FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation and Improvement of Ethereum Light Clients

Empowering Ethereum on Resource-Constrained Devices

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Patrick Weißkirchner, BSc
Matrikelnummer 01328933

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Mag.rer.soc.oec Nicholas Stifter
Dipl.-Ing. Aljosha Judmayer

Wien, 11. Dezember 2019

Patrick Weißkirchner | Edgar Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation and Improvement of Ethereum Light Clients

Empowering Ethereum on Resource-Constrained Devices

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Patrick Weißkirchner, BSc
Registration Number 01328933

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Mag.rer.soc.oec Nicholas Stifter
Dipl.-Ing. Aljosha Judmayer

Vienna, 11th December, 2019

Patrick Weißkirchner | Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Patrick Weißkirchner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Dezember 2019

Patrick Weißkirchner

Danksagung

An dieser Stelle möchte ich mich sehr herzlich bei meiner Familie für ihre Unterstützung bedanken. Nur durch sie standen mir viele Türen im Leben offen und ich konnte meinen schulischen Weg so gestalten, wie ich es wirklich wollte. Bedanken möchte ich mich auch bei meiner Freundin, die mir immer zur Seite steht, besonders in schwierigen Zeiten.

Ich möchte mich auch bei meinen Betreuern Nicholas Stifter und Aljosha Judmayer bedanken, die mir die Möglichkeiten gaben, dieses Thema als Masterarbeit auszuwählen und für die sehr hilfreichen Feedbacks.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my family for all their support. It was only through their help that so many doors of opportunity were opened for me, and that I was able to pursue the education I truly desired. I also want to thank my girlfriend, who stands steadfastly by my side, even in challenging times.

I also want to thank my advisors, Nicholas Stifter and Aljosha Judmayer, for letting me choose this thesis topic and for their very helpful and insightful feedback.

Kurzfassung

Seit der Einführung der Peer-to-Peer Währung Bitcoin sind viele ähnliche Projekte vorgestellt worden. Ein beliebtes Projekt heißt Ethereum, welches erlaubt, Smart Contracts in seinem Netzwerk einzusetzen. Diese Contracts können von NutzerInnen entwickelt werden, um die Fähigkeit von Ethereum zu erweitern. Um mit diesem System interagieren zu können, wird eine Client-Software benötigt, die Blockchain-Daten herunterlädt und anschließend validiert. Als Blockchain wird die Datenstruktur bezeichnet, welche alle getätigten Transaktionen im Netzwerk speichert. Da eine große Menge an Daten kontinuierlich generiert werden, ist Ethereum auf schwächeren Computern nicht mehr einsetzbar. Aus diesem Grund vertraut man sich einer Drittpartei an, dass die heruntergeladenen Daten valide sind, um den zeitaufwendigen Validierungsschritt zu umgehen. Eine Alternative, die nicht die Validierung aller Daten benötigt, wird Simplified Payment Verification (SPV) genannt, welche nur ein Teil der Blockchain verarbeiten muss. Software dieser Art nennt man auch *Light Clients*. Allerdings ist auch dieses Verfahren zu rechenintensiv für Ethereum. Erst vor Kurzem wurde ein kryptografisches Verfahren namens *FlyClient* vorgestellt, welches eine schnellere Validierung verspricht. Jedoch existiert bislang noch keine praktische Implementierung. Es stellt sich also die Frage, wie man die Validierung der Ethereum Blockchain auf schwächeren Computern wieder ermöglichen kann. Eine Motivation liegt in der praktischen Anwendung, wie beispielsweise Zahlungen per Smartphone tätigen zu können. Bei sicherheitsrelevanten Anwendungen ist es von Vorteil, nicht von einer Drittpartei abhängig zu sein. Das Ziel dieser Arbeit ist daher die systematische Untersuchung von existierenden Verfahren, um Light Clients zu entwickeln. Der Fokus liegt besonders auf einer Schonung von Systemressourcen und die Vermeidung einer Drittpartei. Es werden existierende Ethereum Anwendungen und deren inbegriffenen Sicherheitsannahmen untersucht. Ein Ethereum Light Client Prototyp wird entwickelt, welcher den FlyClient-Ansatz verwendet. Es wird gezeigt, dass mit einer einfachen Modifikation der Ethereum Blockchain Light Clients entwickelt werden können, die: (1) *Payment Channels* unterstützen, (2) eine effiziente Verifikation der Blockchain ermöglichen, (3) in einer dezentralen Art und Weise arbeiten, (4) hohe Sicherheitsgarantien bieten, und (5) auf schwächeren Computern, wie Smartphones oder IoT-Geräten, eingesetzt werden können.

Abstract

Since the introduction of the peer-to-peer currency Bitcoin in 2008, a great number of similar projects have emerged. Ethereum is a popular cryptocurrency that extends Bitcoin's functionality by allowing users to define program code, so called smart contracts, that can be deployed and executed on top of its network. These contracts enable users to extend the Ethereum platform with applications beyond the relatively basic transaction types that are supported by Bitcoin. In order to interact with such networks a client software can be used, which downloads and verifies the required blockchain data. Blockchain is the data structure, which records all transactions of the system. Because of the continuous generation and validation of huge amounts of blockchain data, Ethereum is not deployable on resource-constrained devices anymore. For this reason, third-party services have to be trusted in the meantime to guarantee the correctness of the gathered data. An alternative to the verification of all data is the so-called Simplified Payment Verification (SPV), where only a small part of the blockchain has to be validated. A software using this approach is also called a *light client*. However, this approach is still too computationally expensive for Ethereum, resulting in the burden of many hours of work for resource-constrained devices. Recently, a cryptographic technique called *FlyClient* was introduced in order to speed up the validation process, but a practical implementation is not yet available. The question arises, how a client software can be constructed, so that Ethereum can be usable again on such devices. A major motivation for placing the research focus of this thesis on Ethereum light clients lies in the practical usage, in particular for payment applications on smartphones. For such security-sensitive applications it is favorable to avoid the necessity of third-party trust. The aim of this work is to systematically explore existing light client approaches with a focus on low resource consumption and avoidance of needing to trust a third party. This thesis enumerates various existing Ethereum applications and proposals and their implied security assumptions. A prototypical implementation of an Ethereum light client is developed, which makes use of the FlyClient's approach. It is shown that with a simple modification to the Ethereum blockchain light clients can be constructed, which: (1) support payment channels, (2) allow an efficient verification of the blockchain, (3) work in a decentralized way, (4) provide high security guarantees, and (5) are deployable on smartphones or IoT devices.

Contents

CHAPTER 1

Introduction

In 2008 Bitcoin was presented as a new peer-to-peer version of electronic cash by Satoshi Nakamoto [1]. Previously, a trusted third party, e.g. financial institutions, was required to process electronic payments. These transactions are reversible, because the institutions themselves handle disputes. In order to cover the expenses resulting from disputes, transactions are not free of charge. Nakamoto argues that small transactions are not feasible in this system, because of the high transaction costs. These costs can be avoided by using physical currencies, which cannot be used over a communication channel, for example over the Internet. Bitcoin attempts to solve these issues by avoiding the trusted third party and by avoiding the possibility of reversing transactions.

An electronic coin can be defined as a chain of digital signatures [1]. If a coin should be transferred to a new owner, the hash of the previous transaction has to be digitally signed by the current owner, who also has to include the public key of the new owner. This procedure results in a chain of ownership, where the last owner is the current owner of the coin. A resulting problem of this concept is the so-called *double spend*, which means, that the new owner of the coin cannot be assured, that the previous owner has signed the coin only to him. In order to solve this issue, all transactions have to be publicly agreed and only the first transaction of this coin counts as valid. One problem remains: Who decides, which transaction of the same coin occurred first? With a technique called *proof-of-work* (PoW), computers can generate blocks of new transactions in a complete peer-to-peer setting without a trusted third party. The goal is to add new blocks to the longest blockchain. A blockchain is the concatenation of blocks. The incentive for participants to add blocks to the longest blockchain is given in the form of new coins. These participants are also called *miners*. In this system double-spends are not feasible anymore, because computers have to follow a specific protocol, which does not allow a coin to be spent more than once. In case an entity adds a new invalid transaction to the blockchain, the other entities would notice that and would ignore the invalid block. Only one of the double-spent transactions can be in the blockchain at all. Merchants,

who accept Bitcoin, are able to look up transactions in the blockchain without needing a third party institution. If the correct transaction has arrived, they can now hand over the purchased products. Nevertheless, the merchants should wait a specific time to be assured, that the probability of a possible transaction reversal by adversaries is very low. Sompolinsky et al. [2] define *confirmations* as the number of blocks, which are already built on top of the block containing the specific transaction. They argue that, with every new block, the probability to reverse a transaction decreases exponentially. Garay et al. [3] introduced the term *common prefix*. This term refers to the blockchain, which is obtained by deleting a specific amount of the last blocks. The resulting blockchain should be equal for different honest miners. The goal of analyzing the Bitcoin network in a formal way is to describe attacks of possible adversaries in a rigorous way and to suggest a reasonable confirmation time for merchants in order to accept transactions safely, id est with a negligible amount of probability for an adversary to succeed in reversing them.

After the invention of Bitcoin, a new peer-to-peer blockchain project called Ethereum was presented [4]. The goal was to build a generalized technology, which means, that the intended purpose is not only to transfer money, like Bitcoin, but to be used in other types of applications as well. Therefore, developers can extend the platform with the use of smart contracts [5]. These contracts are applications, which can be written in a supported programming language. Afterwards, they can be compiled and uploaded to the Ethereum network. The goal is to enable users to execute arbitrary code in a decentralized way, such that the correct outcome of a function invocation is enforced by the miners. Chapter 3 describes Ethereum and its contract mechanism in more detail. Since not every computer is powerful enough to validate all conducted transactions and does not want to mine new coins, a verification technique called SPV (simplified payment verification) can be used instead. This technique allows the possibility to check the inclusion of a specific transaction in the blockchain without iterating over all transactions in the chain. Software, which only uses SPV verification, is called a *light client*. The focus of this thesis is the exploration of this type of clients in Ethereum. Therefore, the next section describes the goals, which these clients will have to reach.

1.1 Light Clients

Ethereum is more resource-intensive than Bitcoin (see Chapter 5 for a comparison). This fact has an influence on the construction of light client applications, which should be deployed on resource-constrained computing devices, such as smartphones or IoT devices. The ever-growing Ethereum blockchain size and the corresponding long validation time prevent the usage of SPV validation techniques on such devices. For this reason, some applications make use of external computational resources, where some amount of third-party trust is required.

Light clients are important in order to enable users to pay for goods and services in a fast and decentralized way. Especially for Ethereum, these clients should also be able to interact with smart contracts. The following list describes favorable properties of a

potential light client:

- The application should not rely on third-party trust. This means, that the application can verify the state of transactions without trusting another entity to get correct data.
- The user should not have to wait a long time in order to be able to conduct a transaction. Also, the user should not to have to wait a long time in order to get the result of the transaction.
- The application should run on an ordinary smartphone, which means, that the processing power of CPU/RAM and the bandwidth connection is limited.
- The application should be reasonably safe to use. In the context of blockchains, this means, that the light client should be as secure as full nodes. If it is less secure, then only in an acceptable small amount of degradation. An example for circumstances which influence the security of blockchains is the presence of adversaries.

1.2 FlyClient

In order to tackle the problem of the enormous SPV validation time for Ethereum mentioned in the previous section, FlyClient was proposed [6]. The authors of this light client approach promise proof sizes for Ethereum with less than 500 kilobytes, which are approximately 6600 times less than the average SPV proof size. SPV demands downloading data of a size linear to the blockchain length, whereas the FlyClient only needs a size logarithmic to the chain length. It is a probabilistic block sampling algorithm, which uses a Merkle Mountain Range tree (MMR) for its commitments. An explanation of the MMR data structure is given in Section 2.3, whereas the FlyClient is described in more detail in Chapter 6.

1.3 Scalability Issues

A huge problem of blockchain-based applications is the scaling issue [7]. The increased adoption of cryptocurrencies leads to more and more conducted transactions, but in Bitcoin the maximum transaction throughput is about 7 transactions per second due to the block size limit. Since there were conflicts in the Bitcoin community about finding different solutions to tackle this issue, Bitcoin was split into the original Bitcoin and Bitcoin Cash [8]. In the Ethereum network such an artificial limit does not exist, but scalability issues are also present for similar reasons: every transaction has to be processed by every full node in the network and the nodes also have to store the complete state, which is growing continuously [5]. One solution to this problem is the Bitcoin Lightning network proposed by Poon et al [9]. This technique is also called *payment channel*. The idea is, that not all transactions are recorded on the blockchain, rather channels are

opened and closed on the blockchain and all intermediate transactions are done off-chain. Off-chain transactions are messages, which participants can send directly to each other without the need to get processed by the Ethereum network. Only in case of a dispute, where participants do not agree with each other, the underlying blockchain will eventually resolve the issue. The Ethereum pendant is called Raiden [10]. Section 8.2 explains this approach more precisely.

Such payment channel techniques usually require entities to monitor the blockchain continuously in order to detect fraud. Since watching every new transaction arriving in the blockchain network is not feasible for resource-constrained-devices, a light client would have to find another solution. An interesting question is the following: how can we design a light client, which (1) is able to efficiently detect fraud, (2) does not have to be online the whole time and (3) can also work in a completely decentralized way? Section 11.2 attempts to answer this question.

