, Blockchain, Peňaženka, Optimalizácia


# Reference

SLÁVKA, Samuel. *Mobile Cryptocurrency Wallet Based on zk-SNARKs and Smart Contracts.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ivan Homoliak, Ph.D.

# Rozšířený abstrakt

V tejto práci sme navrhli a implementovali prostredie pre mobilné peňaženky založené na zk-SNARK a smart kontraktoch. Prostredie poskytuje rýchlejšiu a menej náročnú synchronizáciu, ako súbežné používanie viacerých lokálnych ľahkých klientov. Celková optimalizácia úložiska rastie priamo úmerne s počtom použitých Blockchainov, pretože na správne fungovanie je potrebný iba jeden ľahký klient. Klient aj server používajú Ethereum, ako svoj primárny Blockchain, a ako ich jediný dôveryhodný zdroj pravdy. Okrem toho smart kontrakt nasadený na Ethereu slúži, ako dôveryhodné úložisko a na overovanie fragmentovaných reťazcov hlavičiek.

Prostredie taktiež podporuje dva sekundárne Blockchainy, Bitcoin a Bitcoin Cash. Server vytvára lokálne reťazce hlavičiek týchto sekundárnych Blockchainov a generuje dôkazy o ich úspešnom overení. Dôkazy server následne pošle v transakciách do smart kontraktu, kde sa overia a ak sú validné, ich výsledky validácie sa pridajú do kontraktového úložiska. Okrem počiatočného nasadenia je odoslanie dôkazov jediná akcia, ktorá si vyžaduje platbu v našom prostredí. Komunikácia pre klientov je bez akýchkoľvek poplatkov. V tomto prostredí je klient mobilná aplikácia, ktorá spúšťa inštanciu ľahkého uzla v Ethereu. Tento ľahký uzol sa používa na prístup k smart kontraktom so záchytnými bodmi sekundárneho reťazca. Po prijatí záchytného bodu reťazca hlavičiek, klient môže začať budovať nový lokálny reťazec. Tento reťazec začína priamo v tom záchytnom bode a končí v hlavičke, ktorú klient chce overiť. Keďže mobilných klientov zaujímajú len transakcie, ktoré sa ich priamo týkajú, na fungovanie potrebujú len malé časti celého reťazca hlavičiek. Toto prostredie im umožňuje dôveryhodne overovať bloky pri zachovaní bezpečnosti pochádzajúcej z ľahkých klientov.

Okrem nákladov na údržbu a počiatočného dobiehania sekundárnych Blockchainov, výsledné prostredie ponúka dôveryhodnú alternatívu k súčasným mobilným Blockchainovým peňaženkám. Prostredie poskytuje rozhranie na ovládanie smart kontraktu a Zokratess prostredia zo skriptu napísanom v jazyku Python. Naše prostredie sme navrhli tak, aby bolo rozšíriteľné pre rôzne veľkosti postupnosti hlavičiek a pridávanie podpory viacerých Blockchainov, pokiaľ sú založené na dôkazu o prevedenej práci.

Implementovali sme mobilnú aplikáciu v React native, ktorá predvádza funkčnosť tohto prostredia a vykazuje zlepšenia v požiadavkách na úložisko v porovnaní s inými dôveryhodnými alternatívami. Vyžadovaný úložný priestor a množstvo prenesených dat potrebných na mobilnú synchronizáciu pre sekundárne Blockchainy rastie s počtom blokov, ktoré chceme overiť. V najhoršom prípade, ak má klient transakciu v každom jednom bloku v Blockchaine, dosiahne nanajvýš potreby ľahkých klientov.

# Mobile Cryptocurrency Wallet Based on zk-SNARKs and Smart Contracts

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Ivana Homoliaka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Samuel Slávka
May 16, 2022


## Acknowledgements

I am grateful towards my supervisor Ing. Ivan Homoliak, Ph.D. for his professional approach and his helpful insights during this work.

# Contents

# Chapter 1

# Introduction

Blockchain is an ever-growing decentralized database with a single shared state [3]. Its state consists of a chain of blocks, which are used as a basic storage units containing lists of state changes expressed through transactions. Every block must contain a link to its predecessor to be included in the state. These links create the chain that starts at the initial block and ends at the most recent accepted block. The only way to verify information correctness in a blockchain is by recreating its state locally.

## 1.1 Motivation

Blockchain clients want to work with this decentralized state. Therefore, they must store the entire chain locally and constantly update it with newly created blocks. However, with constantly growing blockchain sizes, locally storing their chains is becoming infeasible for smaller devices. For this reason, light clients [28] were introduced. Light clients also build the blockchain locally. However, they only use metadata of its blocks called headers. Therefore their storage and networking requirements are significantly smaller than regular (full) clients.

However, the header chains are also constantly growing, and even this much lighter approach has become too demanding [25] for mobile devices in recent years. For the growing needs of light clients, mobile clients switched mainly to ultralight clients [25] or fully hosted wallets. Ultralight clients utilize various techniques to lighten the load on devices while preserving the security of light clients. They act as light clients underneath while still providing secure access to blockchains. As oposed to another popular apporach in mobile devices called fully hosted wallets. The fully hosted wallets provide an endpoint for all the revelant data, and the client must trust it to be secure. Which trades trustlessness for the gain of usability and speed.

## 1.2 Contributions

This thesis proposes an optimization of mobile light client storage and networking demands by designing and implementing an ultralight client framework. This framework consists of an off-chain mechanism where a server creates zk-SNARK proofs of the header chain validation. These proofs are then submitted and verified in an Ethereum smart contract, where they create a fragmented header chain. The header chain is split into checkpoints made from spaced-out header identifiers. A batch of proofs can create a new checkpoint

only if its headers start with previously-stored checkpoints and the proofs are valid and they continue from some previously stored checkpoint. The smart contract that stores the header chain is fork-resistant and automatically selects the correct strongest chain.

The implemented ultralight client directly accesses the smart contract checkpoints and uses them as temporary starting points for its local header chains. The client verifies the inclusion of the block by querying the smart contract for the closest checkpoint. The client then builds a fragment of the header chain, starting at the received checkpoint and ending at the disputed block. This action is significantly faster than synchronizing the whole chain from its beginning, especially after a prolonged time without synchronization. The resulting framework supports Ethereum as its primary blockchain, and Bitcoin and Bitcoin Cash as its secondary blockchains. However, it is designed to be extensible for other proof of resource based blockchains.

## 1.3 Organization

In Chapter 2 we describe blockchains, their principles, purposes, and use cases. In Chapter 3 we describe zero-knowledge proofs and their variants. In Chapter 4 we propose the design of the ultralight client and its framework. In Chapter 5 describe our implementation of the proposed client and framework. In Chapter 6 we evaluate both implementations and their usability. In Chapter 7 we discuss the advantages and disadvantages of this framework. In Chapter 8 we summarize the achieved goal and propose future developments.

# Chapter 2

# Principles of blockchains

The blockchain is an open system for the execution and processing of transactions under transparent rules [28]. The system is an immutable, constantly growing list of records. Each record is represented by a block that changes internal state of this system. The new state can only be appended to the history of previous states and must reference them to maintain a single chain of events. Blockchains work as decentralized peer-to-peer networks, where nodes synchronize and propagate their local state versions. The system is trust-less, 100% available, and has immutable data storage [19].

Each blockchain node can alter and validate the state of this network. However, the changes must comply with the rules of the blockchain source code. The transaction is a message containing directives for nodes to change state inside of the network [19]. The transactions are validated by blockchain nodes before grouping them into blocks and then broadcasting these blocks to other nodes in the blockchain. Blocks are a fundamental data storage unit that aggregates sets of transactions created roughly at the same time [19]. When a node adds a compromised or incorrect block, other nodes will act based on the consensus mechanism, whose types we describe in Section 2.2.

Each block is linked to its predecessors by including the cryptographic hash of the previous block. This linking creates immutability since the previous state is always part of any new changes. The blocks form a hash chain that contains a cryptographic hash of each leaf node's data labels.

The entire network eventually settles to a single version of its state, bringing trustlessness to its communication. Since every blockchain user can also participate in the functionality of the network, every user can verify every transaction that occurs there. Furthermore, the users do not have to trust anything outside their supervision because all blockchain functionality executes in every blockchain node.

100% availability comes as a result of its decentralization. Because every node can perform all of the the blockchains functionality, there needs to be at least one node for the blockchain to function.

## 2.1 User authentication

A decentralized and open system is prone to attacks and fraud. Blockchain participants are charged for every transaction to discourage most illicit activities. However, the charge must remain completely enclosed in the system to determine the possibility of making such transactions before they occur. For this purpose, user authentication is required to dis-

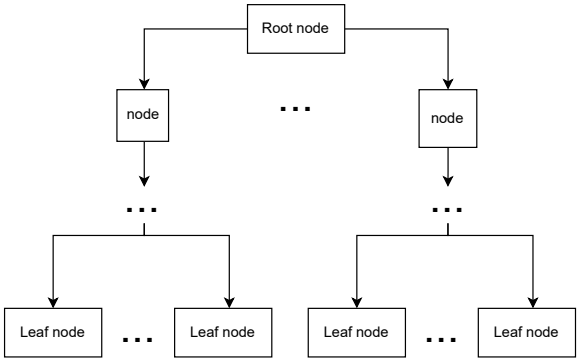


Figure 2.1: State storage structure based on Merkle trees.

tinguish and grant them access to the state. Among blockchains, public-key cryptography serves as a user identification.

In public-key cryptography, each user has a pair of keys [22], private and public. According to the blockchain rules, the user generates a private key during account setup. Through a blockchain-specific algorithm, users generate a private key, which generates a public key. Afterwards, the users can use the public key as their unique address. The address serves for locating part of the ledger that the user can access and utilize. Furthermore, the private key serves as a password, without which transactions on the given address will not work. However, the password cannot grant access to the address directly. Since the blockchain is a distributed system, every other user would find out about the key and gain access to the contents of the address. For this reason, only transactions contain a signature obtained from the key. A signature is the output of a trapdoor function that inputs the sender's private key and the unsigned transaction itself. The final transaction also contains the resulting signature, so anyone with access to the sender's public key can easily verify message authenticity.

Trapdoor functions are also used for public-key generation [18]. The trapdoor function is a function that can be easily computed in one direction and is difficult in the opposite direction [18]. Therefore, computing the public key from the private one is relatively easy. However, the computation of private keys from the public keys is computationally partially impossible.[1] Blockchain security heavily relies on the assumption that the reverse computation will not be feasible [24].

To unequally identify any valid Merkle tree, we must calculate its root nodes' cryptographic hash, and any change in its data contents will propagate directly into the root hash. Merkle trees provide an easy way to verify the data, called the Merkle proof. Merkle proof is a result of recomputed Merkle tree with unverified data [17]. When verifying, we do not need to recompute the whole tree, only the parts that could have been changed [2]. This process speeds up the verification and does not allow other data to be used to build the proof. After re-computing, we only need to check if the newly created root hash equals the expected one.

[1] With current resources and knowledge [23].

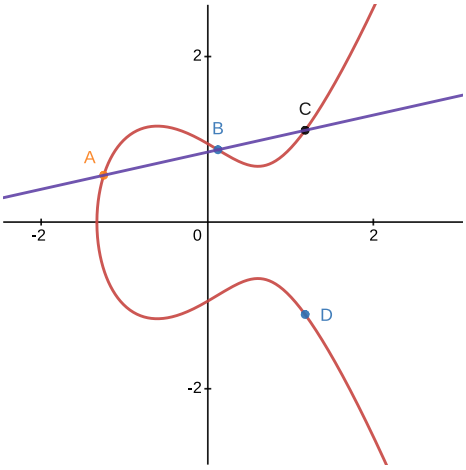


Figure 2.2: Points A and B create point C, which is reflected into point D [23].

## Commonly used signature algorithms in blockchains

Blockchains have different limitations and requirements, so their creators need to choose suitable algorithms for singing. Due to its decentralized nature, on-chain resources are scarce. Furthermore, signing, one of the most used algorithms, needs to be secure and efficient. The following are algorithms currently used on the most prominent blockchains [24].

### Elliptic Curve Digital Signature Algorithm (ECDSA)

An elliptic curve is a set of points satisfying an equation with two variables, having one in a degree of two and the second in a degree of three [23].

$$y^2 = x^3 + ax + b$$

The elliptic curve has a unique property. Given any two non-vertical points, the line between them will intersect the curve precisely one more time. Furthermore, on the reflection of the intersection point on the x-axis, a new point will be created, which can be seen in Figure 2.2.

Keeping the first point and redrawing the line with a new point once again, at the x-axis reflection of intersection, creates a new different point, as can be seen in Figure 2.3.

We can repeat this action any amount of times and what will remain are the first point, the final point, and the number of actions to reach the end. This action creates a trapdoor function because finding the number of actions when only the first and last points are available.

Because computers are more efficient with natural and relatively small numbers. The curves in ECDSA are in blockchains represented as a finite set of natural numbers on the curve. Furthermore, it wraps them into a given range [23].

To apply this algorithm to blockchains. The first point is the user's public address, the last point is the signature, and the number of points is the private key. The current versions of Bitcoin and Ethereum both use this algorithm. However, it also has its disadvantages. They are prone to bad or compromised random number generators. Furthermore, there is no efficient way of compressing and verifying multiple signatures together [24].

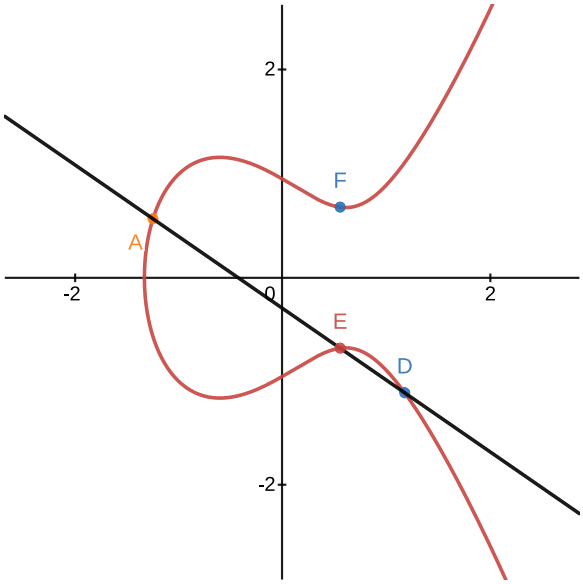


Figure 2.3: Points A and D create a new point E, which is reflected into point F [23].

### Schnorr signatures

Schnorr signatures are one of the solutions to scaling problems and security concerns proposed in the Bitcoin blockchain. It is a variation of ECDSA, and often blockchains using Schnorr signatures utilize the same curves as in ECDSA chains [4].

Schnorr signatures enable aggregating multiple signatures into a single verifiable signature natively. Aggregation is also possible in ECDSA. However, the addresses of participants needed to be known, which is inefficient and disallows any form of privacy. In the Schnorr algorithm, the aggregated messages are indistinguishable from the regular ones and do not lose any security [20].

When a Schnorr signature algorithm uses a sufficiently random hash function and under the assumption that the elliptic curve discrete logarithm problem is hard, it was formally proven that breaking Schnorr signatures is as hard as solving the discrete logarithm problem [21]. In contrast to ECDSA, which has not been proven to have any specified hardness.

### Pixel signatures

Pixel signatures allow grouping multiple signatures into a single trust-able, efficiently verifiable signature. A posterior corruption problem occurs when multiple nodes have corrupted signing keys. When the number of corrupted nodes within network specification, it should be able to cope with them. Since the network is dynamic, if those nodes have been in the network long enough, a fork can be created such that they have the longest chain [9].

Pixel signatures are forward-secure signatures, which means they solve the posterior problem. They force the nodes in the network to periodically change their private keys. After a secure key corruption, the key cannot create a false chain history.

### Boneh-Lynn-Shacham (BLS) signatures

BLS signatures utilize bilinear pairing for verification of signatures described in Section 3.1. BLS scheme consists of key generation, signing, and verification. The key generation algo-

rithm selects a random unsigned integer $x$ as the private key, creating the public key $g^x$. Signature is the output of the hash function of some message $h = H(m)$, where we put it at the power of the private key $h^x$. Verification is confirming whether the results of the bilinear pairing are equal [5].

$$e(h^x, g) = e(H(m), g^x)$$

Ethereum 2.0 uses BLS because of its ability to aggregate multiple signatures and provides smaller and easier-to-calculate signatures. BLS signatures are also built on top of the ECDSA principles and offer better scaling through aggregation of signatures [5]. However, they are not quantum safe and should be replaced by zk-STARK-based aggregation, described in Section 3.3. However, they still suffer from the posterior problem [9].

## 2.2 Consensus mechanisms

Consensus mechanisms allow us to determine and guarantee the current state of the network. They ensure the processing and universal acceptance of valid transactions between honest nodes. Furthermore, they try to eventually gather all block proposals in the network into a single sequence of blocks [13]. Since coordinating the whole network of nodes to select one block is impractical, there are multiple ways to reduce selection without affecting security.

One of the more popular approaches is the use of lottery-based protocols [3]. The lottery-based protocols randomly select nodes from those proposed blocks and create consensus only between this subset of proposals. The problem with selecting multiple nodes is that they often have different blocks proposed, which creates a temporary split in the blockchain called fork [13]. A more thoughtful description of forks is in Section 2.3.

Another approach is to use voting-based protocols. Where votes decide the state of the participants, they offer much higher resistance to forking. However, they introduce many problems with network scaling [13]. These two approaches are often combined to ensure scalability and security [13].

### Byzantine fault

Byzantine fault is a condition in a system, where important components can fail and information on their failure is unreliable. It is based on the hypothetical problem that Byzantium has many armies and many generals. Generals must consent to a single joint action while communicating only with messages. Since some generals can be impostors, they can create false messages for different generals. Miscommunication from those messages could lead to part of the generals attacking and others retreating, especially if the impostors' vote decides the majority. Byzantine fault tolerance happens if loyal generals have a majority agreement [8].

Blockchains solve the problems of impostor participants in multiple ways. The most common is requiring some investment behind the directions from participants. If the general has something to lose from his command, the more significant the loss, the less likely they will betray other generals. Furthermore, if the importance and positions of the generals are selected based on the height of their investment, the generals themselves would always have to lose the most from bad decisions [8]. In blockchains, this investment is some real-world action that is unambiguously verifiable on-chain or staking their blockchain resources behind their decision.

### Proof of resource

Proof of resource is a widely adapted and tested consensus mechanism in blockchains. In proof of resource, the main component is an expenditure of some resources without profit outside the validated blockchain. The resource is external to the blockchain system. However, there must be a way to create verifiable proof to prove its expenditure. All nodes decide the resulting sequence of blocks in the blockchain.

This expense during validation can guarantee that nefarious actors will lose resources during an attack on the network. In this mechanism, the attacker would need more than 50% of the blockchain resources to succeed in an attack. Miners are the participants in block validation action [26]. The chain incentives miners to extend work in two ways. First, the miners receive fees added by transaction creators. Miners can choose which transactions to process based on the transaction fee size. This choice enables users to speed up transaction processing by raising the fee value. Then, as a second incentive, some blockchains mint new tokens and distribute them to the final block miner during block creation. The minting process makes mining more attractive for earlier stages of the blockchain [19]. However, the final block creator is only a single entity, and the chance of becoming this creator is minimal for regular miners. Therefore, miners pool their resources and distribute the resulting rewards through the pool proportional to the expended resource.

Mining is a different process for each blockchain. Most of the differences are in the type of problem that the miners solve and how its difficulty changes over time. It consists of miners collecting transactions, validating them, and bundling them into blocks. Then miner starts to produce proof. Miners need to expend sufficient resources for a block to be accepted. Only the block with the most resources accumulated is correct in the final state of the blockchain.

The main disadvantage of proof of resource consensus mechanisms is that the work produced during validation grows with the network and has an ecological impact. The resource expenditure does not produce anything of value outside of blockchain security. Furthermore, if a single entity were to amass over 50% of the blockchain validation power, the entity would receive full power to modify the whole state of the blockchain.

### Mining in Bitcoin blockchain

Proof of work in Bitcoin is a 256-bit number that results from double-SHA256 of data in the blockchain. This number must be smaller than the current difficulty set for the whole blockchain to prove that work investment is sufficient.

Since SHA256 is pseudo-random from its design, changing the nonce property in the block unpredictably changes the function output. Miners need to guess the nonce so that the hash fits the required difficulty [19]. This function allows mining using application-specific integrated circuits (ASICs). Since the SHA256 algorithm is not resource-intensive, devices specialized for parallel execution are much more efficient than general-purpose devices.

### Mining in Ethereum blockchain

Mining in Ethereum was created to be accessible to all participants. The network rewards miners and should be easily verified even by light clients [28]. The chosen proof of work function is difficult to optimize in specialized hardware, to achieve accessibility for all users. The mining function requires high memory bandwidth, thus disabling the parallelization of proof-of-work computation since the bottleneck is available memory, not the processing

power. Proof of work in Ethereum starts with an extensive semi-permanent data set. The data set is then randomly sampled to create a proof. Regenerating parts of the data set servers as proof verification. The data-set size requires a large amount of memory in active use. Therefore, parallelization or ASICs are mostly ineffective.

#### Proof of stake

Staking is a process of locking funds for the exchange of blockchain authority. Slashing is used as a security measure. It permanently removes some of the staked founds after false block propagation [28]. This mechanism omits outside of blockchain expenses and purely focuses on blockchain inactivation. This mechanism is based on game theory and expects that most participants will work in their favor.

#### Proof of authority

It is a widely used mechanism, differentiated from others by utilizing delegation of votes. The owners of the authority are responsible for the blockchain and have access to its inner workings. This mechanism allows for the creation of new data for the current authority. This mechanism can easily be swapped instead of proof-of-work, while the rest of the chain remains identical. Authority manipulation is helpful for testing purposes.

## 2.3 Transaction validation in proof of work blockchains

Mining is a process dedicating effort to promoting transactions assembled into batches called blocks. Those blocks form the Merle tree, which connects them to the previous state of the blockchain. Miners are nodes connected to the network that receive broadcasted transactions based on the miner fees included in transactions. They choose ones to include. The fee will be for miners after the transaction's block has been included. Each block contains its identification, located in the context of the whole state, both parents' hashes, and the state hash after all transactions have finished their execution. Furthermore, a hash has been generated based on current difficulty, and the blockchain algorithm is proof of work-based blockchains. This hash serves as an investment by the miner in the block and will be lost or unrewarded if the block is rejected [19].

Forking is an event when there are multiple different blocks created concurrently. The choice of the main one falls onto the consensus mechanism described in Section 2.2. In proof of work, consensus is the block that has accumulated the most work behind itself. Every other block is later called an orphan block and is no longer valid. In each blockchain, there are multiple temporally valid states that will eventually resolve to a single final state [28]. The amount of work invested decides the chains' eventual state, and by measuring the length of trees that make blocks, it is easy to determine the longest and also the most propagated state [26].

#### Network incentivization

Miners are motivated to participate by rewarding them with native tokens proportional to their work. Those currencies are unique and separate for each blockchain. There are many variations in the way currencies are stored and distributed throughout the network.

There are multiple ways to differentiate users within blockchains. Below we will describe users identified in the most prominent blockchains by their market capitalization.

#### Account balance based state

Account-based state representation creates accounts that contain a balance of currency. Addresses serve to identify accounts, which can transfer any amounts between themselves, as long as it is available. The account-based state is common in many blockchains, including Ethereum. The global blockchain state uses the Merkle tree as its data structure. Each block has a hash stored inside of it as a state root [6], identifying the state during the block creation. The state contains account balances, contract storage, contract code, and account nonces under each account. The account structure is stored in blocks. The accounts are leaf nodes of Merkle tree containing state which build into root node calleblockchaind State root. This root node can be used to verify the given state through Merkle proofs.

#### Unspent transaction outputs (UTXO) based state

UTXO represents any amount of digital currency that is the output of a bitcoin transaction [12]. They cannot be split into smaller amounts by themselves. However, after performing a transaction with a larger UTXO than required, new UTXOs are minted from the remaining UTXOs [19]. UTXO is also common in many blockchains, including Bitcoin.

## 2.4 Smart contracts programming languages

Various blockchains are using different programming languages for smart contracts as can be seen in Section 2.4. Any peer-2-peer network able to process transactions and store a particular state can be considered a blockchain. Nevertheless, this has minimal functionality, as users can communicate only in the most direct way without any complications or improvements.