1.4 Contribution

The goal of this thesis is twofold. The first is to systematically evaluate existing types of Ethereum applications, such as wallets, state and payment channels, and other popular examples, in the context of decentralization and to explain the security assumptions they are subject to. Because of the size of the blockchain and the long validation time, different trade-offs have to be made to enable such applications on resource-constrained devices. Furthermore, different existing approaches to build light clients are evaluated. The second goal is to explain the FlyClient's approach introduced by Bünz et al. [6] and to develop a proof-of-concept implementation using this approach with the assumption that the Ethereum blockchain does indeed meet the requirements of the FlyClient (see Chapter 6 for more details). This thesis attempts to answer the question whether light clients can be utilized without the necessity to rely on third-party trust. Additionally, procedures are outlined, how applications can be built on top of the introduced client. Different attacks are explained afterwards in order to evaluate the security of the client.

Therefore, the structure of this thesis is as follows: Chapter 2 introduces cryptographic primitives, which are important to be understood for subsequent chapters. Chapter 3 deals with the technicality of the Ethereum network and explains important concepts, which serves as the foundation for the rest of this work. Important aspects of state and payment channels are summarized in Chapter 4, which are later discussed in the context of a light client. Chapter 5 lists related work regarding light client concepts within different blockchain projects and the usage of Ethereum on resource-constrained devices. Chapter 6 deals with the FlyClient, which is used as the base for the proof-of-concept implementation. Chapter 7 defines the problem and its corresponding threat model, which the following chapters make use of. Chapter 8 enumerates existing applications of various types, which are used on the Ethereum network in context of light clients. Chapter 9 features a proof-of-concept implementation of a light client, which could be used in the Ethereum network, if the requirements to deploy the FlyClient were

satisfied. The subsequent Chapter 10 shows the resource consumption of the introduced approach measured on a Raspberry Pi 3B+ model [1]. Chapter 11 re-evaluates existing Ethereum applications within the context of the proof-of-concept implementation and outlines procedures, which enable them to operate without requiring a third party service. Chapter 12 describes possible attacks on the proof-of-concept dependent on different threat scenarios. Comments about the introduced application and possible improvements can be found in Chapter 13, and the summary of the paper is located in Chapter 14. Terminology used in this chapter and throughout the paper can be found in A.

[1] *https://www.raspberrypi.org/products/, Accessed: 2019-05-24*

CHAPTER 2

Technical Background: Cryptographic Primitives

This chapter outlines cryptographic primitives, which are used in Ethereum and throughout the paper. At first the cryptographic hash function is described, which enables the construction of a blockchain in the first place. Naranyanan et al. [11] define a hash pointer as a pointer to some information together with a cryptographic hash of that information. Therefore, hash functions are required in order to construct such pointers. Blockchain projects make extensive use of these hash pointers to interlink blocks to a chain. This enables the construction of a linear data structure, where transactions can be recorded and participants can agree on an ordering of these transactions.

Next, the Merkle tree is described. It is an important data structure for blockchain projects, which also relies on hash pointers. This tree is responsible to interlink all transactions in a specific block in a way, such that efficient proofs of transaction inclusions can be created.

The most relevant data structure for this paper, the Merkle Mountain Range tree (MMR), is introduced afterwards. It is similar to the Merkle tree. This construct is required to build proofs, which can be checked by light clients in order to verify the blockchain efficiently. The FlyClient makes extensive use of MMRs.

The section afterwards deals with digital signatures and their usage in Ethereum. Signatures are crucial in order to create accounts, such that only the owner of an account has the ability to transfer money to other addresses.

The last section defines the concept of proof-of-work, which most blockchain projects make extensive use of in order to agree on the current state of the blockchain in a decentralized way.

2.1 Cryptographic Hash Functions

This section summarizes important concepts from Chapter 5 of *Introduction to Modern Cryptography* [12]. A hash function H is a function, which maps long input strings to short output strings. The output is often called *digest*. An example, where such functions are often used, is a table data structure, which contains items. Item x is stored in a row with key $H(x)$. In order to retrieve the element, $H(x)$ has to be calculated and then the resulting key can be looked up in the corresponding row. A *collision* happens, if two different items x and y map to the same digest: $H(x) = H(y)$. In this case the items are stored in the same row one after the other. Therefore, a good hash function should minimize the number of collisions in order distribute items evenly on all possible rows. Otherwise, the usage of only a few rows can result in the deterioration of the lookup algorithm from a near-constant to a linear running time in the number of items.

The primary requirement of a *cryptographic* hash function is the prevention of two inputs having the same output digest. Because of the cryptographic context, we have to deal with an adversary, who has the goal to create collisions. This has to be avoided within the defined threat model. Theoretically, it is impossible to develop a hash function with fixed-size output, which never generates a collision if the input space is larger than the output space. Let $\{0,1\}^l$ be the output range of size l of a hash function H, then we only have to evaluate H $(2^l + 1)$ times with ever-changing input to find a collision with a probability of 1. Therefore, we cannot theoretically avoid collisions, but we have to make it hard for an adversary to create one. The following definitions are required to define the *collision resistance* trait for hash functions:

Definition 1. *We say that an algorithm A with input x runs in polynomial time if there exists a polynomial p, where the computation of A terminates within $p(\|x\|)$ steps for every input $x \in \{0,1\}^*$.*

Definition 2. *We say that an algorithm A with input x runs in probabilistic polynomial-time, if the algorithm fulfills 1 and additionally A has access to a random tape of length $p(\|x\|)$.*

With other words, a probabilistic polynomial-time algorithm has a source of randomness, which can be used for the computation.

Definition 3. *A function $f(n)$ is called negligible, if for every positive polynomial p there exists an N, such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

Definition 4. *A hash function consists of a pair **(Gen, H)**, both are probability polynomial-time algorithms, which satisfies the following requirements: (1) **Gen** takes as input a security parameter 1^n and outputs a key s. It is assumed that 1^n is implicit in s. (2) **H** takes s and a string $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{l(n)}$.*

Definition 5. *A hash function **Π=(Gen, H)** is **collision-resistant** if a negligible function $negl(n)$ can be defined for a probabilistic polynomial-time adversary $\mathcal{A}$, such*

that $\mathcal{A}$ can only succeed in generating a collision with probability $\leq negl(n)$, given input s and H.

We have defined a hash function with the usage of a generator *Gen*, which generates a key. In practice, a hash function H is often simply defined as: $\{0,1\}^* \rightarrow \{0,1\}^l$, where l is the length of the output. Theoretically, this definition cannot be used, because there always exists a constant-time algorithm, that outputs a hash-collision for a pair of inputs: the collision only has to be hard-coded in the algorithm. Using the security parameter 1^n with a size of length n, it is impossible to hard-code a colliding pair for every possible input length. For the next (informal) definitions, we omit the generator.

Definition 6. *A hash function **H** is **second-preimage resistant**, if it is infeasible for a probabilistic polynomial-time adversary $\mathcal{A}$ to find an x', such that for a given x it evaluates to $H(x') = H(x)$.*

Hash functions, which are collision-resistant, are also second-preimage resistant.

Definition 7. *A hash function **H** is **preimage resistant**, if it is infeasible for a probabilistic polynomial-time adversary $\mathcal{A}$ to find an x, such that for a given y, which was uniformly picked, it evaluates to $H(x) = y$.*

In this thesis we assume, that collisions cannot be found in a reasonable time and the deployed hash functions are also preimage-resistant. Bitcoin uses the hash function SHA-256 [1], whereas in Ethereum Keccak-256 [4] is used. Both have a fixed-size output of 256 bits. The proof-of-concept implementation makes use of Keccak-256, but it also uses SHA3-256, where only a different padding is applied compared to Keccak-256 [13] [14]. Another hash function is RIPEMD-160 with a digest size of 160 bits. It is used in Bitcoin [15] and it exists as a *precompiled* smart contract in Ethereum [4]. The functionality of precompiled contracts is already built into the Ethereum node. On the other hand, the code of ordinary smart contracts has to be stored in the blockchain.

2.2 Merkle Tree

An important data structure in blockchain projects is the Merkle tree [16], which can be built on top of cryptographic hash functions [11]. Figure 2.1 shows an example tree with eight inserted elements. Every non-leaf node in the tree is the result of a hash calculation of its concatenated children. Algorithm 2.1 lists a procedure to calculate the root hash of such a tree. It needs access to a cryptographic hash function H (like one mentioned in Section 2.1). Data to insert has to be given as input. Note that the number of items is assumed to be even, otherwise a padding has to be applied to Figure 2.1 as well as to algorithm 2.1. The '||' operator means the concatenation of the elements on the left and right side of the symbol.

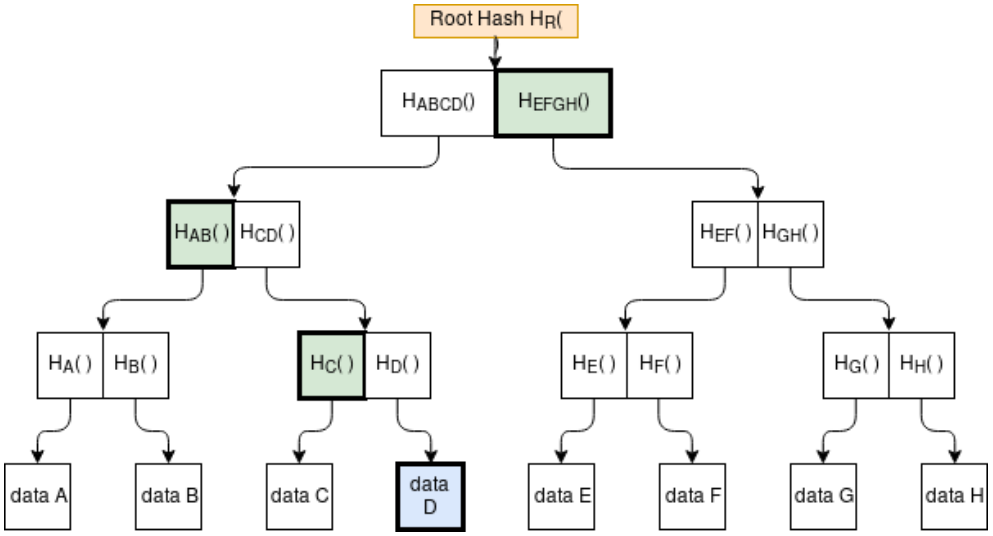

Figure 2.1: Merkle tree with eight elements

Algorithm 2.1: Calculate root hash of a Merkle tree

Input: cryptographic hash function H, hashed data set S to insert

Output: root hash R

```
1 L = S;
2 while length(L) > 1 do
3     M ← ∅;
4     for k = 0, k = k + 2, k < length(L) do
5         hash = H(L[k] || L[k+1]);
6         M.push(hash);
7     L = M;
8 R = L[0];
9 return R
```

An advantage of this data structure is the property called *proof of membership*. In order to prove the inclusion of *data D* in the Merkle tree of Figure 2.1 with a given root hash R, it is enough to have access to the hash calculation outputs of H_C, H_{AB}, H_{EFGH}. A verifier can start with the concatenation of the hash output of *data D* and H_C and hash the result. After that, the new hash is concatenated with the given output of H_{AB} and then hashed again, and so on. If the calculated root hash is the same as the root hash to counter-check, then the verifier can be assured, that *data D* is indeed in the Merkle tree. This fact follows from the collision-resistance property of cryptographic

hash functions, because it is not possible to calculate the same root hash with different elements. Inclusion proofs are logarithmic in the size of the inserted data items due to the tree structure.

Bitcoin uses Merkle trees to be able to prove transaction inclusion efficiently [1]. Ethereum uses a similar construct called *Modified Merkle Patricia tree* [4], which also enables proof sizes logarithmic to the inserted elements. An advantage of this data structure is the potential to insert and delete elements efficiently in an existing tree [5].

2.3 Merkle Mountain Range Tree

FlyClient makes extensive use of the Merkle Mountain Range tree (MMR) data structure [17]. Therefore, Bünz et al. [6] formalized the MMR and proved important properties. It is similar to the previously introduced Merkle tree, because non-leaf nodes also store the hash of the concatenation of its children. The data structure is used as an append-only log, where a new leaf can only be inserted at the end of the existing leafs. Figure 2.2 shows the resulting MMRs for inserting the first four elements. L1 denotes the first element of the tree, L2 the second and so on. Intermediate nodes and the root node are labeled as N. If the leaf number n of the current MMR is a power of 2 ($\exists k \in \mathbb{N}_0(n = 2^k)$) then a new root node has to be created to get the next valid MMR, which connects the old one with the new element. Otherwise, the new leaf node gets inserted in the right branch of the current MMR in the rightmost position in such a way, that the number of leaves in the left subtree of every non-leaf node is (1) always a power of 2 and (2) higher or equal to the number of leaves in the right subtree. Only a logarithmic number of changes has to be made to the current MMR in order to insert a leaf node and the positions of the inserted leaves are not allowed to be changed. This helps software implementations to easily extend this data structure without the need to rebuild the complete tree. The advantage of this procedure is a succinct constant-size *commitment* in form of the root hash of the MMR to all inserted elements and their positions. Like in the case of a Merkle tree, an MMR proof can be given in order to prove the inclusion of an element and its position in the existing tree if the root hash is already known.

2.4 Digital Signatures

This section summarizes important concepts of a digital signature scheme described in *The Elliptic Curve Digital Signature Algorithm (ECDSA)* by Johnson et al. [18]. A digital signature scheme can be defined as the digital counterpart to handwritten signatures. The signature depends on a secret, which is only known to the signer. An unbiased third party should have the ability to verify a signature without the knowledge of the signer's secret. A key pair consisting of a private key and a related public key makes this scheme possible. The signer keeps the private key in secret and uses it to sign data. He has to make the public key available for other entities in order to allow them to verify the signature of the signed data. An important goal of such schemes is security. It should

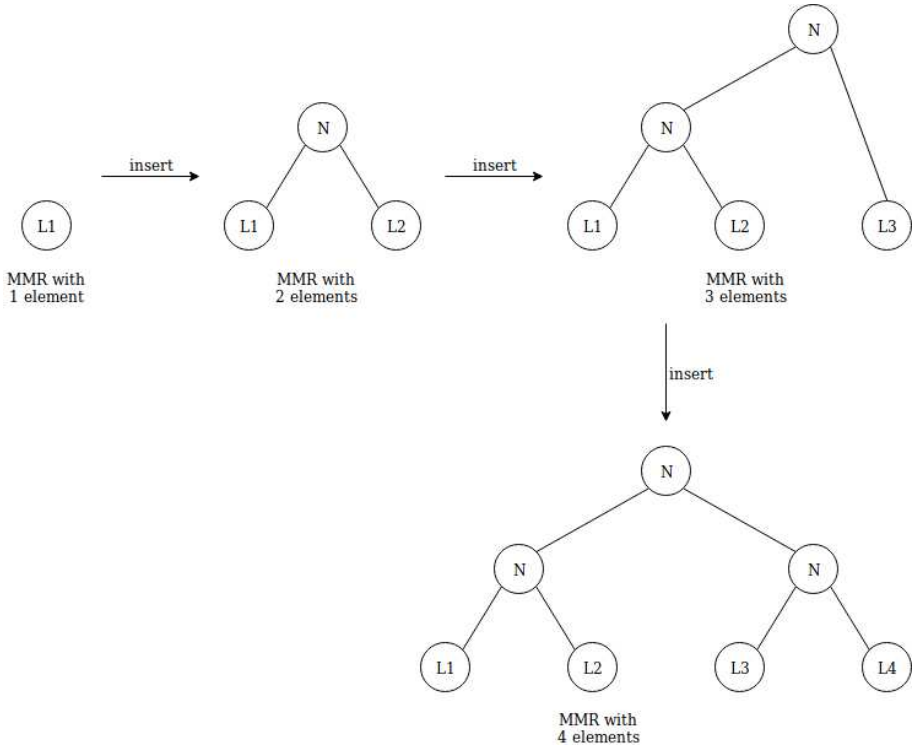

Figure 2.2: Insertion of elements into an MMR

not be feasible for adversaries to generate valid signatures without the knowledge of the corresponding private key.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature scheme, which makes use of a mathematical construct called *elliptic curves*. The security is derived from the computational intractability of calculating the discrete logarithm on the curves. Let Alice be the entity, who wants to generate a key pair and let Bob be the entity, who wants to verify a signature generated by Alice. The required steps are listed below:

1. At first Alice has to choose some parameters for ECDSA. Examples are the curve parameters and the choice of the field, where the curve is defined on. Let Alice pick *secp256k1* [19], where the corresponding parameters are contained in the set denoted by D. Alice also has to choose a cryptographic hash function H, which is preimage- and collision-resistant (see Section 2.1).

2. Alice has to generate a random number d between the interval $[1, n-1]$, where $n \in D$. The private key is the random number d. The public key Q can be calculated with $Q = dG$, where $G \in D$.

3. In order to sign a message m, she calculates the digest $H(m)$ and transforms the output to an integer e afterwards. She picks a new random number $k \in [1, n-1]$ and calculates $(x, y) = kG$. After that, she converts x to $\bar{x}$ and computes $r = \bar{x} \mod n$ (see [18] for the conversion of field elements to integers and hash outputs to integers). If $r = 0$, she has to repeat step (3) with a different k.

   Otherwise, she has to calculate $k^{-1} \mod n$ to use it in the formula $s = k^{-1}(e + dr) \mod n$. If s is 0, she has to repeat step (3) with a different k. Finally, the signature for the message m is (r, s).

4. Alice sends Q, D, H, m and the signature (r, s) to Bob. The only secret is d.

5. Bob wants to validate the signature. He calculates the hash of m with $H(m)$ and transforms the output to an integer e. Then, he checks if r and s are indeed in the interval $[1, n-1]$ and computes $u_1 = es^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$. With u_1 and u_2, X can be calculated with $X = u_1G + u_2G$. If $X = O$, he rejects the signature. O is called the *point at infinity* and can be calculated with the curve parameters of D (for more details see [18]).

   Afterwards, he converts the x-coordinate of X to $\bar{x}$ and computes $v = \bar{x} \mod n$. The signature is valid if and only if $v = r$.

6. If Alice wants to sign a new message, she can reuse Q, d, D, H. She only has to transmit the new message m and the new corresponding signature (r, s) to Bob, who can then validate the signature again.

Bitcoin uses ECDSA with the secp256k1-curve, which enables users to sign transactions in order to send funds [20] [21]. Ethereum also makes use of ECDSA with the same curve parameters for signing transactions. The algorithm is also available as a precompiled contract in order to write smart contracts with the ability to validate signatures [4].

With the explanation of the cryptographic primitives out of the way, the next chapter describes Ethereum and its inner workings, which makes extensive use of the introduced concepts.

2.5 Proof-of-Work (PoW)

In 1992 Dwork et al. [22] presented an approach with the goal to counteract electronic junk mail. They define a *pricing function* as a function f, which (1) is moderately easy to compute, (2) should not have the property that in the case of l messages $m_1, \ldots, m_l$ the cost of computing $f(m_1), \ldots, f(m_l)$ is comparable to the cost of computing $f(m_i)$ for any $1 \leq i \leq l$, and (3) is easy to the check if $y = f(x)$ for a given x and y.

The sender of an electronic mail has to compute $y = f(H(m, d, t))$ in order to send the message m to the recipient d at time t. H is a cryptographic hash function. Then, he sends the message (y, m, t) to d. The mail program of the recipient verifies $y = f(H(m, d, t))$.

Because of the definition of f this task is easier compared to the sender's task. If (1) the given y is not correct, or (2) the time t significantly differs from the current time, the recipient's mail program discard the message. Otherwise, the program shows the message to the recipient. With this approach, the possibility to send a huge amount of mails is discouraged by the amount of processing work required to calculate a valid y for every recipient. Dwork et al. [22] also outline ways in order to adjust the time difference between calculating $y = f(x)$ and checking $y = f(x)$ for a given y. In addition, an approach is described to introduce a shortcut in order to enable legitimate organizations with the knowledge of a secret c to circumvent the expensive calculation of f to send bulk mails.

Jakobsson et al. [23] argued that the previously mentioned approach to counteract electronic junk mail by Dwork et al. [22] was perhaps the first description of a *proof-of-work*. Next, several definitions regarding proof-of-work are given, which summarizes the formalization stated by Jakobsson et al. [23]. The goal of a prover P is to prove to the verifier V that he has performed computational work within the time interval $[t_s, t_c]$, where t_s is the starting time of the protocol execution and t_c the completion time. It is assumed that the prover may perform computational steps over the time interval $[-\infty, t_c]$. Both parties are able to conduct private coin flips during the protocol execution.

Definition 8. *We say that a proof-of-work algorithm is (w, p)-hard, if (1) the prover P with a total of m memory resources is only able to perform on average at most w steps of computation in the interval $[t_s, t_c]$ over all coin flips conducted by both parties, and (2) the verifier V accepts afterwards the solution with a probability of at most $p + o(\frac{m}{poly(l)})$, where l is a security parameter.*

Definition 9. *We say that a proof-of-work algorithm is (w, p, m)-feasible, if (1) there exists a prover P, such that P is able to perform an average of w steps of computation in the interval $[t_s, t_c]$ with a total of m memory resources, and (2) the verifier V accepts afterwards the solution with probability at least p.*

Jakobsson et al. [23] give the following example of a proof-of-work algorithm and claim, that (1) the example is $(w, \frac{1}{2^k - w}, O(1))$-feasible and (w, p)-hard if $w \in [0, 2^k - 1]$ and $p = \frac{1}{2^k - 1}$.

Example 1. *The verifier V generates x, which is a random string of bits with a length l, and computes $y = H(x)$ with a one-way function H defined as $\{0,1\}^l \to \{0,1\}^l$. Then, V sends (x', y) to the prover P, where x' denotes a string of the first k bits of x. The goal of P is to calculate a preimage $\bar{x}$, which satisfies $y = H(\bar{x})$.*

Note that this section only introduced a simplistic view on a formalized PoW. The paper [23] states additional definitions and proofs in order to formalize proof-of-work more rigorously. The overall goal of proof-of-work is to enable a prover P to prove that a specific amount of computational resources is spent to find a solution, which a verifier V can easily check. Section 3.2 will deal with the proof-of-work algorithm, which is currently deployed in Ethereum.

CHAPTER 3

Technical Background: Ethereum

In this chapter technicalities of Ethereum are elucidated. At first, an overview of the block structure and its important data fields is given. Then the currently deployed proof-of-work is briefly outlined and how clients can verify the correctness of the blockchain. Afterwards, the concept of a smart contract is described, which is used to build applications on top of Ethereum. The last section explains the built-in currency of Ethereum, namely Ether, and the idea of tokens.

3.1 Ethereum Blockchain

Figure 3.1 shows a simplified version of the Ethereum blockchain. A block consists of its transactions and a block header, which includes important variables. To form a chain every block has to contain the hash output of the parent block [4]. Other important fields are the block number, the state-, transaction- and receipt-root, and the nonce value. Root in this context means a root hash value of a tree data structure. The transaction data structure contains all transactions included in a specific block, whereas the receipt data structure includes outcomes of these transactions. The state root shows the outcome of all transactions issued on the blockchain so far. Examples are address balances of accounts or values of variables in deployed smart contracts. The purpose of the nonce value for proof-of-work is described in the next section.

3.2 Proof-Of-Work

Section 2.5 already introduced the general concept of proof-of-work. This section deals with its usage in blockchain projects. Proof-of-work in Ethereum is similar, but a slightly more complex technique compared to Bitcoin. In Bitcoin, a block field named 'nonce' is altered until the hash value of the complete block is numerically below a specific *target value*. The target depends on the current difficulty, which in turn depends on the average

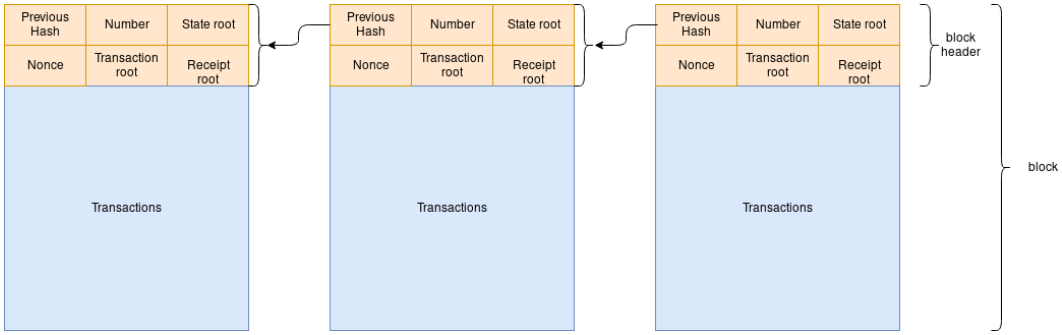

Figure 3.1: Simplified Ethereum block structure

number of blocks generated in the last time interval of a specific size. This approach enables the system to regulate itself by incorporating the change of the miners' computing power over time [1]. For example, if the blocks are generated too fast, the difficulty will increase with the next update of the difficulty value.

The reason, which forces a miner to try out different nonce values, is the *preimage-resistance* property of the hash function (see Section 2.1).

In Ethereum the underlying consensus algorithm is called Ethash. A seed must be computed, which depends solely on the block number. A client, who wants to verify a block, can generate a cache file of about 16 megabytes from this seed. Miners, who want to extend the blockchain, have to generate a 1 gigabyte dataset from the mentioned seed value. The cache and dataset change every 30000 blocks. Like in Bitcoin, a 'nonce' field has to be altered till the hash value of the block meets a certain criterion, but in this case the dataset also has to be used and there are more complex rules to follow. A complete definition of Ethash can be found in [4] and [24]. The reason, why the client has to use a cache file for block validation, is that it is the only possibility to check if the miners have chosen the correct values from the generated dataset. Using huge datasets should discourage the development of specialized mining hardware.

Nakamoto [1] outlines a simplified approach, called SPV, to verify the proof-of-work of the Bitcoin blockchain without validating any transactions. The approach validates every Bitcoin block header and its connection to the previous header. Proof-of-work is checked by computing the hash of every Bitcoin block header and by comparing the result with the target difficulty stored in the corresponding header. The assumption for SPV to work is that the blockchain with the most aggregated difficulty does not contain any invalid transactions, because the client omits checking them. In order to verify a transaction inclusion in a specified block, the SPV client can use the approach outlined in Section 2.2. In Ethereum, the same approach can be applied, but the client has to use the cache files in order to check the proof-of-work.