Therefore, adding functionality to clients directly affects the data stored inside. Creating a blockchain-specific language adds client functionality and access to the blockchain state, allowing program writing to be executed by transactions outside the blockchain.

The languages vary heavily according to their language capabilities. Each participant's program execution needs to run with the same result, and it is costly to process more complex programs. Furthermore, it is not always beneficial or usable to have a powerful language. Some blockchains like Bitcoin offer just basic scripting functionality directed at transaction manipulation. However, some blockchains, such as Ethereum, provide a general computing network. This network can provide an interface for processing critical or transparent functions [28].

From this point on, we will primarily focus on blockchains, whose languages are touring complete in terms of their processing power. Touring completeness is a necessity for more generalized computing.

## 2.5 Blockchain client types

In terms of blockchains, the client is a participant in information exchange. Clients do not need to participate in security, not even receiving data from the chain, except when they decide to initiate communication. They are only concerned about their own transactions and those with whom they interact. There is no real benefit for them to keep other data, except for security. Continuing from here, we will describe the types of client inside the Ethereum network. Other blockchains use different terminology, but the underlying func-

tionality remains very similar. Clients are implemented based on chain specifications, and the development teams often differ from core blockchain teams.

### Full clients

The full client validates the entire block of transactions. The client builds Merkle trees from blocks and, when validating, recalculates the tree with the suspicious block. Full clients always store the entire state and participate in network actions. Full nodes verify that blocks and states may or may not provide access to headers for light clients. However, they need to provide other data on request to participate [28].

### Consensus clients

Apart from thin clients, full clients can actively participate in blockchain growth. Because they must possess copies of the whole network, they can act by a consensus mechanism and try to append new blocks [13].

### Thin clients

The thin client, also known as the light client [28] or simple payment verification (SPV) [19], is a client that uses only headers of blocks to validate the state of the blockchain. They store the chain of these headers but also need to request data from other blockchain participants when creating transactions. The verification of data provided is against state roots in their headers. This verification requires significantly smaller performance and storage capacity. The headers build the Merkle tree [17] and, similarly to full clients, verification is done by creating Markov proofs with the inserted changed disputed state root [28].

The primary purpose of the creation of thin clients was to allow mobile or less resourceful divides to participate in the network. However, even header chains have become impractical in less powered or network-constrained devices in more popular blockchains.

### Archiving clients

Archiving clients store everything that the full node stores and its historical changes. Archiving nodes require much more storage than full nodes. However, bringing more security and reference points when forking is required. They act as full nodes from an outside perspective [28].

### Ultralight clients

With the growing size of the blockchain, thin clients ceased to be viable solutions for mobile devices as solution ultralight clients were created. There are many approaches to optimizing synchronization time and resource requirements [25].

### Examples of clients in Ethereum

Most client implementations offer multiple types of clients since the difference is technically slight. In Ethereum, one of the most prominent clients is Geth. It is written in Go-lang and provides an API interface to communicate with the blockchain. Geth offers a full node that acts as an archiving and pruning, which stores less data. Geth[2] also offers thin clients and

[2] https://geth.ethereum.org

connections to most testing Ethereum networks. Geth also offers to mine. However, it is only CPU-based, which is significantly slower than other implementations with GPU-based mining, and Therefore, it is the most useful for test-net applications.

OpenEthereum, written in Rust, is designed for reliable deployments. It also provides an API interface to the blockchain, but uses Warp Sync, which significantly shortened the synchronization time[3].

Miner clients collect raw transactions into blocks and execute work to prove their investment in their block. If the block is accepted, they receive compensation. Mining clients usually have separate implementations, as their primary goals are vastly different from other clients. The most prominent mining client currently is EtherMine[4] which allows faster GPU-based mining.

## 2.6 Blockchain wallets with examples

A wallet is a means of storage of private keys. The wallet can be just a piece of paper, but that would be quite unsecured and difficult. Therefore, multiple types of wallets were created with varying amounts of security and use cases. Wallets can be differentiated by the location of private keys, into Self-Sovereign wallets and Hosted Wallets [13].

- Self-Sovereign wallets store the keys locally and should never expose them to the internet. They utilize these keys to communicate with the blockchain directly. Depending on the wallet, the keys are stored using software inside the user's computer (e.g., Exodus[5]), or they are separated with hardware(e.g., Ledger[6] ) [13]. A hardware-separated wallet is a specialized device that securely stores private keys without connecting to the Internet. They are one of the most secure wallets, frequently in the form of a USB with an encrypted private key and application to communicate with the key.

- Hosted wallets utilize a third party that provides access to a wallet. The wallet is located inside the user's browser (e.g., Metamask[7]) or inside providers server(e.g., Binance[8]). They are possibly less secure since they provide more incentive for attackers because they require trusting a single centralized entity and creating a single point of failure with key storage.

### Wallet data validation

Wallets depending on the type, use different types of state validation. Hardware wallets are validated inside specialized programs made for decryption of stored keys. They can be using any node for state validation. However, full nodes, even thin nodes, are too resource intensive to use on personal devices. The resource limitation increases in mobile wallets. Therefore, currently, most commonly used mobile and hardware wallets use centralized providers. On the other hand, online wallets are entirely run on servers and can easily afford to run full nodes. Furthermore, as such, it brings the most reliable data.

[3] https://openethereum.github.io/
[4] https://github.com/ethereum-mining/ethminer
[5] https://www.exodus.com
[6] https://www.ledger.com
[7] https://www.metamask.io
[8] https://www.binance.com

# Chapter 3

# Zero-knowledge proofs

Zero-knowledge (ZK) proof is a method of providing information without trust. It allows for the transfer of knowledge of information without providing the information itself. Although the provided statement should be a verifiable proof, the verifier should reliably distinguish between false and true statements [15]. ZK proofs are not only limited to information. They can also provide proofs of the computation without knowing the computation or its results [29].

## 3.1 Zero-knowledge proofs for polynomial expressions

Polynomial expressions are the foundation of zero-knowledge proofs. They are all expressions in the following form:

$$c_n x^n + ... + c_1 x^1 + x_0 x^0$$

Where $c_n, ..., c_0$ are coefficients having constant values, $x^n, ..., x^1$ are unknowns, and $n$ is the polynomial degree. The polynomial expression has the attribute that any two polynomials of the nth degree can have at most n intersections, which implies that a polynomial value in the nth degree can indisputably identify the polynomial. Under the assumption that the prover will not use any other means of getting the resulting values, we can create a simple zero-knowledge proof for knowledge of polynomials [15].

If a verifier and a prover have a knowledge of a polynomial, the verifier can ask for a value at some point. If the returned values are equal to those verifiers calculated in their polynomial, they can be confident that the prover knows the polynomial without providing it to the verifier [15].

However, this protocol does not entirely check the knowledge of polynomial but rather the knowledge of vales at a given point. Furthermore, participants exchange knowledge of the polynomial, which is actually the exchange of knowledge of coefficients in the polynomial.

### Information obfuscation

To improve these problems, we need to obfuscate the data. Information obfuscation has multiple approaches. We are using homomorphic encryption, which utilizes expressing values as chosen base values to the power of value we want to encrypt. However, the base value is public, and it is pretty easy to reverse this operation. Therefore, modular arithmetic is also applied. It utilizes wrapping values into some limited scope with *modulo* operation.

The wrapped values, if sufficiently large, are infeasible to reverse, and the wrapped values, except for multiplication, preserve the arithmetic properties of the unwrapped ones. This preservation is the reason for choosing homomorphic encryption for polynomial wrapping [29].

To express an encrypted polynomial, we need to express each of its coefficients tied to the unknown in place of the powers of a base. Therefore, the resulting value will be in the following form:

$$base^{c_n x^n + \ldots + c_1 x^1 + x_0 x^0}$$

Furthermore, this expresses the value of the encrypted polynomial. Prover, in this exchange, will provide the encrypted polynomial and provide two other polynomials. The first divides the original polynomial, and the second represents the result of the given division. The verifier checks whether the first polynomial of the prover is a cofactor of the secret one. Since the secret used to evaluate these polynomials is different for the prover and verifier, it provides security for the entire exchange.

However, the prover can still use other methods to get proof. When the exponent is too tiny, brute force guessing of the polynomial is currently feasible. To force provers to create their proofs using exponentiation. This method is called the knowledge-of-exponent assumption. It utilizes sending another value together with the verifier's encrypted polynomial. Furthermore, a random amount shifts the other value. Exponentiation performs this shift and consequent *modulo* operation on the result. Provers must exponentiate both the shifted and the encrypted values with the same powers. The verifier can then quickly check whether the values are correct and trust that the prover did not use other means of getting the result.

### Secure proof validation

We can utilize bilinear pairings to allow grouping of multiple separate proofs into one verifiable proof. It is a function that bilinearly pairs two encrypted inputs to single encrypted output represented as their multiplication.

$$e(base^a base^b) = e(base, base)^{ab}$$

However, these functions can only take two proofs as input and not pairings, as they are in different domains. This problem can be solved by differentiating the domains of the functions and performing the pairing function on the specific domains. Therefore, given four encrypted inputs, we can pair them to two pairs of proofs and once again pair those pairs to achieve a single pair of pairings. This method can allow secure verification proofs because a multiplication operation is needed to check whether the same value indeed exponentiates the proofs provided by the prover as the encrypted polynomial. The resulting pairing is usable even with different base domains due to the properties of the elliptic curve described in Section 2.1. The reason for the multiplication requirements is the initial shift applied to the secret value [29].

$$e(base^{f(s)}, base^{shift}) = e(base, base)^{shift * f(s)}$$
$$e(base^{shift * f(s)}, base) = e(base, base)^{shift * f(s)}$$

Verifiers will be provided withing proof $base^{shift}$, and if they have the polynomial $f(s)$, they can compare their pairing to the proof containing shifted encrypted pairing.

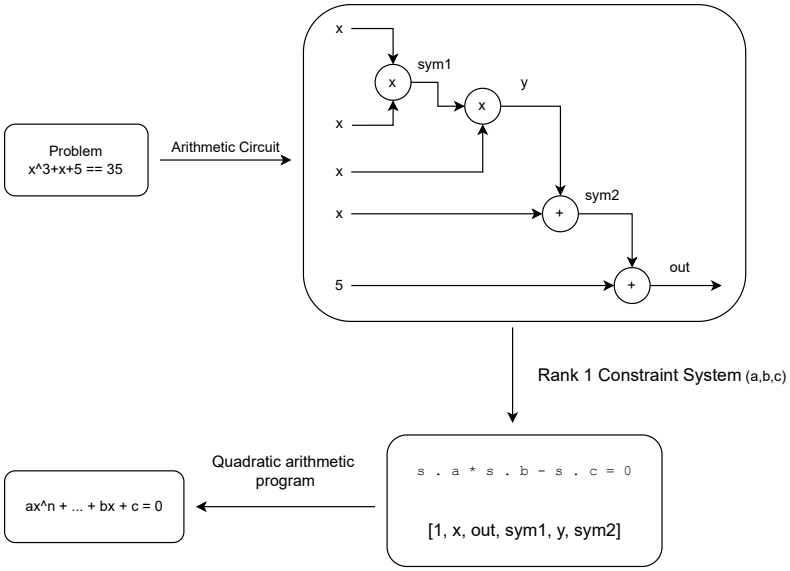


Figure 3.1: Transition of a computation to zk-SNARK.

### Zero-knowledge

The result is a zero knowledge proof, however, only from the point of view of the prover. The verifier can still extract knowledge from the two polynomials sent by the prover. To combat this, shifting the values is also used on the prover's side.

### Non-interactiveness

The obvious problem is that we require a single trusted entity to create a setup completely randomly and not to store the secrets. One way to solve this problem is to create a composite trusted setup involving multiple parties. Here, each participant will create their setup, and the resulting used one is a combination of all those involved. The cryptographic pairing described above creates this combination. Moreover, we use the resulting pairing of all the individual setups as the final setup, which enables the creation of a setup that only requires a single party from the creators to be honest and delete their secrets.

## 3.2 Zero-knowledge proofs of computation

To prove the execution of a computation, we first need to express the computation function in polynomials. However, first, we must restrict the problems to only the problems that polynomially expressed functions can calculate.

NP-complete problem is a problem within an NP class to which every other problem in the NP class is reducible [14]. As an NP-complete problem, SAT is reducible to the problem of evaluating polynomials [29] therefore, the polynomial evaluation also belongs to the NP class and by itself is NP-complete. Furthermore, polynomials can express any problem within the NP class.