3.3 Smart Contracts

Luu et al. [25] define a *smart contract* as a program that runs on the blockchain and its correct execution is enforced by the consensus protocol. Rules can be encoded in such a contract in order to allow building different kinds of applications.

Smart contracts can be deployed on the Ethereum network, where an address is assigned to the position of the contract in the blockchain. Users can invoke such a contract by sending a transaction to the specific contract address. On contract invocation different things can happen, like the change of variables or different function calls. The incremental execution of such applications is known as state transition. Ethereum features an execution environment for state transitions with the help of the Ethereum Virtual Machine (EVM) [4]. It specifies how the state is getting changed for a given series of bytecode instructions. The Turing-completeness of the transitions is only bounded by the *gas* parameter, such that the execution of a specific contract stops when the provided gas is depleted. Every operation, like addition or subtraction of contract variables, has a specific gas value [5]. Therefore, the concept of gas is used in order to require users to pay for the execution of smart contracts and to avoid smart contracts to run infinitely if it gets stuck in an endless loop. In order to pay for gas, a transaction in Ethereum has to contain a specific fee.

A programming language called Solidity can be used to develop such smart contracts for the Ethereum network. The documentation [26] of this language explains the constructs of variables and functions in more detail. Functions can be called with the help of transactions to the Ethereum network. Deploying a new contract can be done with a special transaction, which includes the compiled Solidity code.

There are two different types of accounts in Ethereum: *externally owned accounts* and *contract accounts* [5]. Users can interact with Ethereum via an externally owned one, which has an Ether balance (see next section for more details). Contract accounts also have Ether balances, but also contain their smart contract code. Transferring Ether between users typically involves only the interaction between the two corresponding externally owned accounts, whereas the invocation of a deployed smart contract typically involves sending a signed transaction from the externally owned account to the contract account of the specific smart contract.

3.4 Ether and Tokens

The built-in currency of Ethereum is called Ether [5]. It is used to pay for transaction fees, where it has to be first converted to gas. The user has to specify how much Ether the required gas is worth to him for the execution of the transaction. Ultimately, miners decide if the transaction is worth executing. Ether also serves as currency. It is created by mining blocks. This should give miners an incentive to build new blocks on top of the blockchain.

Token systems can be used in different kind of applications [5]. For example, tokens can represent currencies, property, coupons or other assets. Smart contracts are used to implement such a system. Such currencies built on top of Ethereum can have entirely different characteristics compared to Ether, because arbitrary rules can be encoded in the contract. Therefore, the issuance of tokens is controlled by the corresponding smart contract. The contract also defines functions to enable holders of tokens to send them to other addresses. A popular standard of this generic concept is called ERC-20 [1]. The following code, which is taken from OpenZeppelin [27], lists the required functions in form of an interface. The implementation to the interface is necessary in order to be compliant to the ERC-20 standard:

```
pragma solidity ^0.5.0;

interface IERC20 {
  function totalSupply() external view returns (uint256);

  function balanceOf(address account) external view returns (uint256);

  function transfer(address recipient, uint256 amount) external returns (
    bool);

  function allowance(address owner, address spender) external view returns
    (uint256);

  function approve(address spender, uint256 amount) external returns (bool)
    ;

  function transferFrom(address sender, address recipient, uint256 amount)
    external returns (bool);

  event Transfer(address indexed from, address indexed to, uint256 value);

  event Approval(address indexed owner, address indexed spender, uint256
    value);
}
```

The purpose of every function is explained below:

- totalSupply(): returns the sum of all existing tokens in the contract.
- balanceOf(): returns the number of tokens, which a specific address possesses.
- transfer(): allows to transfer a specific number of tokens to a specified address.
- allowance(): this function returns the number of tokens, which another address is allowed to spend on behalf of the token owner
- approve(): this function can be used in order to let a specified address spend a number of tokens from the owner address.

[1] *https://eips.ethereum.org/EIPS/eip-20, Accessed: 2019-05-25*

- transferFrom(): if a user has the right to spend tokens from the token owner, then this function can be used to conduct such transactions.
- Transfer(): this is an event, which should be emitted when tokens are transferred.
- Approval(): this is an event, which should be emitted when tokens are approved by the token holder to be sent by another address.

Token systems are a popular application. Friedhelm et al. [28] identified more than 75,000 ERC-20 compatible token contracts out of more than 7,000,000 created smart contracts on the Ethereum platform. The blockchain also contains more than 97,000,000 Transfer()-events. Tokens can be traded on centralized exchanges, but also decentralized exchanges were built with smart contracts in order to trade tokens without requiring a third party [29].

3.5 Resource consumption

As seen in the previous sections, full nodes have to store and process a lot of data. According to [30], as of 30th August 2019, the complete blockchain has a size of over 170 GB and it is growing with every newly mined block. For the purpose of this thesis a full synchronization from scratch was conducted with an Intel Core 2 Duo E8400 processor with 3 GHz, 4 gigabytes of RAM and an SSD. It took more than two weeks. The synchronization included the verification of all block headers and a check of all included transactions. In contrast to Bitcoin and its simple money-transfer transactions, Ethereum transactions can also invoke smart contracts. Therefore, all contract invocations ever conducted together with the corresponding contract functions had to be executed again during the synchronization. The fast block interval of roughly 15 seconds also contributes to the excessive blockchain growth [31]. Because of the mentioned facts, Ethereum is approaching the limit of average consumer hardware. In order to counteract the blockchain bloat, the next chapter introduces the concept of state and payment channels, which enables to scale up the transaction throughput, but at the same time requiring less transaction data to be stored on the blockchain.

The resource consumption of Ethereum also motivates the construction of light clients. Even SPV verification, which only demands the download and validation of all block headers and Merkle proofs of specific transactions, has to deal with a growth of about 1 GB of data per year (see Section 5.4) and requires therefore more powerful computing devices. Chapter 6 introduces an approach, which should make light clients in Ethereum possible without the need to do an expensive SPV verification.

CHAPTER 4

Technical Background: State and Payment Channels

State and payment channels are a promising solution in order to scale beyond the transaction limit of blockchains. Therefore, this chapter introduces the general concept. A light client has to support this technology to remain useful in the future. In the last section the noteworthy problems are mentioned regarding combining this concept with light clients.

Jourenko et al. [32] provide a taxonomy of different aspects and protocols regarding state and payment channels, which are often named 'layer-2 solutions'. The term *channel* refers to to the approach, where two participants can directly communicate with each other in order to exchange funds. A channel has to be opened first, where both parties commit certain amounts of funds to the channel via transactions to the relevant blockchain network. For example, a specially constructed smart contract in Ethereum can process the transactions and locks the funds for a specific time. The goal of a channel is that the participants can send signed off-chain messages to each other stating new balances, and they are assured of redeeming the messages at a later point with an on-chain transaction, because the smart contract has the functionality to distribute the locked funds at the ratio stated in the given message signed by both parties. Basically, the only interaction with the blockchain is to open and to close a channel. In case of a misbehavior settling the dispute is also done on-chain. Concatenation of channels yields a payment channel network. Networks are used to transfer funds between parties, who do not have a direct channel to each other, with the help of intermediate nodes. The following sections are based on the three defined levels of such constructs outlined by Jourenko et al. [32] (note that the descriptions are adapted to be more relevant to Ethereum):

4.1 Off-chain channels

The simplest channel construct is called *Simplex Payment*, where there only exists one payer and one payee. Funds can therefore only be sent in one direction. The payer opens a channel and specifies the amount of money he wants to lock and the expiration time of the channel. This means that he is not able to access these funds in this time. Then, he transmits signed messages to the payee. The signed messages allow the payee to send the stated amount to his address, but only before the channel expires. The benefit of this approach is, that signed messages can be transferred directly to the payee without the interaction with the blockchain. A more complex construct is called *Duplex Payment*, where both parties have to lock funds in a smart contract. Figure 4.1 shows a possible procedure. Imagine Alice wants to exchange funds with Bob. Both parties have to lock funds in a channel and only the signature of both on a specific message enables them to close the channel and distribute funds according to the values specified in the message. Then, either Alice or Bob can create messages with new balances. To be a valid transaction, both have to sign it. Also in this case an expiration time has to be specified in order to close the channel if one party does not respond anymore. In order to detect if a party wants to transmit an invalid (old) transaction to the blockchain, transactions also have to contain a sequence number. A more generalized version of a channel construct is called *State Channel*. Besides balances, signed messages can contain any form of state, for example variable assignments.

4.2 Network

The usage of payment channels are not enough to provide massive scalability. Imagine the need to open a channel to every party, who should be paid. Participants would have to lock huge amounts of funds, which can not be shared between different channels. A solution to this problem is the concatenation of channels in order to exchange funds between parties, who are not directly connected. Therefore, funds for a transaction are sent over more than one channel: they *hop* along intermediate nodes, which are directly connected, until the payee is reached. The technology, which allows these payments to execute atomically, is called Hash Time Locked Contracts (HTLC). The Lightning network [9], Raiden [10] and Sprites [33] use such constructs. Basically, if node A wants to send a payment to node C via one middle node B, then C has to compute a random number and shares it with A. Afterwards, A sends the payment to B with the condition that B has to show the random number in order to claim the payment and B sends the payment to C with the condition that C has to show the random number. Eventually, C wants to claim the money and shows the secret. Then, B learns the secret from C and can also claim the money. This procedure can be generalized to an arbitrary number of middle nodes. Section 8.2 describes this payment procedure in more detail for the Raiden network.

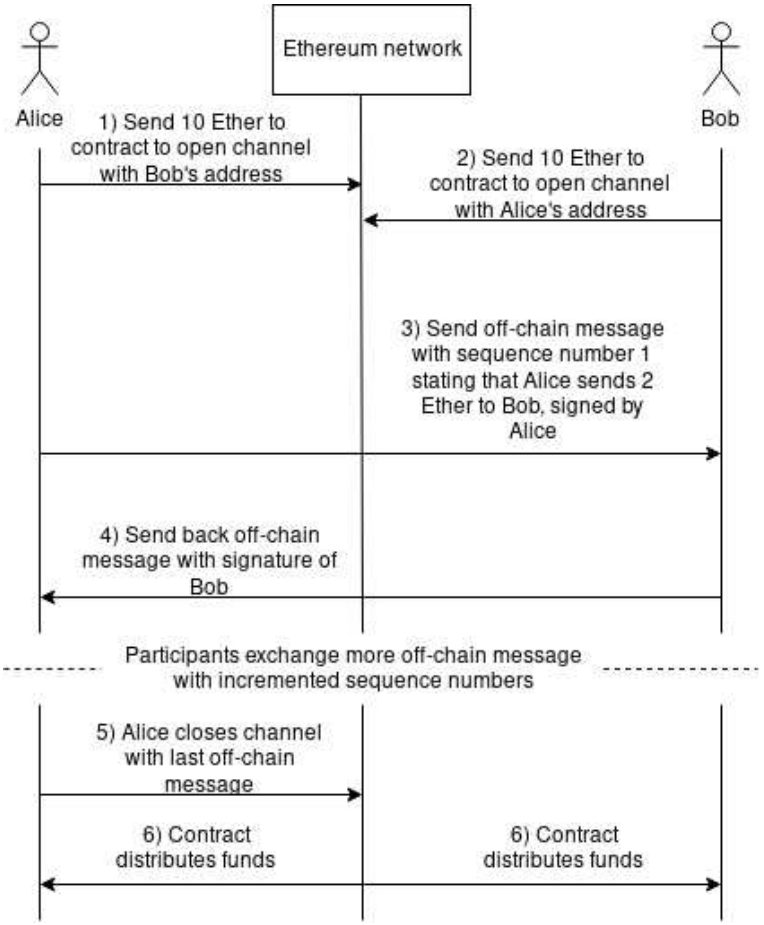

Figure 4.1: Duplex payment channel

4.3 Network Management

Payment network constructions are quite complex because they have to deal with a lot of different issues. Jourenko et al. [32] mention the following:

- **Malicious nodes.** Two parties, who want to exchange funds, have to deal with the case, that intermediate nodes can behave malicious and deviate from protocol rules. Malavolta et al. [34] describe an attack to steal fees from honest intermediate nodes.
- **Routing.** If there does not already exist a channel between two parties, who want to interact, they have to find a route in the network. This is a hard problem and can lead to a high overhead in required communication.
- **Fees.** Because intermediate nodes can request an arbitrary amount of fees in order

to forward funds, the problem to find a cheap route also increases the complexity of the protocol.

- **Fund rebalancing.** Let A and B be two nodes in a network of payment channels. If funds in the network are sent in a non-uniformly way, it can happen that node A has already sent all possible funds to node B. Rebalancing is the procedure to conduct payments in a way such that the funds of the nodes are balancing each other. Solutions to tackle this problem can be found in [35].
- **Node disconnections.** The network has to support the case, that nodes can go offline at any time, because of network or device failures.
- **Privacy and anonymity.** Intermediate nodes should learn as little as possible about conducted payments, because of privacy and anonymity reasons. Privacy is not an afterthought, it has to be built into the core architecture of payment networks. Malavolta et al. [34] emphasis the privacy aspect in their network construction.