The process of expressing operations through polynomials is shown in Figure 3.1. By adding variables as constants to polynomials, we can create simple logic components that build equations describing the function. Moreover, to turn it into a proof of computation,

we need to include the result in the proof. However, simply including the result does not prove that it came from the computation or that it is correct. Instead, we need to include the polynomial of the operation. The prover will create three polynomials for each operation, left side $l(x)$, right side $r(x)$, and result $o(x)$, where $l(x)$ *operation* $r(x) = o(x)$. The prover will use $l(x)$ *operation* $r(x) - o(x)$ as the polynomial for calculating the proof.

In this way, while verifying the knowledge of a polynomial, we are also verifying its correct execution and the knowledge of the operation. If the result, the polynomials, or the operation were wrong, it would be easy to discern using the algorithm above.

## 3.3 zk-SNARKs in blockchains

Zero-knowledge succinct noninteractive arguments of knowledge are a specific variation of zero-knowledge proofs. In the previous sections, we describe the functionality of their properties. However, not their eventual results on communication.

- Zero-knowledge allows participants to communicate without revealing any knowledge about the information, just that of their possession. In blockchains, providing only the result of computations or transactions reduces the required space and computing power.
- Succinctness means communication in relatively small amounts of data per information. Polynomials do not have to be entirely verified, only their encrypted values at specified points. As the most expensive part of blockchains is storage [3], reducing the size of the message is always beneficial.
- Non-interactiveness means the ability to transfer knowledge in a single message without interaction between participants. Messaging every participant is expensive on distributed systems. Therefore, zk-SNARKs provide all the required information with a message.
- Trustless communication allows for omission of introductions and exchange and verification of credentials. Because participants do not need to trust other sides, they need to check messages.
- Arguments of knowledge are transferred rather than the knowledge itself. The argument states knowledge of knowledge without requiring the transfer of knowledge itself. Which once again brings more minor storage requirements.

As mentioned in Chapter 2, decentralization also has many disadvantages, and one of the most prominent is the cost of computation. Zk-SNARKs allows offloading the computation outside of blockchains while keeping trust in the results.

## zk-STARKs

Zero-knowledge scalable transparent arguments of knowledge are a post-quantum secure variant of zero-knowledge proofs that do not require a trusted setup [11]. A trusted setup is the cornerstone of SNARKs security. If it were compromised, anyone with the secret and shift would be able to create fake and valid proofs [1]. ZK-STARKs try to solve this problem by introducing public randomness.

Another significant advantage is their resistance to quantum computing. SNARKs security is built on the assumption of the hardness of discrete logarithm in elliptic curve groups, which are potentially vulnerable to quantum computers [1]. However, zk-STARKs are built on assumptions of the existence of collision-resistant hash-functions and shared access to a random function [11]. So far, these assumptions have been considered post-quantum secure [11].

### Bulletproofs

Bulletproofs is a slower version of zk-snarks that does not need a trusted setup and is post-quantum secure [11]. Their proof size with more complex proofs is significantly larger than that of SNARKs and STARKs [11]. Bulletproofs is an extension of the Bitcoin blockchain that allows private transactions. Instead of relying on elliptic curves as in SNARKs, they only rely on the discrete logarithm assumption [7]. Private transaction hides the amount and participants but is still fully verifiable by their proof. Bulletproofs also support proof aggregation [7].

## 3.4 Use-cases of zk-SNARKs

Considering the advantages of zk-STARKs and bulletproofs, the reason most real-life use cases in the zk proof blockchain choose SNARKs comes from another disadvantage of blockchains, which is storage [10]. The proof size of SNARKs and Bulletproofs is in terms of bytes, whereas in zk-STARKs it is in tens of kilobytes. The verification time is in terms of milliseconds with SNARKs and STARKs, but this time among Bulletproofs it is in terms of seconds. Therefore, even with all the disadvantages of SNARKs, it remains the most helpful tool for the current limitations of blockchains [10].

Zero-knowledge proofs have many use cases where their inherent properties provide many advantages. The most common are authorization, validation of private data, outsourcing computation, and anonymizing transactions [15].

### Off-chain computations

Off-chain computations are offloading the execution from the blockchain and creating proofs of its correctness. The proof of the computation must be universally trusted, and its verification must be much easier than the computation. They are a solution to the scaling problems of blockchains [27]. Zero-knowledge proofs are a secure and efficient way of implementing them.

An example of an off-chain computation is a chain relay. The chain relay links blockchains that can securely transfer data between them. Program zkRelay implements header verification in an off-chain mechanism and only stores proofs of this verification to blockchain [27]. Utilizing on-chain computation would be 187 times more expensive [27]. In the zk approach, zkRelay acts as an intermediary light client that validates headers from the source blockchain and creates a proof of this validation. The target blockchain stores the proof in a smart contract. The target blockchain can securely verify the transactions of the source blockchain.

### Blockchain privacy

By design, blockchains are public ledgers where private transactions can not be trusted and are traceable. Private transactions need to be easily verifiable and still private. ZK proofs allow doing precisely that.

An example of private blockchains is Zcash. Zcash is a private cryptocurrency based on the bitcoin codebase. All transactions are transparent but can shield with zk-SNARKs, anonymizing them.

### Blockchain multi-layering

Zero-knowledge proof enables offloading of computations outside of the network. This technology can also create a second layer of blockchains built on top of the original's security. While it provides some extended but cheaper functionality. The price and required storage space drop drastically by storing the higher levels of blockchain in a compounded form inside the original. An exmaple of implemented multi-layering is zkSync[1]. Where zero knowleadge proofs are used for batching, validating and executing transactions in a layer separatle from the main chain.

### Implementing zk-SNARKSs

Zero-knowledge proofs variants are complex algorithms. However, securely implementing them to accept general computations would be complex. Some frameworks provide high-level interfaces to utilize zk-SNARKs to decrease their difficulty.

#### Zokrates

Zokrates[2] is a toolbox for using zk-SNARKs on blockchains. It facilitates trusted setup in both single-party and multi-party computations during setup phase. It provides its domain-specific language(DSL) that, after execution, leaves a trace. This tracing of Zokrates transforms into polynomial expressions that serve as parts of the proofs [27]. It also facilitates blockchain integration by generating verification contracts.

Zokrates provides exportation of proof verifier directly into a solidity smart contract, which is supported in Ethereum blockchain. For this reason, we chose Zokrates as a tool for creating proofs in this framework.

#### Libsnark

Libsnark[3] is a c++ library that provides a programming framework for zk-SNARKS. It also contains implementations of several NP problems. It provides a high-level approach to zk-SNARKs implementation. However, it also contains access to their low-level functionality. The framework does not directly provide access to blockchains. Its gadget libraries help with circuit specification [16].

[1] https://zksync.io/
[2] https://zokrates.github.io
[3] https://github.com/scipr-lab/libsnark

# Chapter 4

# Framework design

The resulting framework comprises of four functional components: client, server, zokrates verifier, and blockchain smart contract. These components interact with two types of blockchain, Primary and Secondary. The primary blockchain stores and executes the smart contract, and the secondary blockchains serve as data sources for both the client and the server. We decided to use Ethereum for its smart contract capabilities for the primary. We chose Bitcoin and Bitcoin Cash for the secondary blockchains because both are proof of work-based and quite similar in terms of structures and functions.

In Figure 4.1, we can see the flow of framework functionality. In the first step, the server downloads batches of headers and validates them. The server then creates ZK-SNARK proofs of the header chain validity in the second step. The third step is publishing, which the server executes through a smart contract public method call. This call requires the server to provide funds to validate and store new data. The smart contract is deployed in The primary blockchain and is responsible for validating ZK-SNARK proofs. If they are valid and the header batch starts with an already saved and validated header, a new batch is created and appended to the header chain stored in the Smart Contract. The contract can accept an arbitrarily long set of proofs of header chain validations. These proofs are the output of the Zokrates toolbox that receives parsed input from the server. The smart contract header chain does not contain every header; only hash, position, and difficulty for selected headers. The space between the stored headers depends on the size of the incoming batches and the number of those batches.

Both the client and the server connect to the blockchain for their data source. In the fourth step, the client queries secondary blockchains for clients' transactions. For validating those transactions, the client needs to build a local header chain to the blocks that contain them. The local header chain starts after the fifth step by first receiving the closest checkpoint to the queried block. From the nature of smart contracts and zk-SNARKs, the client can be confident that every header stored there is in the main chain of the blockchain. Furthermore, the client can assume all the information received from the smart contract is truthful. The client builds a local header chain starting from the received checkpoint header in the sixth step. If the chain builds successfully, the client queries the secondary blockchain for proof of transaction inclusion into the block in the seventh step. The client will validate the inclusion proof locally, and if successful, the client can be confident that the transaction is included in the secondary blockchain.

Zokrates exports the verifier into a callable smart contract from the main smart contract. Verifies the validity of 32 consecutive headers and that they follow the correct predecessor. It validates the hash, target, and constructiveness of headers.

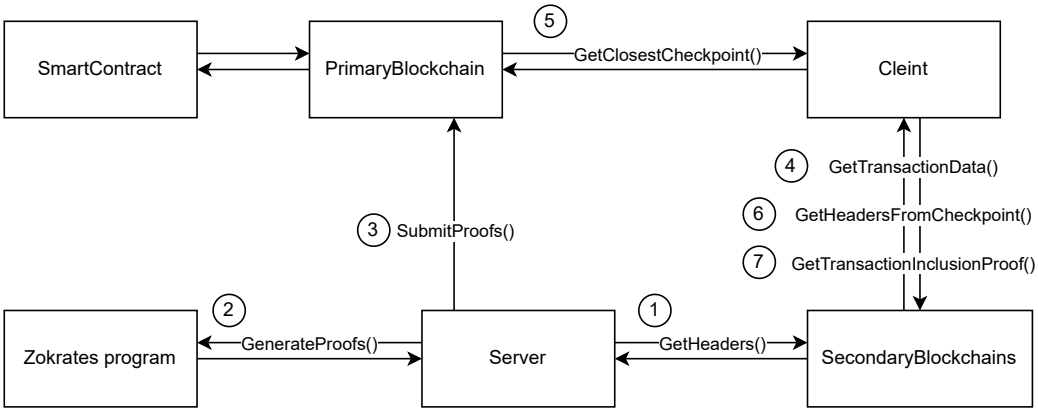


Figure 4.1: Framework action flow.

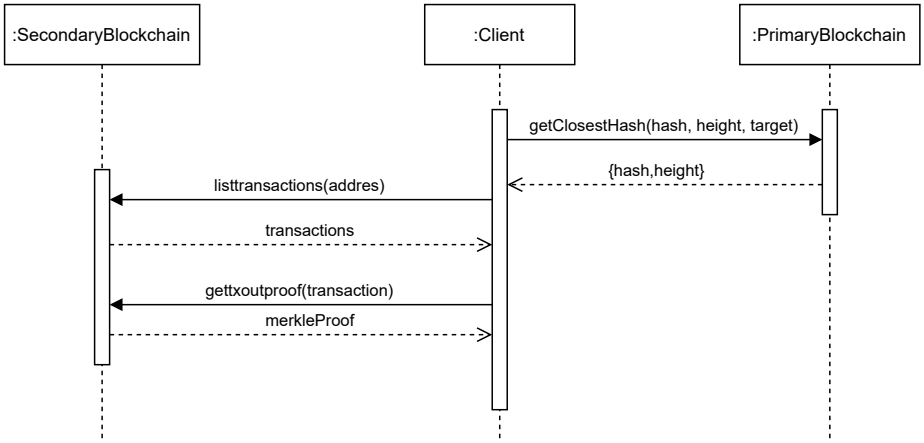


Figure 4.2: Transaction creation and validation by the client.

Each secondary blockchain needs separate storage on the server and a smart contract. Validators can still work for blockchains that are similar or are forks of each other. They need to use the same data structures, hashing functions, and consensus mechanisms. An example of such blockchains is Bitcoin and Bitcoin cash. Both are similar on the SPV client level, only with different block values and transactions. However, blockchains, such as Litecoin, will require a new custom validator because of different hashing functions. We chose this framework as the primary Ethereum blockchain, where the smart contract will be deployed, and for secondary blockchains, we chose Bitcoin and Bitcoin Cash.

## 4.1 Client

The client can validate the inclusion of the transaction in the blockchain described in Figure 4.2. When validating a transaction, the client first builds the local part of the header chain. The local header chain always starts with a checkpoint received from a smart contract or some previously validated header. This local chain build-up consists of light validation by recalculating hashes, difficulties, and pointers to previous headers. After the local header chain reaches the block, which supposedly contains chosen transactions, the client can be confident that the block is in the main chain. The client then asks a full node

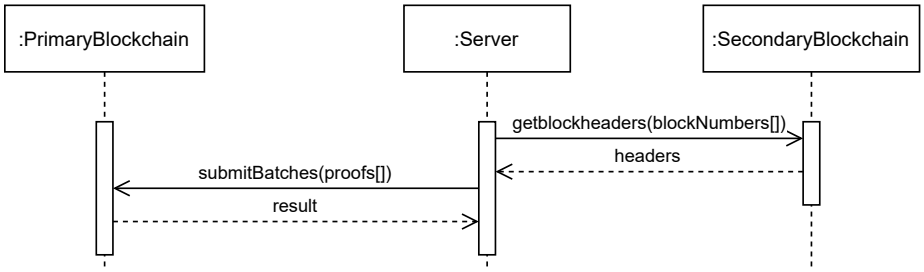


Figure 4.3: Creation and storage of proofs by the server for one blockchain.

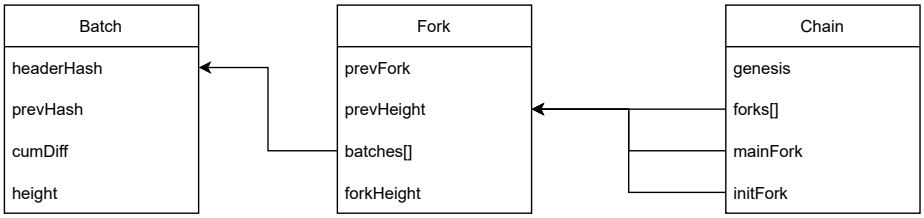


Figure 4.4: Data representation within the Smart contract.

for a Merkle proof of inclusion of a transaction in a block. By validating the proof, the client can be confident that the transaction is in the block and in the blockchain.

## 4.2 Server

The server also connects to the primary and secondary blockchains. The primary blockchain is used for proof submitting and the secondary blockchains for header gathering. The server builds local header chains of the secondary blockchains and validates them in batches of fixed length. The server is also responsible for contract and zokrates management. It deploys, updates, and submits data to the contract. Given the high limitations of the Zokrates toolbox, the server needs to specifically parse data to be easier to parse in contract and later in the Zokrates program.

## 4.3 Smart contract

The smart contract is used as a transparent and trust-less data source. The client connects to the master blockchain that contains the smart contract for storage and validation. Smart contracts on the Ethereum blockchain store multiple representations of other secondary blockchains. Each chain is composed of forks and sets of batches.

The Smart Contract at the top level contains a single mapping of blockchains to their predefined IDs. The smart contract can only verify and build header chains in proof of work-based blockchains. Verifying proof of stake or other consensus mechanism-based blockchains is possible, but much more complex to implement. The smart contract can automatically select the main chain and handle forks and attacks. Provides two functions. The first needs some gas to execute and is for submitting an arbitrarily long array of batch validation proofs, and the second is for gathering the closest validated header to a given block height. The second function is a simple call that executes in EVM(Ethereum virtual machine) without the need for any payment. Returns the closest valid header for the given height.

**Smart contract security**

Any user can submit batches to the smart contract and build their blockchain. However, every batch has calculated cumulative difficulty, and the strongest chain is selected based on its trough batch accumulation. The attacker would need more difficulty than in the main chain to successfully attack the smart contract. That user would be able to attack all of the networks, and this scenario is considered infeasible.

## 4.4 Zokrates header chain verifier

Zokrates verifier receives 32 headers where the first and last header is public, and the rest are private. It also receives the block's hash that we want to append to this batch. Validates that each header's previous block hash points to the previous block and that the target difficulty included in the header is higher than its hash. The limitation of this framework and zk-SNARKs, in general, is the computational requirements for compilation and proof creation. The Zokrates program is easily extendable to bigger batches, which would significantly improve costs associated with smart contract storage. However, it will require high RAM and processing resources. Zokrates functionality has multiple phases, and each phase can be controlled by the Python script by its input parameters.

1. In the first place, the code is compiled into the arithmetic circuit. This process is quite resources hungry and limits the testing of larger batches.

2. The second phase is setup. During this step, Zokrates executes zk-SNARKs, which also generates toxic waste. If published, this toxic waste is usable for false proof creation, and therefore, we must ensure disposal of these data. We utilized single-party computation since this framework serves as proof of the work of created algorithm. However, in serious deployments, multi-party computations are necessary during the setup phase. Fortunately, by default, Zokrates supports this type of setup.

3. The third phase is verifier exportation. This phase generates a Solidity smart contract that is deployable to the Ethereum blockchain. Verifier complexity heavily depends on the number of parameters that the main function accepts. Private inputs do not increase their size but cannot be stored or even viewed inside the smart contract function. For this reason, we pass the first and last headers as public parameters, and the rest in between them are private. The smart contract only needs to store the checkpoints and does not need to verify or view what is between them.

4. The fourth phase is witness generation. During this phase, we finally inputted the formatted headers. The headers are split into 5 256 bit values because it is the largest single value type supported in Zokrates. These headers are in form to be ready for the double sha256 function. The function receives a transformed header with padding. The padding is a static value of 640 since headers also have a fixed length of 80 bytes. After the witness phase, a witness of execution is generated, but only if the execution finishes successfully. If some assert failed midway through, non-valid witness is generated.

5. The fifth and last phase is the proof generation. During this phase, Zokrates transforms the witness into zero-knowledge proof of computation. This proof is acceptable by the Verifier smart contract. This action is also rather resource-heavy. However, it

is a single-core processor and should be without significant problems parallelizable. After submitting the proof to the verifier smart contract, this verifier will return a Boolean value indicating its output.

# Chapter 5

# Implementation

The implementation is divided into four main components. We describe their functionality in the previous chapter. In the following sections, we will describe their internal design and usability.

In this implementation, we use providers for the gathering of data from blockchains, except for the direct mobile connection to Ethereum. We chose to use providers because of the high requirements of running full nodes locally for all the needed blockchains. All the endpoints and data sources used have existing alternatives in all nodes and are easily replaceable. This provider usage does not damage the trustlessness of this framework since the client connection to the primary blockchain is direct in the peer-to-peer (p2p) network.

## 5.1 Server

The server serves data handing, smart contract actions, and Zokrates-related tasks. We decided not to implement automated chain building since the proof creation is very resource heavy, and we would not be able to catch up to the current state for multiple blockchains in a reasonable time. However, we created an interface for creating smart contracts, dynamically creating proofs and submitting them to the smart contract, which we use for automating this task.

A simple CLI interface controls the server. It should not be used in production deployment but as a base or a template for automating available actions. It is an interface to perform the whole workflow of this framework. It can set up Zokrates and smart contracts. And then perform actions around batches such as their proof creation, submitting, and contract interactions.

The server provides five main functions:

- Compile - Compile the Zokrates verifier and update the existing smart contract
- Deploy - Take the latest version of the smart contract and deploy it to the configured blockchain. The deployment executes through transactions that need to be signed and paid.
- Proof - Creates a witness and proof for a given header range. The range splits into 32 header-sized parts, and, for each, we generate separate witness and proof. This action also takes as a parameter `blockchainId` to identify the source of the data for proof. From the source header, batches are gathered, parsed, and in the end, validated. We store the resulting proof in a file identified after input parameters.

- Interact - Takes the same parameters as the proof, selects created files based on those parameters, and creates a transaction containing all the proofs concatenated into a single array. Then we send this transaction to the currently deployed smart contract. Interaction is a complete transaction, requiring signing and enough balance in the singer's account. The contract will parse and validate the whole array and, if successful, will record the last element as a checkpoint.
- Call - This function is for verifying contract functionality. It calls for the contract method to return the closest hash to the given height.

### Data gathering and parsing

For data sources, we once again utilize `API` providers because of the high costs of local full nodes on required blockchains. However, all of the endpoints used have existing replacements in standard full-node implementations. The server validates all headers before creating proofs by recalculating their hashes. We represent the headers received from the APIs as a Python object serialized to forms required by different parts of the application. Our chosen secondary blockchains are Bitcoin and Bitcoin Cash. We chose those because of their high similarity. Therefore, the differences in server between those two are only in the extent of changing endpoints. We also transform the received data into binary representations of hex numbers during serialization, which is the expected format in the Bitcoin sh256 function. There are two possible serializations, the first is a basic header, where the raw data are concatenated and padded if needed. The second is Zokrates input. The server must split the Zokrates input into 256-bit parts due to the Zokrates type sizes. It takes the header created in the previous serialization and creates a space-separated list of header parts. Then it passes this list into the Zokrates toolbox during witness creation.

### Zokrates control

For controlling Zokrates, we use the `CLI` interface on the Zokrates toolbox that runs directly from Python. Frameworks such as `zokrates-js` exist to handle this task from within the code. However, we decided on more straightforward contract handling to fully implement the server part in Python. Each phase creates its output files and names them based on the chain and header range. The operations are all single-threaded and blocking. Before each phase, python checks if the zokrates toolbox is available in the system and, if not, installs it inside the project folder. Created proofs persist after submitting the smart contract to allow batching or re-sending them. During implementation, we grouped phases described in Section 4.4 into broader actions. The compilation also sets up the environment and updates the verifier contract. Witness and proof creation are grouped into single actions because there is not much point in doing one without the other.

### Smart contract control

Smart contract control divides into two parts. The first is its deployment. For this part, we create a raw transaction with contract `ABI` and, through a `web3` connection to Ethereum, we publish this transaction. The web3 once again connects to the provider of the Ethereum node. The transaction sender is defined by a private key in constants and must have sufficient funds for this action. We chose to deploy with built web3 transactions as opposed to frameworks like `Truffle` or `Hardhat` because this way provides much more flexibility in

terms of dynamic price and blockchain management. Contract transactions cannot migrate contracts to newer versions instead of smart contract management frameworks. However, our framework also contains a functional Truffle project that can act as an alternative. The server provides a simple `API` interface for the application to update smart contract information. However, the smart contract data can be easily hard-coded into the application, and they can become fully separate entities. The interface is a simple endpoint `REST` using the Python framework `Flask`.

## 5.2 Client

The client is a react native application compatible with Android devices. Its purpose is to showcase the usability of this framework on mobile devices. For the primary blockchain Ethereum, the application shows the user account stored inside `Geth keystore`. Geth instance generates the account and, for this blockchain, we only show the current state, since the `Geth` instance inherently validates transactions. So there is no point in showing them to the user. By default, the Geth instance connects as a light client to the Ropsten testnet. However, since in testnets, there is practically no motivation for full nodes to provide light clients connection and functionality, finding cooperating peers can take some time. This full node behavior is not a problem in the mainnet of Ethereum, but the initial setup required for contract interaction can take a relatively long time, even if the amount of data needed to download for header chain catch up is relatively small. Therefore, the application offers the option to connect to a provider while the Geth node is synchronized. This connection should only be used in development mode since it allows for breaking clients' trust.

### Ethereum connections

Ethereum connection is achieved by running Geth instance in Native module called `CommunicationNative`. The instance runs directly `Go-lang` in Java using the wrapper already created. There is no available documentation for this wrapper, and it is limited to basic functionality. Therefore, the available Ethereum actions are minimal within the application.

Native modules, by default, do not provide an interface for communication with react-native. A callback function is passed to each call from a native function to solve this. This callback can only be used once and will contain an error message or the function result. Ethereum instance is stored in the instance of class `NodeHolder` and is initialized at the start of the application within `MainActivity`. The class `NodeHolder` also serves as data storage for the module and contains and provides an interface to access the user's address and the location of the KeyStore. The module `CommunicationNative` interacts with Ethereum by utilizing functions in Geth CLI that has a wrapper to support Java. The most significant function in this module `getClosestHash` creates a call to the Ethereum network. Since calls do not require payment, the user is not limited or penalized for re-freshing data. The function returns the closest hash to height in the blockchain identified by its id. The client verifies the returned value as a block hash and its height. The client can assume that these data represent a verified header at that height and store them in the persistent local storage of verified headers.