4.4 Watchtowers and Light Clients

Using a payment channel is not as simple as a money transfer on the blockchain. The question arises how a participant in a duplex channel can detect, if his counterpart has closed the channel with a signed message containing an old sequence number? The problem is, that by such a behavior the participant can lose money in the belief that the channel is still open while exchanging signed messages with his counterpart. A solution is to continuously look for a channel-closing transaction on the blockchain. This is not suitable for light clients, for example smartphones. Reasons can be that (1) they cannot stay connected to the Internet the whole time while a channel is open or (2) watching the blockchain is too resource intensive. The Lightning network [9] proposes the usage of a third party. A common term for this third party is *watchtower* [36]. In order to incentivize watchtowers to notify misbehavior, the user has to pay them. A favorable property of light clients, as defined in Section 1.1, is the absence of a third party, therefore watchtowers are not considered as a solution for the construction of a light client. Section 11.2 describes how a light client can operate in order to support payment channels, but without the need of watchtowers in order to detect fraud.

There are several use-cases, which motivate the construction of light clients capable of handling payment channels. If a customer wants to pay in a shop, then he is only able to use a light client because of the resource-constrained smartphone. To pay via a blockchain transaction requires confirmation time, whereas a payment over an already opened channel can happen instantaneously. In order to confirm the payment the shop owner only has to check a signed message. To enable the mentioned use-case, the next chapter evaluates light client concepts of different cryptocurrencies and highlights shortcomings of Ethereum on this aspect, which the subsequent chapter tries to solve.

CHAPTER 5

Related Work: Light Clients

In this chapter some approaches of light clients are explained, which aim to validate the blockchain without third-party trust and with the assumption, that the blockchain with the most aggregated difficulty (sum of all difficulty values stored in the blocks) is honest. The goal is to do the verification with a focus on resource minimization, such as CPU/RAM/bandwidth and disk space. At first, light client approaches are explained for several cryptocurrencies: Bitcoin, Zcash, Grin, Beam and Ethereum. Then a comparison of different existing light client wallets is given. A selection criterion is the absence of the necessity to trust a third party. The section afterwards deals with permissioned blockchains and the hypothetical development of a light client in such a setting. In the last section a summary is given, which focuses on Ethereum light clients and their problems, which have to be solved. The goal of this chapter is to evaluate different light client constructions and how their capabilities may be restricted by properties of the underlying blockchain.

5.1 Bitcoin

As mentioned in Chapter 1 light clients in Bitcoin use SPV to synchronize the blockchain [1]. In order to validate the proof-of-work, the client only has to fetch block headers (without the actual transactions), validate header fields like *previous hash* and *difficulty* and check with a single hash calculation, if the block header hash satisfies the target difficulty. To find out the balance of an address, the client can ask full nodes to send back Merkle proofs, which affects the specified address. The client can validate these proofs with the help of the stored block header chain.

A block header has a size of 80 bytes [37]. The system regulates itself by generating 2016 blocks every 2 weeks on average [38]. For a 52-week year, these facts result to an increase of only about 4 megabytes per year. Due to the low memory consumption and ease of

verification, the SPV approach in Bitcoin is feasible for resource-constrained devices and further developments are not crucial.

5.2 Zcash

Zcash was built on top of the existing Bitcoin implementation with the additional feature of privacy preserving transactions [39]. Other features are the change to the memory-hard proof-of-work algorithm called 'Equihash' and the reduction of the block interval to 2.5 minutes.

Wüst et al. [40] argue that it is not possible to directly use the SPV verification algorithm in Zcash in order to validate shielded transactions. The reason is, that the client would need to transmit the decryption key to the server for the decryption of corresponding transactions, which would render the privacy property useless. Another problem is the fact, that the client needs a Merkle proof to received funds in order to build a transaction. In Zcash this proof is a non-static information. This means that it is changed with every new transaction, which is added to the Merkle tree. The authors of the mentioned paper propose a system called 'ZLiTE', which helps to support light clients by outsourcing computation of sensitive data with the help of a so-called Trusted Execution Environment (TEE), like Intel SGX.

Intel SGX stands for Intel's Software Guard Extensions [41]. It consists of hardware, where computation and data cannot be observed from outside, such as from the operating system or other applications. Imagine Alice does not trust Bob, but she wants to perform computation on Bob's server, which is equipped with SGX compatible hardware. SGX uses a concept called software attestation. This enables Alice to communicate with the hardware, and she can be assured that she is really exchanging messages with this specific hardware. The reason is, that SGX can send a signature along with the computation data to Alice. The signature is generated with SGX's private key, which is only known to SGX and the hardware manufacturer (in this case Intel). Alice can check the signature for its validity with the help of Intel, because the hardware manufacturer serves as a certificate authority. Furthermore, the communication between Alice and SGX can be encrypted, so that she can securely outsource computation of sensitive data to this hardware, get results back and verify that the computation was actually done in SGX.

Wüst et al. also outline disadvantages of such a hardware:

- The hardware manufacturer has to be trusted to design a secure processor and also to be honest about the certification process. Attestations from the hardware must not be forgeable.
- Often the hard- and software are not open source.
- Design flaws, which enable side channel attacks or other physical attacks, have to be avoided.

ZLiTE enables light clients to create and receive shielded transactions with the help of Intel SGX. If the trusted hardware is completely compromised, the attacker is still not able to steal funds, but the client loses the privacy property. In addition, he cannot be assured anymore that he has complete information about payments, because an attacker can omit transactions.

5.3 Grin and Beam

Grin [1] and Beam [2] are two cryptocurrencies, which make use of Mimblewimble. Poelstra [42] claims that Mimblewimble allows to remove most historic blockchain data while users are still able to fully verify the chain. The approach defines an additional *compact* blockchain besides the normal blockchain with significantly smaller size, such that clients only need the reduced version and a specific amount of last blocks of the normal blockchain in order to verify the correctness of transactions of the complete chain. Other goals are confidential transactions and capabilities to obfuscate the transaction graph. According to Bünz et al. [6] these two blockchain projects already use a Merkle Mountain Range tree in their block header, therefore the FlyClient's approach can be directly used to build efficient light clients.

5.4 Ethereum

Currently, the two most popular Ethereum nodes are Geth [3] and Parity [4]. In order to allow for a similar verification to SPV in Bitcoin, Geth can be used with the "–syncmode "light"" flag. In this mode it only stores the header chain and requests other data from full nodes on demand. For the Parity client the "–light" flag has to be used in order to do the same thing.

Assuming that the average block interval is about 15 seconds [31] and the block header size is 508 bytes [6], Ethereum produces about 1 gigabyte of block headers in a 52-week year (compared to about 4 megabytes for Bitcoin). In addition to the huge increase in data (508 bytes every 15 seconds) compared to Bitcoin (80 bytes every 10 minutes), the block validation is also more complex (see Section 3.2). These circumstances make Ethereum unusable on resource-constrained devices, which want to conduct an SPV verification of all existing block headers in a reasonable time. For this reason, Geth and Parity do not start with the Genesis block (which is the first block in the blockchain), but with a specific number of blocks behind the latest known block number [43]. This allows to synchronize the chain at faster pace.

In order to further minimize SPV validation time, Kiayias et al. [44] propose Non-Interactive-Proofs-of-Proof-of-Work (NiPoPoWs). It is a verification mechanism with a

[1] *https://www.grin-tech.org/, Accessed: 2019-05-31*
[2] *https://www.beam.mw/, Accessed: 2019-05-25*
[3] *https://geth.ethereum.org/downloads/, Accessed: 2019-05-25*
[4] *https://www.parity.io/ethereum/, Accessed: 2019-05-25*

runtime logarithmic in the length of the blockchain. This approach requires a hash of a specific data structure to be added in every block header. The data structure defines links to so-called 'super blocks' (blocks with higher difficulty compared to normal blocks).

The FlyClient [6], which is described in the next chapter, improves the technique of Kiayias et al. They argue, that NiPoPoWs have some disadvantages: (1) the succinctness property is only fulfilled as long as no adversary attacks the blockchain and (2) the technique can only be used in blockchain projects with a fixed block difficulty. A fixed difficulty cannot be found in most projects, because of the need to adjust the block generation interval to the variable hashrate in order to obtain a near constant average block generation time.

FlyClient promises to verify the blockchain with a runtime logarithmic in the chain length in order to enable Ethereum light clients. This property is also preserved in the presence of adversaries. Additionally, it also works in blockchain projects with changing difficulties.

5.5 Comparison

In this section several light client implementations of popular cryptocurrencies are compared with each other. In order to be comparable, only clients with SPV verification are tested. Clients, which require third-party services, are not considered here. Research was done on more cryptocurrencies, but they presumably do not have light clients. It seems that Monero does not have an SPV client, because the privacy-property of the currency would then be violated [5] [6]. For Zcash it is also problematic, because giving up the privacy-property on a privacy-focused currency cannot be tolerated (see 5.2). Table 5.1 shows some wallet implementations for different cryptocurrencies. Information about the usage of a third-party entity in a light client implementation is often not clearly stated on its official website. The blockchains in the table are not chosen dependent on some specific criteria. In fact, it appears, that the mentioned blockchains in the table are the only ones, which have light client implementations that do not require a third party. Bitcoin Cash and Litecoin are similar to Bitcoin, therefore the existence of a light client is not a surprise. The light client functionality of Geth and Parity are also mentioned because they are the only Ethereum implementations, but they are not as efficient as the other ones.

[5] *https://www.reddit.com/r/Monero/comments/8y9bv1/question_is_mymonero_desktop_wallet_a_spv_one_or/, Accessed: 2019-06-30*

[6] *https://monero.stackexchange.com/questions/10866/are-there-any-neat-tricks-to-enable-spv-like-functionality-for-monero, Accessed: 2019-06-30*

[7] *https://github.com/bitcoin-wallet/bitcoin-wallet, Accessed: 2019-06-29*

[8] *https://github.com/bitcoinj/bitcoinj, Accessed: 2019-06-30*

[9] *https://bitcoinj.github.io/limitations, Accessed: 2019-11-13*

[10] *https://electrum.org, Accessed: 2019-06-30*

[11] *https://www.reddit.com/r/Bitcoin/comments/3c3zn4/whats_the_difference_between_an_api_wallet_and_a/, Accessed: 2019-06-30*

Name	Cryptocurrency	Description	Used Infrastructure
Bitcoin Wallet [7]	Bitcoin	This wallet uses the Java library *bitcoinj* in order to interact with full nodes [8]. Bitcoinj uses the Bitcoin peer-to-peer network to retrieve blockchain data and it uses the already introduced SPV verification [9].	Bitcoin peer-to-peer network
Electrum Bitcoin [10]	Bitcoin	Other than *bitcoinj*, this wallet uses Electrum servers in order to fetch blockchain data [11] [12]. Compared to centralized services, Electrum servers can be set up by anyone and the client conducts an SPV validation in order to verify gathered information [13].	Electrum servers
Electrum Litecoin [14]	Litecoin	Litecoin [15] is another popular cryptocurrency, which can be used inside an Electrum wallet. The wallet uses SPV validation and interacts with Electrum servers.	Eletrum servers
Electron Cash [16] [17]	Bitcoin Cash	Bitcoin Cash is a fork of Bitcoin. They are not very different regarding the offered features. Therefore, this wallet for Bitcoin Cash uses a concept similar to Electrum.	Electron servers
Geth/Parity in light mode	Ethereum	As already mentioned in the previous section Geth and Parity support light client functionality via SPV verification, although in Ethereum disk requirements are much higher for this type of synchronization compared to other cryptocurrencies. Other full-nodes in the Ethereum network are used in order to fetch blockchain data.	Ethereum peer-to-peer network

Table 5.1: Comparison of different light client implementations

5.6 Permissioned Blockchain

De Angelis et al. [45] define a permissioned blockchain as a chain, where an additional authentication and authorization layer for miners is deployed. In this setting, different types of consensus algorithms can be used, for example Byzantine fault tolerant (BFT), Practical BFT (PBFT) or Proof-of-Authority (PoA). PoA is an algorithm, which is available in Parity and Geth for the setup of permissioned Ethereum chains. In Parity this algorithm is called *Aura*, whereas in Geth it is called *Clique*.

In PoA there exists a set of nodes called authorities, which are the only ones allowed to create new blocks. It is assumed, that more than 50% of them are honest. A malicious entity can be removed from the set of such nodes by conducting an election where a majority of votes is required.

If we assume that the set of authorities is always the same, then a hypothetical light client is straightforward to construct. The client can always trust the obtained message if it is signed by a majority of authority nodes. Otherwise, the client has to follow the chain from the genesis block to the current last block in order to find out about the latest valid set of authority nodes. This procedure is required, because the set of nodes is dynamic and the validity of a signature depends on the valid set of nodes at the time the signature was created.

5.7 Summary

As outlined in Section 5.1, Bitcoin light clients currently only have to deal with a low amount of data, only about 4 megabytes per year, because of the small block header and the slow block generation. The additional absence of privacy features, such as private transactions in Zcash or Monero, simplifies the development of a Bitcoin light client. Other cryptocurrencies, like Bitcoin Cash and Litecoin, are similar to Bitcoin and its resource friendliness, therefore they also have light client implementations.

In Ethereum the amount of data for an SPV verification is quite high. Section 5.4 states a data growth of about 1 gigabyte per year. Therefore, resource-constrained devices have to find another solution in order to verify the Ethereum blockchain. In the next section a new approach called FlyClient is introduced with the goal to minimize the required verification data considerably.