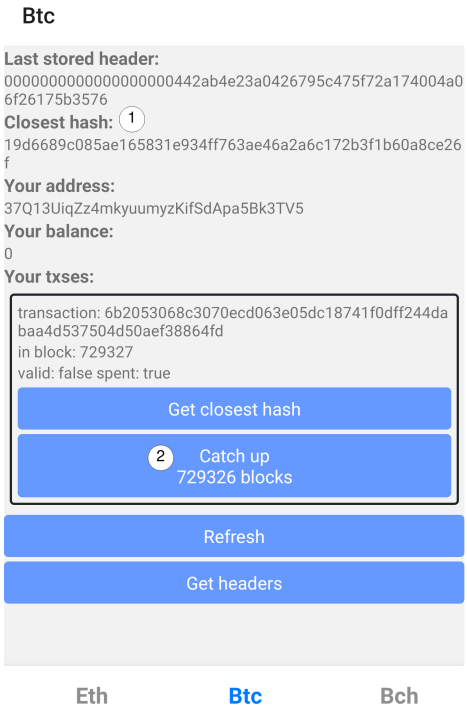


Figure 5.1: Empty smart contract.

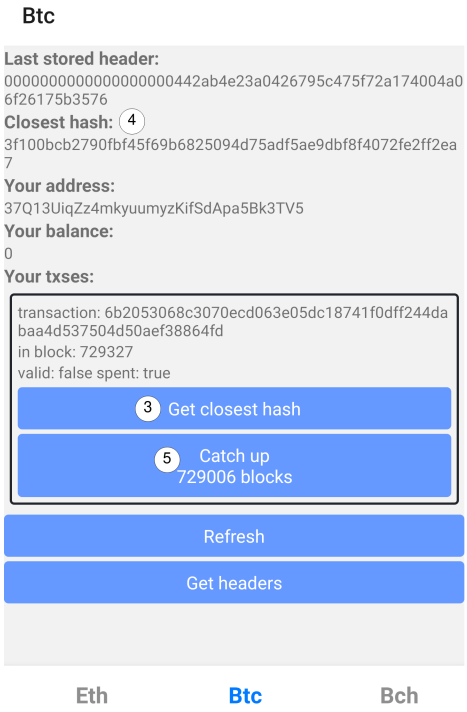


Figure 5.2: Smart contract with 10 batches.

### React native application

As the main application, we chose react native because we prefer Typescript over Java. The application uses Redux for all its interactions with outside data. It is used for both Native module calls and API interactions.

The application offers two main functions. Button labeled `Get closest hash` sends a request to the native module to query the smart contract for the closest hash to the given block. We display the closest hash for each transaction for a given account. After receiving this checkpoint, the application can catch up with the blockchain head starting there. In Figure 5.1 the smart contract only contains the Bitcoin genesis block, as can be seen in `Closest hash` marked by (1). After sending a transaction with ten batches of 32 headers, called `Get closest hash`, the user can execute the catch-up action marked by (2). This action queries the smart contract for the closest checkpoint to the given block number. The Figure 5.2 shows that `Closest hash` after the user presses the Get closest hash button marked with (3), the closest hash value marked with (4) has changed, and also that the catch-up length marked with (5) is smaller by 320 blocks, which is the number of blocks we submitted in ten batches.

### Data storage handling

For all asynchronous actions, the application utilizes `Redux` state management library. For permanent data such as contact information, validated header chain, and more, the application uses an extension of Redux called `redux-persist`. The extension uses permanent storage with crucial value inside the phone. In this way, validated transactions remain validated even after the application is shut down without external state management.

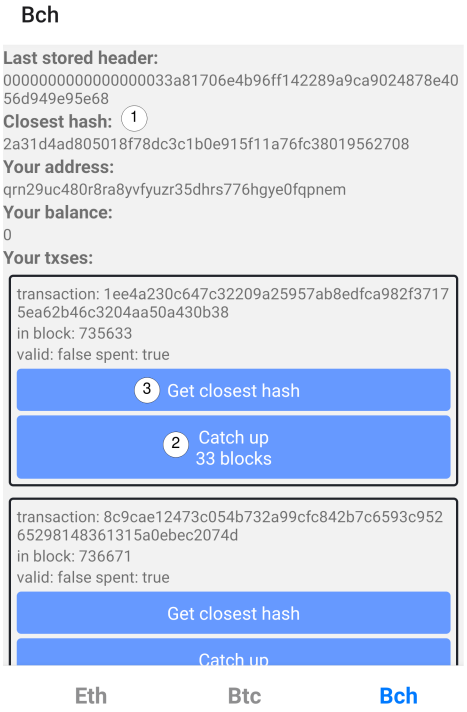


Figure 5.3: Empty smart contract.

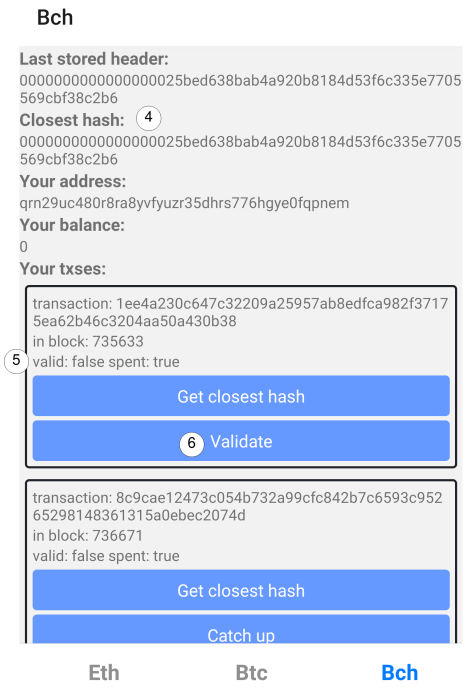


Figure 5.4: Smart contract with 10 batches.

The application stores validated headers in a similar data structure as a smart contract. They are mapping their block height to their hash. This way, finding closest valid hash is a matter of looping downward through the local header chain. We chose to use this structure due to its inherent sorted nature without any manipulation. We add a new header only if it originated from the blockchain or forms a header chain originating from the blockchain. When forming local header chains, we recalculate a hash of the headers and check the target and previous header link. Like servers' implementations, the headers are represented as objects that provide the functionality to their data.

### Blockchain synchronization

The client synchronizes with secondary blockchains by calling the smart contract with the required block height and checking the local header chain as to which header is closer. The client builds a local chain from the received checkpoint, after which we can query the block for the inclusion of transactions.

Transaction inclusion is determined by asking a full node for Merkle proof for the selected transaction and validated block. This proof is then locally validated, and if the resulting Merkle root matches the root stored inside a valid block, the client can be confident that the transaction is included there. The endpoint for receiving proof is compatible with the standard implementation of bitcoin-based nodes.

In the application, Figure 5.3 shows a transaction that is in a block of 33 headers above the currently loaded header. Note the hash value in the closest hash field marked with (1). When the user presses the catch-up button marked with (2), the application downloads 33 headers from the secondary blockchain Bitcoin cash. The headers start at the closest hash

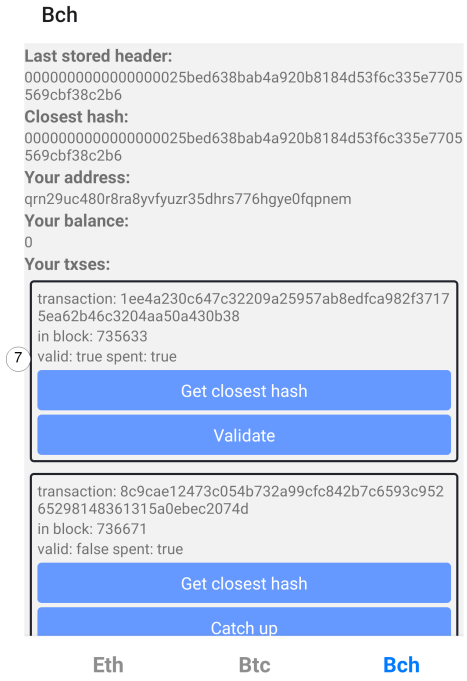


Figure 5.5: Smart contract with 10 batches.

value and continue until the header with the selected transaction. Afterward, the user can press the `Get closest hash` button marked by (3). The result of this action is visible in Figure 5.4. Note that the closest hash field, now marked with (4) has been updated. And furthermore note the current value of the field valid marked with (5). This transaction can now be validated. The button `Validate` marked with (6) queries a full node for Merkle proof of transaction inclusion and recalculated this proof if the resulting Merkle root is the same as the validated local block, the flag `valid` inside the local transaction object changes to true.

In Figure 5.5, we can see the result of the validation. The valid field marked with (7) in the transaction has turned out to be true. This field shows that the transaction is valid and is included on the blockchain.

## 5.3 Smart contract

The smart contract is primarily used for the storage and retrieval of validated headers. It builds its own headers chain composed of checkpoints and keeps track of accumulated difficulty to support forking. During its initialization, we set up multiple blockchains. The blockchain ID, its genesis block, and the height of the genesis block are stored in smart contract storage during the setup phase. In addition to adding genesis, blockchain initialization also adds an initial fork with a pointer to itself at the place of the previous fork. Since the default Id of this first fork is zero, the search will stop after reaching this pointer without trying to search further down. After this initialization, the contract can build chains of valid batches on top of this fork.

We do not need to verify hash calculation from submitted headers, since zokrates verifies that. We only need to calculate the difficulty target to store the cumulative difficulty in the local chain version. The difficulty is calculated from the maximum difficulty divided by the target. We can utilize bit-wise operations on 256-bit numbers in solidity, which Zokrates does not support. Therefore, we can use masks for a faster and much cheaper target calculation.

### Fork handling

Forks are handled by constantly storing the cumulative difficulty of the whole blockchain in each batch. Forks always start with the previous fork linked to its last height at the last fork and its current height. The main chain will most likely be composed of multiple forks with this structure. After each addition of forks, we automatically choose a new main fork depending on the total cumulative difficulty inside the highest batch at available forks. Since the previous main fork has already been decided to its state by this action, we only need to compare the difficulties of the new and main forks.

The cumulative difficulty of the batches automatically updates after each batch to the same current and previous batch difficulties. If the batch is first in a new fork, we acquire difficulty from the batch at the height of the previous fork stored in the current fork.

When determining the fork for the newly validated batch, we scan all forks in the current chain. When a fork contains a previous hash at a height lower than the verified hash, we can be confident that the fork can accept that hash. Unwanted collisions during this action are improbable since two different forks would need to have the same hash at a given height and different contents.

### Client interaction

Client interacts with a single function `ClosestHash`. This function returns the closest validated hash and its height at the selected blockchain and height. The function triggers a private recursive function `getClosest`.

```
function getClosest(
        uint chainId,
        uint height,
        uint forkNumber
    ) private returns (uint256[] memory) {

    Chain storage headerChain = chains[chainId];

    // using undefined array length for geth warpper compatibility
    uint256[] memory ReturnVal = new uint256[](2);
    Fork storage mainFork = headerChain.forks[forkNumber];
    if (height > mainFork.forkHeight) {
        height = mainFork.forkHeight + 1;
    }

    for (uint i = height; i >= 0; i--) {
        // if reached some hash return it
        if (mainFork.batches[i].lastHeaderHash != 0) {
            emit ClosestHash(mainFork.batches[i].lastHeaderHash);
            ReturnVal[0] = mainFork.batches[i].lastHeaderHash;
            ReturnVal[1] = mainFork.batches[i].height;
            return ReturnVal;
        } else if (i == mainFork.previousHeight) {
            // if reached previous fork continue searching in it
            return
                getClosest( chainId,
                    mainFork.previousHeight,
                    mainFork.previousFork
                );
        }
    }
    emit ClosestHash(0);
    return ReturnVal;
}
```

Listing 1: Chain traverse in smart contract

The function in Listing 1 traverses all batches in forks that precede the main fork. The traversal is downward because, for the client, building header chains from the bottom-up is easier and once smart contract reaches top of blockchain it is easier to enforce consistency downwards. The traversal ends when we reach height zero or when the current fork has a verified block hash at the current height. When the function reaches fork height zero and the fork has the previous fork defined, the function recursively runs in the previous fork at the starting height that is stored in the current fork. The function returns both hash and its height in the header-chain in a single array of unspecified length, because of expected type limitations of the go wrapper in clients native module.