[12] *https://www.reddit.com/r/btc/comments/aubq4x/bitcoin_cash_spv_wallet_options/, Accessed: 2019-06-30*

[13] *https://www.reddit.com/r/Bitcoin/comments/2feox9/electrum_securityprivacy_model/, Accessed: 2019-06-30*

[14] *https://electrum-ltc.org/, Accessed: 2019-06-29*

[15] *https://litecoin.org/, Accessed: 2019-06-30*

[16] *https://electroncash.org/, Accessed: 2019-11-13*

[17] *https://github.com/Electron-Cash/Electron-Cash, Accessed: 2019-06-29*

CHAPTER 6

Technical Background: FlyClient

Bünz et al. [6] argue that light clients in Ethereum are no longer practical. The reason for this statement is, that a client has to download more than 3.6 gigabyte of data for a simple payment verification (SPV). The amount of data is growing continuously with every new block. In addition, the storage and bandwidth requirements increase linearly with the blockchain length. The goal of this chapter is to summarize important concepts of FlyClient, which is a technique presented by Bünz et al. [6] that should help to overcome these issues.

Note that this thesis is based on the FlyClient paper, which was published on February 28th, 2019 [1]. The existence of newer versions was recognized at a time, where it was already too late to incorporate the changes made in those versions. The newer versions incorporate adversaries exploiting the *difficulty raising attack* as described by Bahack [46] and their possible mitigations.

6.1 Functional principle

In order to be able to use FlyClient, a modification of the block header format of Ethereum has to be made, namely the root hash of the Merkle Mountain Range tree specific to the current block header has to be included. To get this root hash, the MMR of the previous block is used, where additionally the previous block is inserted as a new leaf. To illustrate this procedure the hash of the MMR with one element in Figure 2.2 would be in the second block of the blockchain, where the MMR only consists of the Genesis block. The third block would contain the MMR with two elements (the Genesis block and the second block) and so on.

[1] *https://eprint.iacr.org/eprint-bin/versions.pl?entry=2019/226, Accessed: 2019-11-14*

A node in the Merkle Mountain Range tree also includes the aggregated difficulty. Leaf nodes represent blocks of the chain, therefore the block difficulties are stored in these nodes. Intermediate nodes and the root nodes store the sum of the difficulty of their children respectively. The need of the difficulty in every node comes from the requirement, that blockchains with variable difficulty should be supported by the FlyClient (see *Handling Variable Difficulty* in [6]).

The MMR root hash in a block serves as a commitment to all previous blocks in the blockchain. Imagine a protocol between Alice and Bob:

1. Alice asks Bob for the last block X in the blockchain.
2. Bob replies with the last block X, which happens to be block number 1000000. The included MMR root hash of this block is denoted by X_{mmr}.
3. Alice does not trust Bob, therefore she asks for random blocks in the blockchain between block number 0 and 1000000. She also asks for the corresponding Merkle proofs, which she uses in order to validate the inclusion of these blocks in X_{mmr}.
    - A Merkle proof for the block Y is a path from the leaf element (hash of block Y) up to the root hash X_{mmr} in the MMR. The path consists of the leaf and its sibling, the sibling of the node resulting from the concatenation and hashing of the two previous nodes and so on until the root hash is reached. If the calculated root hash is the same as X_{mmr} Alice can be assured that X_{mmr} contains block Y (see *proof of membership* in Section 2.2). Figure 6.1 shows an example of six inserted elements. To prove the inclusion of leaf L4 given root N5, the nodes L4,L3,N1 and N4 have to be provided.
4. In order to increase the probability that Bob is not cheating, Alice can repeatedly ask for random blocks and validate their Merkle proofs.

The outlined procedure is interactive, because Alice has to request different random blocks from Bob. Section 6.5 introduces a non-interactive approach, where Bob can construct a proof without the interaction with Alice.

6.2 Security guarantees

In order to reason about the security guarantees of the FlyClient's approach, Bünz et al. [6] defines the following problem together with the threat model:

- There exists a blockchain network, where honest miners build on top of the chain with the most difficulty.

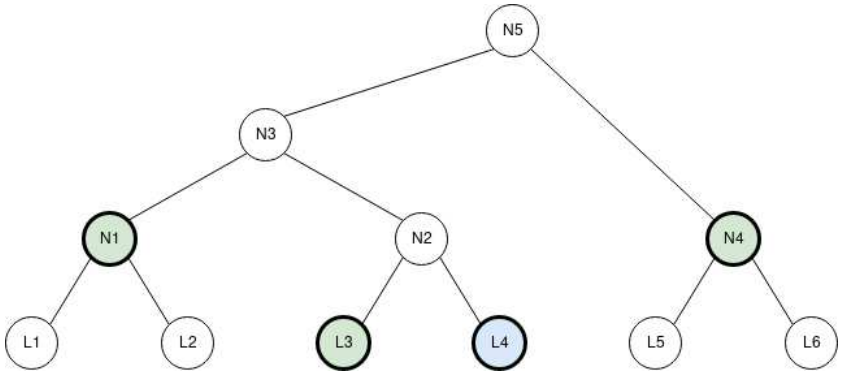

Figure 6.1: Merkle proof for leaf L4

- A client is connected to a number of full nodes, where at least one node is honest. The client cannot distinguish between honest nodes and adversaries without validating the proofs.
- An adversary can corrupt full nodes of his choice. Adversaries are also allowed to build on top of any block of the chain.
- The client is not vulnerable to eclipse attacks and the messages sent between client and full nodes cannot be altered by the adversary.
- The client only knows about the genesis block of the chain. The goal is to verify that a specific transaction is included in the valid chain.

Important parameters for adjusting the security guarantees are:

- $c \in [0, 1)$: c specifies the mining power of the adversary as a fraction of the combined honest mining power. $c = 0.5$ means, that the adversary has half of the total honest mining power. This results to a third of the total mining power of the complete network. In other words, if $c = 0.5$, the adversary has $\frac{1}{3}$ hashing power, the total honest hashing power is $\frac{2}{3}$, and the sum of these gives the total hashing power of 1.
- $L \in \mathbb{N}$: The number of the last, manually checked blocks are denoted by L. $L = 100$ means that the client checks 100 of the last blocks of the chain with probability 1.
- $\lambda \in \mathbb{N}$: An adversary, which controls a fraction of at most c of the honest mining power, succeeds in cheating the client with probability at most $2^{-\lambda}$.

6.3 Sampling

The adversary is bound by the hash power of c, so he can only correctly mine a c fraction of blocks after he has forked the blockchain at an arbitrary block. Because of the reason

that the adversary has to provide a blockchain of the same length or longer compared to the total honest hash power in order to get accepted by the client, he has to create invalid blocks. The verifier does not know, after which block the adversary started the fork. Therefore, he can sample blocks in a way, such that the probability to detect an invalid block is maximized.

A sampling distribution can be calculated, which specifies the strategy how the client should ask for blocks in order to detect cheating. The optimal sampling distribution is given by the following formula:

$$g(x) = \frac{1}{(x-1)\ln(\delta)} \tag{6.1}$$

where:

- $\delta \in (0, 1)$ is the relation of the last, manually checked blocks L to the complete number of blocks n in the chain, therefore δ can be calculated with the following formula: $\delta = \frac{L}{n}$.
- $x \in [0, 1)$ is a relative position in the complete blockchain. $x = 0.5$ means the block X in the blockchain, where the sum of the blocks before X is the same as the sum of the blocks occurring after X.
- $g(x)$ is the probability density at point x. This means that the formula outputs the probability, with which the client should sample the block in position x of the chain.

The number of queries, which are necessary to conform with the security parameter λ, can be calculated with:

$$m \geq \frac{\lambda}{\log_{1/2}\left(1 - \frac{1}{\log_c\left(\frac{L}{n}\right)}\right)} \tag{6.2}$$

where:

- n is the number of blocks in the blockchain
- m is the required number of queries

6.4 Variable Difficulty Handling

The procedure described so far only works, if the difficulty of each block is the same. This is obviously not the case in real blockchain projects, because of the need to handle a variable hashrate and, at the same time, to hold the block generation time constant.

In order to support blockchains with a variable difficulty, the formula 6.1 can be reused to sample the blockchain optimally. The difference lies in a change of the meaning of the input variables, which are described below:

- $\delta \in (0, 1)$ is the relative, aggregated weight of the last L blocks, which are queried with probability 1. The aggregated weight of these L blocks can be calculated by summing up their difficulty. In order to obtain δ, the aggregated weight has to be divided by the sum of the difficulty of all existing blocks in the blockchain.
- $x \in [0, 1)$ is a relative, aggregated weight. $x = 0.5$ points to the block X in the blockchain, where the sum of the difficulty of all blocks before X is the same as the sum of the difficulty of all blocks occurring after X.
- $g(x)$ is the probability density at point x. This means that the formula outputs the probability, with which the client should sample the block, which occurs in the position of the aggregated weight x of the chain.

The paper also describes a modification of the Merkle Mountain Range tree, which is necessary to support handling variable difficulties. Figure 6.2 shows, compared to Figure 6.1, that the difficulty has to be stored in every node of the MMR. Difficulties in leaf nodes represent the difficulty values in the corresponding blocks in the chain, whereas intermediate nodes and the root node show aggregated difficulties, which consist of the sum of the difficulties of their child nodes.

Besides checking the MMR proofs of requested blocks, the client additionally has to check, if the block also has the stated aggregated difficulty. The modification of the MMR allows this procedure, where the client can be assured, that the block is indeed on the position x, for which the client requested the block for.

6.5 Non-Interactive Protocol

In order to create a non-interactive proof, Bünz et al. [6] suggest extracting the randomness from the last block (Bonneau et al. [47] formalize the extraction of randomness from block headers). They argue that the FlyClient protocol is an interactive public-coin protocol, because the client chooses blocks randomly from a known probability distribution. In simple terms, an interactive public-coin protocol consists of a verifier V with limited computing power and a prover P with more computing power compared to V [48]. Interactive in this context means that these two entities exchange messages with each

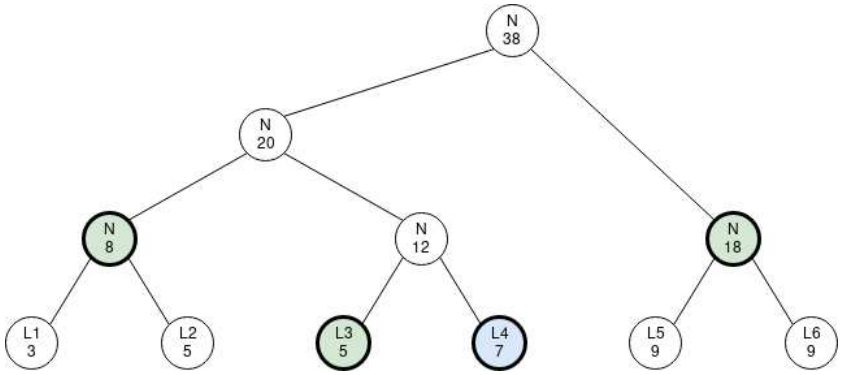

Figure 6.2: Merkle proof for leaf L4 with aggregated difficulty

other with the goal, that the prover wants to convince the verifier about the correctness of a statement. The verifier has access to a coin in order to use it as a source of randomness. Public-coin means that the verifier makes the result of the coin toss public, whereas in private-coin protocols this is not the case.

Bünz et al. [6] further argue that the Fiat and Shamir technique [49], which enables the transformation of interactive public-coin protocols to non-interactive ones, can be applied to Flyclient. To achieve this, outgoing messages of the verifier are replaced with a query to a random oracle H. For the FlyClient's approach this means to use the hash of the latest block as a provider of random bits instead of H. The non-interactive variant is secure, if the following formula is satisfied:

$$p_m < \frac{2^{-\lambda}}{c * n} \tag{6.3}$$

where:

- λ is the security parameter.
- c is the fraction of hash power of the adversary compared to the total honest hash power.
- n is the length of the blockchain.
- p_m is the probability of not detecting a misbehavior of the adversary after m queries. p_m is already $\leq 2^{-\lambda}$ after calculating the required queries with formula 6.2. Therefore, in order to coincide with formula 6.3, m has to be increased. In order to calculate p_m, the formula 6.4 can be used.

$$p_m = \left(1 - \frac{1}{\log_c\left(\frac{L}{n}\right)}\right)^m \tag{6.4}$$

The reason, why the usage of the latest block hash is secure, is the fact, that the adversary has to recalculate a new valid block every time he is not satisfied with the resulting m blocks to query for creating the proof. The hash power c of the adversary is used to price in the attempt to alter the latest block hash, which formula 6.3 shows.

6.6 Summary

In order to deploy the FlyClient, a change in the Ethereum block header has to be made to include the root hash of a Merkle Mountain Range tree. Miners have to extend this tree by adding new blocks as they appear and include the new root hash in a new block to mine. The hash value in the block header serves as commitment to all blocks in the blockchain so far. The following procedure between a potential light client and full node describes all necessary steps in order to validate a FlyClient proof in a non-interactive way and to be assured, that the full node is not cheating:

1. For the current block X with root hash X_{mmr} of the Merkle Mountain Range tree, the full node calculates, with the help of formula 6.2, the number m of required blocks, which have to be inserted in the proof to convince the client. Because the proof should be non-interactive, m has to be increased in order to coincide with formula 6.3.

2. The full node takes the hash of X in order to generate the randomness. Formula 6.1 is used in order to sample according to the optimal sampling strategy and to obtain the blocks in the chain, which have to be inserted in the proof along with their Merkle proofs.