## 5.4 Zokrates header chain verifier

Zokrates program verifies that a sequence of headers is valid and continues from some starting point. The program accepts an array of 32 headers and creates proof of its validation. We designed the Zokrates program for 32 headers, however, it is extendable and reducible for different sizes. Its primary function accepts a list of headers and their hashes. The first and the last values from both lists are public parameters, and the rest are private parameters. This choice of visibility is to optimize the smart contract validator because its size grows rapidly with additional public parameters. Secret parameters are not passed to the final verifier; only public ones are passed. Because the largest single value type in Zokrates has 256 bits, the input must be split into 256-bit hexadecimal values.

### Batch validation

The zokrates program for each header calculates hashes and targets and then checks its continuation of the previous header. The `sha256` function of the Zokrates standard library calculates the header hash. We need to pad this function since we are applying a hash function on headers with 80 bytes. We add a constant value at the end of the `sha256` input and the number of bits needed to reach the end of the input to create padding. The implementation of header hashing is shown in Listing 2.

```
def hash_block_header(u32[5][4] preimage) -> field:
    u32[8] preimage1 = [ ...preimage[0], ...preimage[1] ]
    u32[8] preimage2 = [ ...preimage[2], ...preimage[3] ]
    # hex representation of number at the end of input values
    u32[8] preimage3 = [ ...preimage[4], 0x80000000,
                         0x00000000, 0x00000000, 0x00000000 ]
    # last part of sha256 input with number of padded bits in hex
    u32[8] dummy = [ 0x00000000, 0x00000000,
                     0x00000000, 0x00000000,
                     0x00000000, 0x00000000,
                     0x00000000, 0x00000280 ]
    # first hash of the input
    u32[8] intermediary = sha256for1024(preimage1, preimage2,
                                        preimage3, dummy)
    # second hash with the results from first
    u32[8] res = sha256for256(intermediary)
    # changing endianness
    res = change_array_endainnes(res)
    # transforming into single 256 bit value
    return u32Pack256(res)
```

Listing 2: Header hashing in zokrates

From the limitations of Zokrates types and standard functions, we must transform headers to arrays of 32-bit numbers from input form input forms of 128-bit values. These arrays are passed to function in Listing 2 and then padded into a single 1024-bit value. This value is passed to the first sha256 function, and its result can now be passed without padding into the second sh256 function. These hash functions return an array of 32-bit numbers

that together form a single 256-bit hash. To allow final verification of hash output, we needed to swap the endianness of the resulting hash. To swap endianness, we need first to reverse the array and second reverse bits in each array element. The array reverses in a simple for loop, and elements need to first be turned into bitwise array representation and then spread into a new array in reverse order.

Zokrates guarantees the correctness by asserts, which stop witness creation execution quicker than keeping state, and the resulting witness is not valid. Bitcoin stores the targets in the headers in values called bits. It is a 32bit number, where the first six bits represent the amount left shifts of the rest of the bits to calculate the target of the hash of the header. In the following equation, we can see the entire equation of target calculation, where the head is the first six bits, and the tail is the last 26 bits.

$$Target = tail * 2^{(8*(head-3))}$$

Since Zokrates does not support dynamic shift sizes or exponent calculations, we calculate the amount of shifts from the head value and loop over the tail with 64 times shifting to the left by one. When we reach the number of shifts defined in the head, we cannot escape the loop, so we need to set the shift to zero.

# Chapter 6

# Framework evaluation

Testing performance is concentrated mainly around Zokrates and the smart contract part of this framework since that part provides functionality and is the most expensive to perform. As the client does not require performing any significant or costly tasks, we only tested data requirements for catching up based on different checkpoint margins.

## 6.1 Batch submission cost

The cost of submitting batches comes from three places. The first is the contract storage utilization, the second is the computational resources required for proof creations, and the final is the client network requirements on synchronization. Storage after account creation is the second most costly action within the Ethereum network [3]. Therefore, optimizing this part is crucial for the performance of this framework. We created proofs of 32 header batches for the first 1000 headers for testing. Since the smart contract can accept arrays of proofs, we tested the ideal array length per the price of its submission, the cost of initial catch up and the maximum client synchronization cost.

### Contract storage utilisation

During storage utilisation testing we used function provided by `web3` framework called `estimateGas`. This function receives a raw transaction, executes it locally, and outputs the gas required for its successful execution on the Ethereum network.

In Figure 6.1, we can see that the price of submitting multiple batches increases linearly with their amount. During testing, we found a hard limitation of the Ethereum network in the form of maximum gas per block, which is a constant [3]. During testing, we already breached this limit in 22 batches in a single transaction.

The Figure 6.2 shows the cost of catching up to the main net of Bitcoin in Ethereum based on the average, minimum and maximum gas prices in March 2022[1]. This chart is only for general orientation, and Figure 6.2 shows the exact price of batched batches submission in gas. Since the blockchain always grows, we chose to show the cost of upkeep of this network in current state of Ethereum development. It shows total price in Ether for the continuous publishing of new block proof batches. Since in Bitcoin every 10 minutes a new block is created, assuming 30 day month, 4320 new blocks will be created by its end. The results show that costs drop drastically in smaller batches but begin to stabilize in more

[1] https://etherscan.io/chart/gasprice

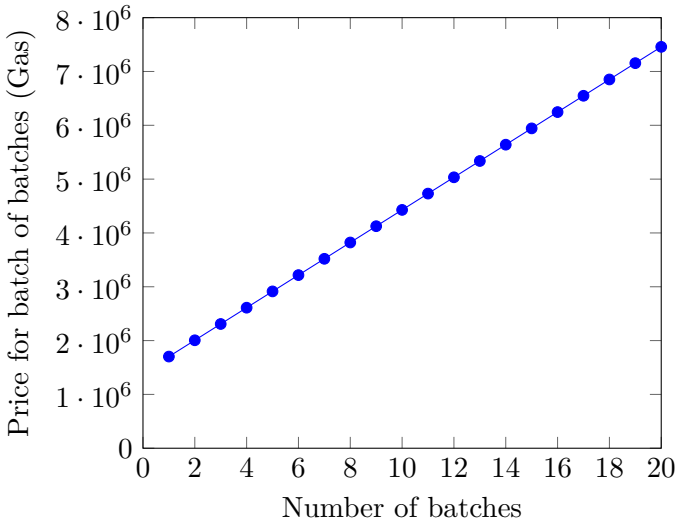


Figure 6.1: Price of submission for batch count.

significant amounts. The cost could be improved further if we switch to a less-constrained blockchain.

### Proof creation

Proof creation is a resource-intensive operation. To measure resource usage, we use utility `psrecord`. It consists of two consecutive actions. The witness computation, shown in Figure 6.4, requires less RAM and is a quick action. The proof computation shown in Figure 6.5 requires more RAM and takes significantly more time than the witness computation. These actions are single-core processes and can be parallelizable if there is enough memory to support them. During our testing, both were constrained by single-core CPU speeds.

We tested multiple configurations of the zokrates verifier. We needed to change the verifier to accommodate the amount of headers in batches. These changes are relatively simple and mainly comprise changing the main size of the loop and input array. An example of such changes can be seen in the file `btc16HeadersValidation.zok`, which accepts a batch of sixteen headers. The result of testing these configurations can be seen in Figure 6.6. The tests have shown that the RAM requirements increase linearly with batch sizes. Epochs in Bitcoin-based blockchains have Epochs of size 2016 [19]. Therefore, we elected the batch sizes to be powers of two. Each epoch has a constant difficulty target that changes between them. Furthermore, selecting batch sizes from powers of two allows us to check these targets outside of zokrates.

## 6.2 Storage optimization

When sending multiple batches in the transaction, the cost is smaller because the blockchain Smart contract only stores the last header of all the batches in permanent storage. However, these overlapping batches create larger spaces between checkpoints and force the client to download more headers for synchronization as a result.

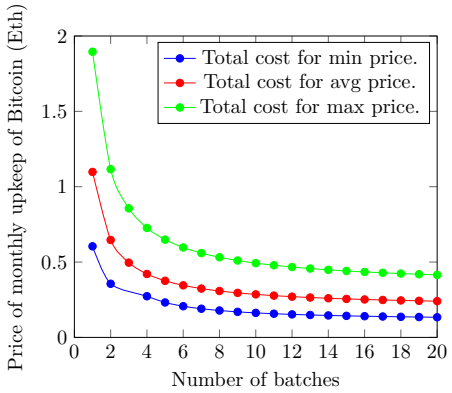


Figure 6.2: Price of monthly upkeep of Btc.

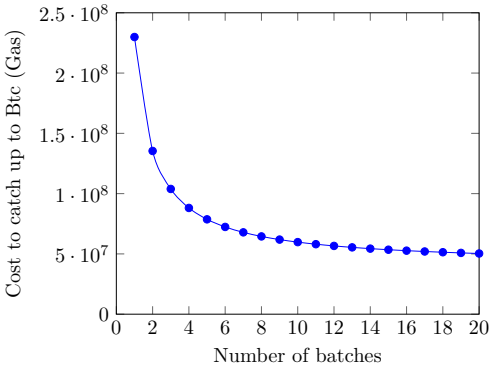


Figure 6.3: Price to catch up to Btc.

Figure 6.7 shows the growth rate for the number of blocks that synchronize according to batch size. Apart from initial synchronization, even batches of 20 or 30 headers are viable for devices with limited data.

### Total storage optimization

This framework works as a replacement for running separate Light clients inside mobile devices. In Bitcoin light client, one would need to download, store, and process around 60MB [19] for initial synchronization per client. With current wallets supporting several different blockchains, this amount would quickly grow to become unmaintainable. This framework provides a single blockchain light node as a source of truth, and the rest are dynamically synchronized. The total storage optimization will be different per user account, since the user only needs the parts of blockchains up to blocks containing their transactions. However, it grows proportionally to the number of supported blockchains.

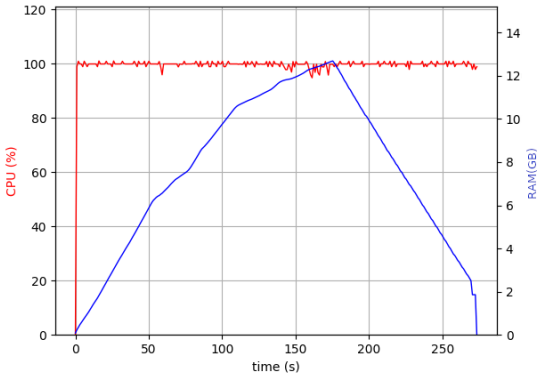


Figure 6.4: Witness computation.

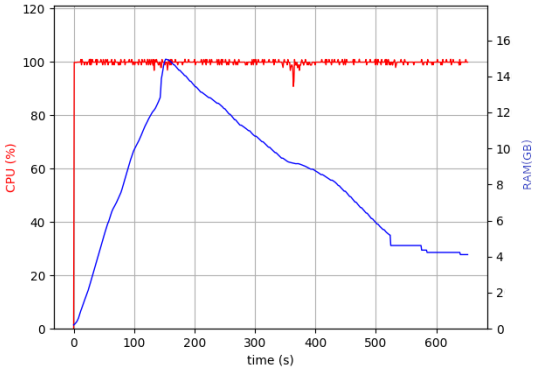


Figure 6.5: Proof computation.

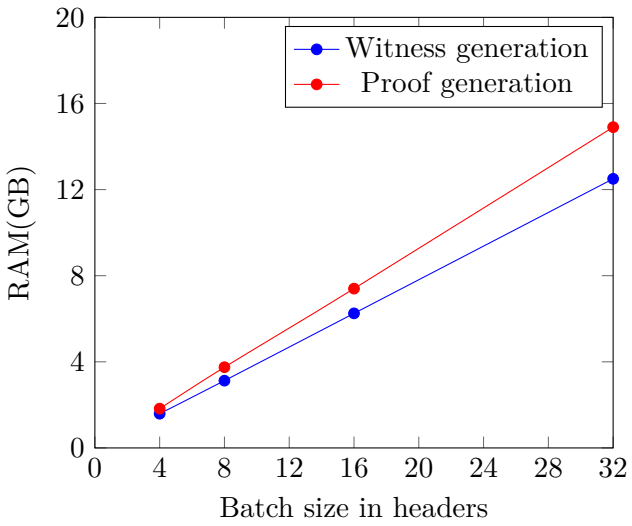


Figure 6.6: RAM requirements in proof and witness generation by batch size.

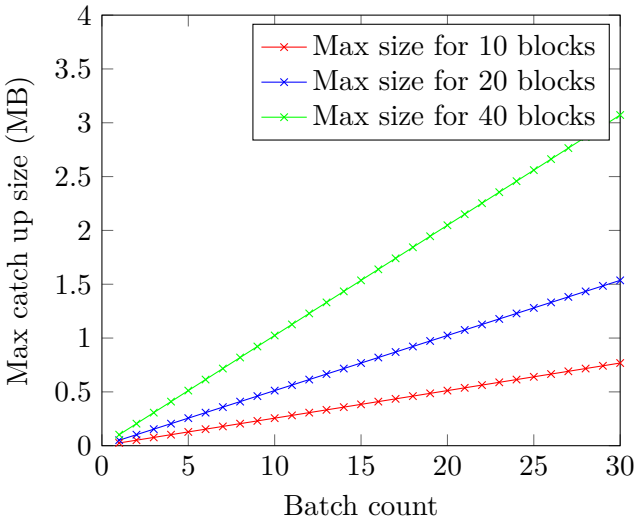


Figure 6.7: Maximum client synchronization size.

# Chapter 7

# Discussion

## 7.1 Application design

The application was designed as a proof of the work of this framework. The interface is not user-friendly for wallet functionality; however, it shows how this framework works with data from multiple sources. In a user facing implementation, the application would not display any of the manual functionality, it would automatically catch-up to blocks with user transactions and validate them without any user input or notice. The application would only have a list of transactions and the usual wallet capabilities.

## 7.2 Performance of proof generation

Proof generation is an action with high hardware demands and does take a significant amount of time. In blockchains, the speed of block generation varies. However, this framework must perform significantly faster proof creation than the interim-between-blocks to be viable in real-world use cases. In Chapter 6 we tested time and hardware difficulties, which have shown that the higher the header count and batch count, the cheaper its submission. The testing was constrained by hardware performance and the single-threaded nature of the Zokrates toolbox. For batches of 32 headers on `Intel i7-10510U`, we recorded an average witness creation time requirement of 255 seconds and a proof creation with an average time requirement of 681 seconds. Therefore, on our setup catching up to bitcoin would require about 936 seconds per block. The current speed of the Bitcoin blockchain is a new block every 600 seconds. This means that proof creation is about 20 times faster on our setup. However, this action is limited by our single-core speeds and nonparallelized computation of proofs. Therefore, we would be able, without further optimizations synchronize 20 secondary blockchains at the same time. However, the initial synchronization requires more powerful hardware. Because the current top block is at the height of 73320[1] and the proof generation of all proofs in Bitcoin would take about 27 days. Therefore, the framework is usable on consumer hardware if the blockchain has already caught up.

## 7.3 Framework limitations

The first limitation is the Zokratess toolbox. Zokrates, by default, supports a range of hash functions naively, and this range sets a hard limit on the choice of blockchains that can be

[1] https://www.blockchain.com/explorer

supported. Blockchains are further limited by the data size required for header validation. Zokrates has a limited number of public inputs. The limitation comes from generated smart contract size, which cannot be published into the main-net blockchain after some amount. This size can be reached because for each blockchain. We need to have at least two public header inputs. As mentioned in Chapter 5, the limitation of the maximum type size to 256 bits means that headers must be split into 256 bit parts. If the two required headers have large sizes, the contract deployment can be severely more expensive or impossible.

Another limitation is hardware and blockchain founds requirements for catching up to the current state of blockchain. The payments decrease directly with the number and size of batches. However, the larger the batches, the higher the requirements for proof calculation and the larger the minimum synchronization distance becomes for clients. It would be expensive to catch up to the top of the selected blockchains at the current configuration. However, even with bigger batch sizes, the Ethereum block size is still limited in its gas consumption, which we reached at 22 batches of 32 header-sized batches.

# Chapter 8

# Conclusion

We designed and implemented a framework for mobile wallets based on zk-SNARKs and smart contracts. The framework provides faster and less data-intensive synchronization than running multiple local light clients. The total storage optimization gain grows in proportion to the number of used blockchains since only a single light client is required to function correctly. The framework supports Ethereum as its primary blockchain, which serves as the source of truth for both client and server. Furthermore, the smart contract deployed on Ethereum serves as trusted storage and verifier of fragmented header chains.

The framework also supports two secondary blockchains, Bitcoin and Bitcoin Cash. The server creates local header chains of these secondary blockchains and generates proofs of their successful verification. The proofs are then submitted to the smart contract and verified in the on-chain computation. Except for initial deployment, this submission is the only action that requires on-chain payment. The client interactions are without any charges. The client is a mobile application that runs an instance of the Ethereum light node. This light node is used for accessing smart contracts with secondary chain checkpoints. After receiving a secondary chain checkpoint, the client can start building a local chain from it as a new temporary genesis block. Since mobile clients are only interested in transactions affecting them, they only require small parts of the whole blockchain to function. This framework allows them to securely verify blocks while preserving the security coming from light clients.

Apart from the initial costs of framework setup, our solution is a viable trust-less alternative to the current mobile blockchain wallets. Provides an interface to control Smart contract and Zokratess from a Python script. We designed it to be extendable for different batch sizes or proof of work-based blockchains with included and tested examples of such extensions. The implemented react native application showcasing the functionality of this framework demonstrates improvements in storage utilization as compared to other trust-less alternatives. The storage and bandwidth required for mobile synchronization for the secondary chains increases with the number of blocks we want to validate. However, in the worst-case scenario, if the client has a transaction in every single block in the blockchain, it will at most reach the requirements of light clients.

## Framework extensions

The framework can be extended to support other consensus mechanisms and other Blockchains. The extension will add a smart contract chain selection configuration, and in the

Zokrates program, we will need to add support for chain-specific batch handling. Another helpful extension is the support for parallelization and automating during proof creation and submission,which we did not implement, as it is applicable only during deployment to production. Furthermore, it needs to be explicitly configured for the different framework configurations.

# Bibliography


[1] Adam, L. *ZK-STARKs — Create Verifiable Trust, even against Quantum Computers*. 2018. Available at: https://medium.com/coinmonks/zk-starks-create-verifiable-trust-even-against-quantum-computers-dd9c6a2bb13d.

[2] Alin, T. *What is a Merkle Tree?* 2020. Available at: https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/.

[3] Antonopoulos, A. and D, G. *Mastering Ethereum: Building Smart Contracts and DApps*. 1st ed. O'Reilly Media, 2018. ISBN 9781491971895. Available at: https://books.google.cz/books?id=oJJ5DwAAQBAJ.

[4] Dashjr, L. *Bitcoin improvement proposal 62*. 2017. Available at: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki.

[5] Boneh, D., Drijvers, M. and Neven, G. Compact Multi-signatures for Smaller Blockchains. In: Peyrin, T. and Galbraith, S., ed. *Advances in Cryptology – ASIACRYPT 2018*. Cham: Springer International Publishing, 2018, p. 435–464. ISBN 978-3-030-03329-3.

[6] Buterin, V. *State Tree Pruning*. 2015. Available at: https://blog.ethereum.org/2015/06/26/state-tree-pruning/.

[7] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P. et al. Bulletproofs: Short Proofs for Confidential Transactions and More. In: Stanford University, University College London, Blockstream. *2018 IEEE Symposium on Security and Privacy (SP)*. 2018. DOI: 10.1109/SP.2018.00020. ISBN 978-1-5386-4353-2.

[8] Castro, M., Liskov, B. et al. Practical byzantine fault tolerance. In: Massachusetts Institute of Technology. *OSDI*. 1999, vol. 99, no. 1999. ISBN 10.1145/571637.

[9] Drijvers, M., Gorbunov, S., Neven, G. and Wee, H. Pixel: Multi-signatures for Consensus. In: Algorand and University of Waterloo. *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020, p. 2093–2110. ISBN 978-1-939133-17-5. Available at: https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers.

[10] Elena, N. *Demystifying Zero Knowledge Proofs*. 2018.

[11] Eli Ben, S., Iddo, B., Yinon, H. and Michael, R. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.* 1st ed. 2018, no. 1, p. 46. Available at: http://eprint.iacr.org/2018/046.

[12] Foundation, E. *Ethereum Whitepaper*. 2021. Available at: https://ethereum.org/en/whitepaper/#bitcoin-as-a-state-transition-system.

[13] Homoliak, I., Venugopalan, S., Reijsbergen, D., Hum, Q., Schumi, R. et al. The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses. *IEEE Communications Surveys Tutorials*. 1st ed. 2021, vol. 23, no. 1. DOI: 10.1109/COMST.2020.3033665.

[14] J., L. *Handbook of Theoretical Computer Science*. 1st ed. Elsevier, 1998. ISBN 978-0-262-72014-4.

[15] Maksym, P. *Why and How zk-SNARK Works*. 2019.

[16] Menon, S. J. *Implementing lattice-based cryptography in libsnark*. 2017. Available at: https://crypto.stanford.edu/cs359c/17sp/projects/SamirMenon.pdf.

[17] Merkle, R. C. Protocols for Public Key Cryptosystems. In: ELXSi International Sunnyvale, Ca. *1980 IEEE Symposium on Security and Privacy*. 1980. DOI: 10.1109/SP.1980.10006. ISBN 0-8186-0335-6.

[18] Mihir Bellare, A. S. and Vadhan, S. *Many-to-one Trapdoor Functions and their Relation to Public-key Cryptosystems* [Cryptology ePrint Archive, Report 1998/019]. 1998. Available at: https://ia.cr/1998/019.

[19] Nakamoto, S. *Bitcoin: A peer-to-peer electronic cash system*. 2009. Available at: http://www.bitcoin.org/bitcoin.pdf.

[20] Nuzzi, L. *Schnorr Signatures & The Inevitability of Privacy in Bitcoin*. 2019. Available at: https://medium.com/digitalassetresearch/schnorr-signatures-the-inevitability-of-privacy-in-bitcoin-b2f45a1f7287.

[21] Seurin, Y. On the Exact Security of Schnorr-Type Signatures in the Random Oracle Model. In: Pointcheval, D. and Johansson, T., ed. *Advances in Cryptology – EUROCRYPT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29011-4.

[22] Stallings, W. *Cryptography and Network Security: Principles and Practice*. 7th ed. Prentice Hall, 1999. ISBN 9780138690175. Available at: https://books.google.sk/books?id=Dam9zrViJjEC.

[23] Sullivan, N. *A (relatively easy to understand) primer on elliptic curve cryptography*. 2013. Available at: https://arstechnica.com/information-technology/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/.

[24] Tam, A. *Digital Signatures in Blockchains: The Present and Future*. 2021. Available at: https://bisontrails.co/digital-signatures/.

[25] Vesely, P., Gurkan, K., Straka, M., Gabizon, A., Jovanovic, P. et al. Plumo: An Ultralight Blockchain Client. *IACR Cryptol. ePrint Arch.* 1st ed. 2021, vol. 2021, no. 1, p. 1361.

[26] Wang, W., Hoang, D. T., Hu, P., Xiong, Z., Niyato, D. et al. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. *IEEE Access*. 1st ed. 2019, vol. 7, no. 1, p. 22328–22370. DOI: 10.1109/ACCESS.2019.2896108.

[27] Westerkamp, M. and Eberhardt, J. ZkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays. In: Technische Universitat Berlin. *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2020. DOI: 10.1109/EuroSPW51379.2020.00058. ISBN 978-1-7281-8597-2.

[28] Wood, G. *Ethereum: A secure decentralised generalised transaction ledger*. ISTANBUL VERSION 80085f7th ed. 2022. Available at: https://ethereum.github.io/yellowpaper/paper.pdf.

[29] Christian, R. *ZkSNARKs in a nutshell*. 2016. Available at: https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/.

# Appendix A

# Contents of the included medium

```
medium
├── zkWallet - Source code of implemented framework
├── thesis - Source code of this thesis
└── thesis.pdf - PDF of this thesis
```

Figure A.1: Contents of the included medium.

| Language | TypeScript | Java | Python | Solidity | Zokrates | JavaScript | Shell |
|---|---|---|---|---|---|---|---|
| Lines of code | 1327 | 458 | 445 | 351 | 218 | 201 | 166 |

Table A.1: Lines of not generated code per language without comments.