3. Finally, the full node transmits this proof together with the last L blocks to the client. The client can check, if the proof is actually the correct proof for block X by regenerating the randomness corresponding to the specific block hash. Then, this randomness is transformed with formula 6.1 to the specific blocks, which have to be present in the proof along with their Merkle proofs. Then, the client checks the Merkle proofs in order to be assured, that the proof is valid. He also has to check the last L blocks. The proof is valid if and only if no checks fail.

The proof-of-concept in Chapter 9 implements the introduced FlyClient's approach. In order to reason about the security of the implementation, the next chapter defines the problem and corresponding threat model, which is similar to the definitions in this section.

CHAPTER 7

Problem Definition and Threat Model

Because this paper is based on the FlyClient's approach, it uses a similar problem definition and threat model. At first the problem is defined in order to reason about the environment, where a client wants to interact with the Ethereum network. Then the threat model is introduced, which states the capabilities of possible attackers.

7.1 Problem definition

The network consists of nodes, which store the complete Ethereum blockchain. The nodes exist as computers, which exchange information about the blockchain with other computers over the Internet in a peer-to-peer setting. Every node in this network can agree on the same Genesis block. This block is the first block in the chain. It is assumed that only the Ethash-based proof-of-work [24] is used in order to agree on valid blocks to extend the blockchain (see Section 3.2). Miners have more incentive to build on the chain with the most aggregated difficulty. This means that they act rationally with the desire to obtain new coins for their mined blocks. In order to ensure that the longest chain is valid, the honest miners have a majority of mining power in this setting. Only the longest, valid blockchain is considered as the right one, therefore mining on shorter blockchains does not have the effect of earning new coins.

A client wants to connect to this network. The knowledge of the client is limited to the Genesis block. It is assumed, that every entity interested in Ethereum is capable of retrieving the knowledge of the correct Genesis block. Another assumption is, that clients are always able to connect to at least one honest node. The goal of the client is to retrieve the latest valid state of the blockchain in the shortest possible time. The state of the blockchain is identified by the hash value of the last block in the chain. This chain is

characterized by the fact that from the Genesis block to the last added block only valid data is present according to the Ethereum protocol. The chain has to be the longest one in this form. The client does not have the possibility to download and validate the complete blockchain, because this approach is too resource intensive. It is assumed, that the client does not have to validate all conducted transactions in order to be convinced that the blockchain is valid. It suffices to get convinced about the correctness of the longest chain in form of its proof-of-work. In other words, the client accepts the chain, if he thinks that he has found the longest blockchain in terms of aggregated difficulty. This comes from the fact, that it is assumed that honest miners are building on the longest chain and there is a majority of honest miners. Finally, it is also assumed that clients also do not have the capacity to download and process all existing valid block headers. This would be an SPV verification, but, as already mentioned in Chapter 6, the client cannot handle the resource demand of SPV in Ethereum.

7.2 Threat model

It is assumed, that adversaries are not able to generate collisions for the used hash functions and that these functions are also *preimage-resistant*. Adversaries are also not able to forge signatures with the deployed digital signature scheme. It is crucial that guarantees about hash and signature functions can be upheld, otherwise there would be a risk of adversaries breaking the complete blockchain, as outlined in Section 12.1 and Section 12.2.

Like in Section 6.2 an adversary can obtain control of nodes and can build chains on top of arbitrary blocks. Nevertheless, the adversary cannot reach more than $c \in [0, 1)$ fraction of the honest mining power. The open interval on the right side means, that adversaries can get arbitrarily close to 50% hashing power, but it is not possible to obtain a power of exactly 50% or more. This means, as already stated in the previous section, that honest miners have the majority of mining power and therefore the longest chain in the long run. The adversaries cannot alter or suppress messages sent between client and other nodes over the Internet. This assumption is necessary in order to get correct information from at least one honest node, to which the client can connect to by definition. Because of the probability aspect of the FlyClient algorithm, a security parameter $\lambda \in \mathbb{N}$ has to be defined, such that the adversary only succeeds with probability at most $2^{-\lambda}$ to cheat the client. The client can set this parameter to an arbitrary value so that the proofs coincide with his safety expectation.

The thesis defines third-party trust as follows: if it can be shown that the client can successfully retrieve the latest valid block header with the assumptions defined in this chapter, then the client does not depend on third-party trust. In [6] the FlyClient's approach has been already proven to conform to the introduced trust model.

Third-party trust is also an important property of the following chapter, where existing Ethereum applications are evaluated in terms of their security assumptions.

CHAPTER 8

Evaluate Existing Applications

This chapter focuses on existing Ethereum applications and possible problems regarding the deployment on a resource-constrained device. Either a resource-intensive Ethereum full node has to be used in order to allow these applications to work in a decentralized way or a light client has to be used, which depends on a third party. The motivation for this chapter is to outline this trade-off, whereas the next chapter introduces a light client, which works on a resource-constrained device without requiring a third party.

One type of applications are simple money-transfers, where Ether and tokens are sent to different addresses. These transactions can be conducted with the help of wallets, which are explained in the next section. Afterwards, the functionality of a payment channel implementation called *Raiden* [10] is described, which promises fast and cheap transfers of Ether and tokens. The last section mentions other existing, interesting Ethereum applications. The goal of this chapter is to take the decentralization aspect into consideration. Because Ethereum is advertised as a secure and decentralized platform [4], fields of applications should be highlighted were these principles are clearly violated in terms of the previously defined threat model.

8.1 Ethereum Wallets

Some wallet applications designed to send and receive Ether and tokens are shown in Table 8.1. *Own private keys* is true if and only if the user is the only one, who has access to the funds. *Require third-party trust* is true if and only if the user has to trust a third party. Jaxx is a light client, which supports over 80 cryptocurrencies. This is only possible because the application only communicates with its trusted backend services, therefore the client cannot detect if Jaxx is cheating about the state of the Ethereum blockchain [50]. Therefore, third party trust is clearly required. Third party trust is also required for MyEtherWallet, which features a client-side interface to communicate with the blockchain, because the state is also retrieved by a number of nodes controlled

Name	Own private keys	Require third-party trust
Jaxx [2]	Yes [3]	Yes
MyEtherWallet [4]	Yes [5]	Yes
Coinbase [6]	No [7]	Yes
Metamask [8]	Yes [9]	No, if it uses a local Ethereum node. Yes, if it uses remote nodes instead.
Geth or Parity	Yes	No, because these clients can validate all blockchain data, but also need a lot of computing power and bandwidth to do so.

Table 8.1: Comparison of different Ethereum wallet applications

by MyEtherWallet [51]. Coinbase is a digital asset exchange company and also has an integrated wallet on its website [52]. Third-party trust is also required in this case. The Metamask plugin trusts nodes operated by Infura [1] [53], which requires third-party trust. Another option to use Metamask is to connect to a local Parity or Geth node. In this case the client can detect adversaries bounded by the defined threat model, because Parity and Geth can validate the blockchain headers and can validate the state of the longest chain, which is built with the honest hashing power.

The diverse wallet applications can be classified in different types, from low security to high security:

- *Centralized.* The user interacts with a centralized service. This service is also responsible for protecting the funds, but in case the service behaves maliciously the user has no way to stop it from draining the funds. Legal actions are not considered

[1] *https://www.infura.io, Accessed: 2019-05-27*
[2] *https://www.jaxx.io/, Accessed: 2019-05-27*
[3] *https://www.jaxx.io/security, Accessed: 2019-06-30*
[4] *https://www.myetherwallet.com/, Accessed: 2019-05-27*
[5] *https://kb.myetherwallet.com/en/diving-deeper/what-happens-if-mew-goes-down/, Accessed: 2019-06-30*
[6] *https://www.coinbase.com/, Accessed: 2019-05-27*
[7] *https://support.coinbase.com/customer/portal/articles/1526452-where-can-i-find-the-private-keys-for-my-wallet-, Accessed: 2019-06-30*
[8] *https://www.metamask.io/, Accessed: 2019-05-27*
[9] *https://ethereum.stackexchange.com/questions/39954/does-metamask-store-private-key-on-server-or-anywhere-else, Accessed: 2019-06-30*

here. Examples: Coinbase, Kraken [10], Bitfinex [11].

- *Own keys but third-party information.* This type enables (forces) users to manage their private keys on their own, which means that the required third-party service is not able to directly steal funds (if the client-side code can be trusted). The information about the blockchain is gathered from third-party servers, therefore the user has to trust this information. Examples: Jaxx, MyEtherWallet, Metamask.

- *SPV.* This type is similar to the previous type. In addition, the user has the possibility to validate obtained information with the help of SPV. Therefore, he can validate the headers of the blockchain and can choose the chain with the most accumulated difficulty. According to the defined threat model this type is already safe to use without trusting a third party. Examples: Geth and Parity in light client mode.

- *Full-node.* This type guarantees the most security. The difference to the previous type is the additional property that all transactions are validated. Examples: Geth and Parity.

8.2 State and Payment Channels

It appears, that Raiden [10] is the sole actively developed payment network implementation on Ethereum. The project states that it is able to send ERC-20 tokens (see Section 3.4) with fees of an order of magnitude lower compared to normal blockchain transactions. Also, these transactions can be confirmed within a subsecond and this technology scales linearly with the number of participants.

According to the official documentation [54] an Ethereum client is required to power the Raiden software, for example Geth, Parity or a connection to a remote client (e.g. Infura) via RPC. An alternative is the Raiden light client [55], which can be connected to Metamask. Because Raiden only works with ERC-20 tokens, Ether have to be wrapped in such a token with W-ETH (wrapped Ether) [56].

The different steps, which are needed in order to transfer tokens in the Raiden network, are explained below:

1. *Bootstrap or join a token network.* If there does not exist a token network for a specific ERC-20 token yet, then it has to be created at first in order to allow other nodes to join this network. The network is required in order to open new payment channels between two parties. Basically, this procedure deploys smart contracts on the Ethereum blockchain for the specific ERC-20 token.

[10] *https://www.kraken.com/, Accessed: 2019-06-30*
[11] *https://www.bitfinex.com/, Accessed: 2019-06-30*

2. *Open a channel.* To open a channel, information is required about four things: (1) the address of the token to transfer, (2) the address of the partner node to build a channel with, (3) the amount of tokens to deposit into the channel, and (4) the timeout period of the channel. If one entity has established a new channel, then it can send payments to the other entity. In order to get a bidirectional channel, the other entity also has to deposit tokens in the corresponding smart contract.

3. *Make transfers.* In order to transfer tokens, only the target address and the amount, which should be sent, have to be specified. If the amount of tokens is available and there exists a path in the network with enough capacity, then the payment goes through and the receiving node should then be able to see the incoming payment. We assume here that there are no malicious entities and no network or device failures (in order to simplify the explanation and omit details about error handling). Figure 8.1 shows an example of the network processing a payment from node A to node D. *Locked Transfer* messages have to be sent through the channels to the target node, which should receive the payment. This message reserves the amount of pending payment in each used channel. After the target node received the message, it requests a secret from node A in order to unlock the message. Node A then reveals the secret to node D, so that D can claim the payment stated in *Locked Transfer* from node C. It then sends the secret to C, which can claim the payment made from node B. C sends the secret to B and B finally sends the secret to node A. After this procedure the payment from node A to node D has been successfully processed and every intermediate node has been paid and has paid the next node in the path.

4. *Close a channel.* If participant 1 wants to close the channel, the settlement timeout period gets started. In this period, participant 2 has the chance to prove misbehavior in case participant 1 has closed the channel with an invalid state. After that period, or if the blockchain has resolved a possible dispute, the channel is closed and the balances of the participants are updated to the last state on the blockchain.

There exists another payment channel implementation called µRaiden [57]. The difference to the Raiden network is the fact, that this is a simple channel implementation, which features only direct, unidirectional token transfers. Networks of such simple channels cannot be constructed within µRaiden. Also in this case an Ethereum node or the Metamask plugin is necessary in order to interact with the smart contracts on the blockchain [58].

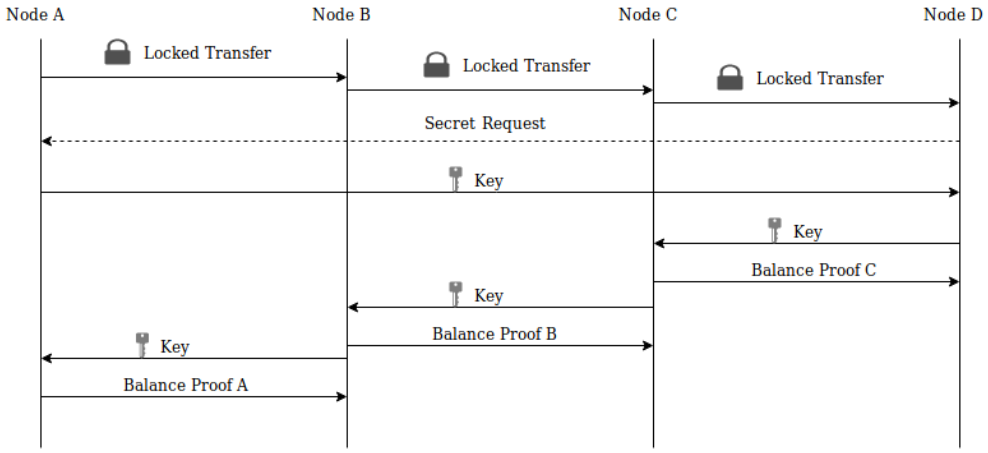

Figure 8.1: Payment in the Raiden network

8.3 Other popular Applications

There exists a variety of different other applications, which are currently deployed on the Ethereum network [12] [13]. A popular one is called CryptoKitties. It is a game, where players can collect or breed digital cats [59]. To be more precise, the cats are represented as non-fungible tokens to emulate properties of a digital, scarce collectible. In order to interact with this application, a wallet is necessary.

Different from the ERC-20 token standard explained in Section 3.4, non-fungible tokens (NFT) can be implemented with the ERC-721 standard [60]. It can be used to represent ownership over digital (virtual collectibles) and physical (houses, artwork) assets or assets with negative value. The main implementation difference is that ERC-20 specifies which address possesses which amount of a token, whereas ERC-721 specifies which token belongs to which address. Therefore, ownership of each token has to be tracked separately.

Another popular field of applications is the concept of a decentralized exchange (DEX) [29]. These exchanges are built with smart contracts, which offer the functionality to exchange tokens. An advantage of the decentralized aspect of this application is the fact, that trades are executed atomically and that funds cannot be stolen. Either the trade succeeds or the previous state is restored and the funds are returned to the participants. 0x [14] is an open protocol, which can be used to build a DEX. Similar to Ether transfers in Raiden, Ether have to be wrapped into a token contract (like W-ETH) before they can be traded on a decentralized exchange smart contract. In order to interact with such a smart contract, an Ethereum node is needed or an application like Metamask [61].

[12] *https://www.stateofthedapps.com/rankings/platform/ethereum, Accessed: 2019-05-25*

[13] *https://www.dappradar.com/rankings/protocol/ethereum, Accessed: 2019-05-25*

[14] *https://0x.org, Accessed: 2019-07-01*

8.4 Summary

In this chapter existing wallet applications to transfer Ether and tokens were described. In order to support scalability, Raiden and its functional principle were mentioned. The previous section described the CryptoKitties game with non-fungible tokens and decentralized exchanges. The motivation to mention these applications lies in the fact that currently either Parity or Geth is required, which are cumbersome with respect to their usage of computing resources, in order to interact trustlessly with these Ethereum applications. An alternative is to use an application like Metamask to power all these applications and without the need to synchronize the complete blockchain. This, as already mentioned, has the disadvantage of trusting third-party nodes.

The next chapter presents a proof-of-concept implementation of a simplified light client with the goal to eliminate third-party trust via SPV verification and also to perform orders of magnitude faster and more memory-efficient compared to the light client mode of Geth or Parity. The light client should be usable on a resource-constrained device. Section 11.2 sketches a payment channel with this light client with the additional goal to not have to be online the entire time compared to Raiden.

CHAPTER 9

Proof-of-Concept Implementation: Simplified Light Client

This chapter introduces an implementation of an Ethereum light client, which is based on FlyClient 6. The first section explains the used technology stack. Then the data structures are explained, which are used to store and transmit proofs and Merkle Mountain Range trees efficiently. Furthermore, algorithms to create and verify proofs are described. The client is only able to verify block headers. Other functionalities, such as querying account balances or the state of a smart contract, are outlined in Chapter 11.

9.1 Used technology

To prove the feasibility of the computation on resource-constrained devices a Raspberry Pi 3B+ (RPi) is used [1]. A 16 gigabytes USB stick is used as the storage device. This thesis defines the computing capabilities of the RPi as the lowest common denominator throughout all mentioned computing devices. This means, that if the RPi is able to execute the introduced applications in a reasonable time, then these applications are suitable for the deployment on resource-constrained devices, like IoT and smartphones. Less-powerful devices are not considered. The operating system of the RPi is 'Raspbian' in version April 2019 [2].

To synchronize the existing blocks of the Ethereum blockchain Parity is used in version 2.4.6 [3]. Figure 9.1 shows the setup for the proof-of-concept (PoC) evaluation. The 'Light Server' application is used to fetch all existing block headers from the Parity

[1] *https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/, Accessed: 2019-05-24*

[2] *https://www.raspberrypi.org/downloads/raspbian/, Accessed: 2019-05-24*

[3] *https://www.parity.io/ethereum/, Accessed: 2019-05-24*

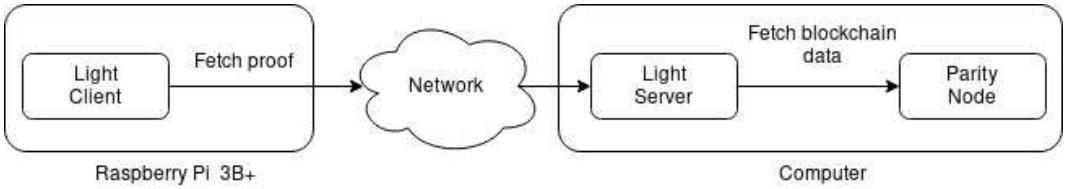

Figure 9.1: Setup for the proof-of-concept evaluation

node, to build the Merkle Mountain Range tree and to create proofs every 128 blocks. A proof generation interval of 128 blocks, which amounts to roughly 32 minutes with the assumption of a block generation interval of 15 seconds on average, seemed to be a reasonable number in order to minimize the required computational work carried out by the server and to maximize the reuse of the same proof but at the same time not to have to use stale proofs. The 'Light Client' application resides on the RPi, it can fetch and validate proofs. For testing purposes the network only consists of a Local Area Network.

The programming language used for the implementation of these two applications is *Rust* in version 1.35 [4], because it is advertised as a fast and memory-efficient language with no runtime and no garbage collector. Another reason for the choice of Rust is to reuse the proof-of-work validation code of Parity in the light client implementation, which is also written in Rust.

9.2 Server-side Data Structures

In this section implementation details about the Merkle Mountain Range tree (MMR) are given. The server has to keep this data structure up to date in order to generate proofs. The MMR is implemented as an array of bytes. This was not the case in the first iteration of the PoC development, but working with an MMR exposed the possibility to implement this data structure as an array of bytes. At the first thought implementing trees as arrays seems unfamiliar, but the properties of an MMR to never delete elements and to only append elements at the end of the structure (which only deletes and adds a logarithmic number of elements at the end) make this possible.

Every element represents a node in the tree and consists of a 32 bytes hash and a 16 bytes aggregated difficulty number (48 bytes in total). The difficulty of a parent node is always the sum of the difficulties of its two child nodes. Figure 9.2 shows the tree representation of the first six leaves of the MMR and the corresponding representation as an array.

Because an MMR with n leaf numbers has always $n-2$ intermediate nodes and 1 root node (see Theorem 1), the total number of nodes is $2*n-1$. This means that the total storage requirement to store an MMR with 7000000 blocks is about 641 megabytes and it is rising linearly in the number of blocks. Therefore, the current server implementation

[4] *https://www.rust-lang.org/, Accessed: 2019-05-24*

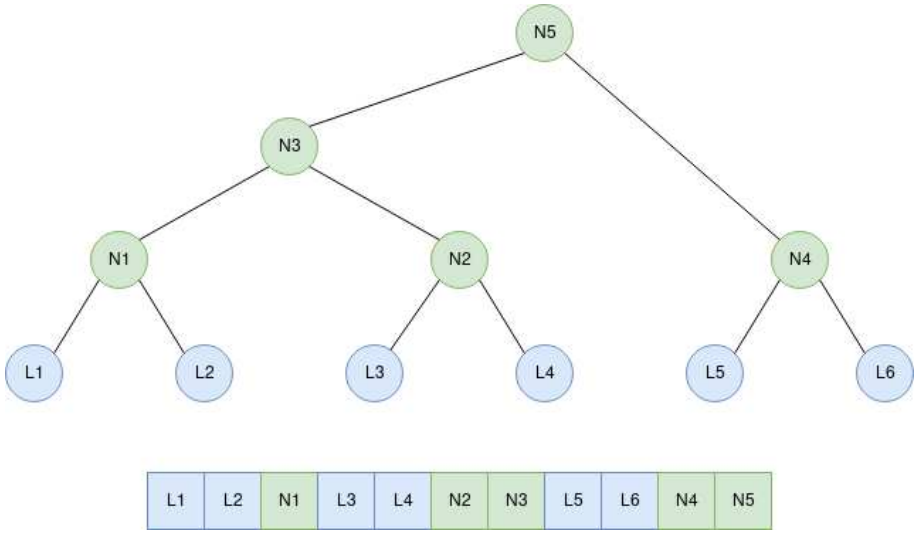

Figure 9.2: MMR represented as tree and as array

can operate in two ways. Either it loads the complete MMR in the RAM or it operates directly on an MMR stored on the disk. Another option, which is not considered in the thesis, is to dynamically build the Merkle Mountain Range tree when it is required.

9.3 Exchange Data Structures

In this section data structures are described, which have to be exchanged between server and client in order to enable an efficient validation of the blockchain. Table 9.1 shows implemented requests with their corresponding responses in Rust syntax. The type of request or response is written as *BlockHeader, LatestBlockNumber* and so on, while the transmitted data are represented by the types written in parentheses. In order to fetch a block header of a specific number, the client transmits the required number as a 64 bit unsigned integer (*u64*) and receives the header (*Header*). If the client wants to query the latest block number, he receives a 64 bit unsigned integer representing the number. The client can ask for a proof by transmitting the desired parameters: λ, c, L (c is represented as percentage points). Obviously, the server only manages proofs of specific parameters, because asking for arbitrary values can lead to a simple but effective denial-of-service attack. If a proof is available, the server sends back an array of block headers (*Vec<Header>* represents the required FlyClient proof blocks and last L blocks), the FlyClient proof (*FProof*), L, the right difficulty (which is basically the sum of the difficulty of the last L blocks), and all blocks issued after the proof creation (see Section 9.5) in form of *Vec<Header>*. The client also has the ability to continue the validation of the blockchain at a specific block number by using the *CP* request. Besides the usual parameters λ, c, L, he also transmits his latest, known blockchain number. Compared to asking for *Proof* the only difference of the server response is the

Request	Response
BlockHeader(u64)	BlockHeader(Header)
LatestBlockNumber	LatestBlockNumber(u64)
Proof(u64,u64,u64)	Proof(Vec<Header>,FProof,u64,u128,Vec<Header>)
CP(u64,u64,u64,u64)	CP(Vec<u64>,Vec<Header>,FProof,u64,u128,Vec<Header>)

Table 9.1: Implemented request-response pairs

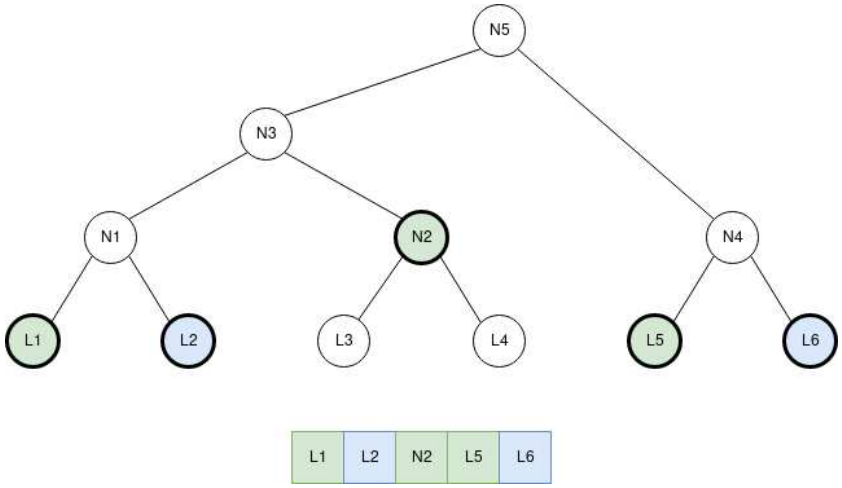

Figure 9.3: MMR proof represented as tree and as array

fact, that all headers, which are not needed anymore, are replaced by an array of 64 bit unsigned integers representing blockchain numbers before the point, where the validation is continued (see Section 10.3).

The FlyClient data structure (*FProof*) is an array of bytes and the contained elements are taken from the MMR. Figure 9.3 shows the tree and array representation of the required information to prove the inclusion of the leaves L2 and L6 in the MMR. Only the additional nodes L1, N2 and L5 are required to prove that the leaves L2 and L6 are indeed in the MMR. Besides the 32 bytes hash and 16 bytes difficulty number, the nodes in the proof have to provide additional information, which are encoded in an additional one byte variable. This variable indicates if the node is a leaf, root or intermediate node and if it is the left or right child of its parent node. This is needed in order to rebuild a partial structure of the Merkle Mountain Range tree for validation purposes.

9.4 Proof Generation

Section 6.5 already described how to turn the interactive FlyClient protocol into a non-interactive one by extracting the randomness of the newest block header. It is important to take into account, that the non-interactive proof requires more block queries. In order

to calculate the correct m, like with formula 6.2 for the interactive case, formula 6.3 and formula 6.4 have to be combined. The result is as follows (the second formula explicitly state m):

$$\left(1 - \frac{1}{\log_c(\frac{L}{n})}\right)^m < \frac{2^{-\lambda}}{c * n} \tag{9.1}$$

$$m > \frac{-\lambda - \log_2(c * n)}{\log_2\left(1 - \frac{1}{\log_c(\frac{L}{n})}\right)} \tag{9.2}$$

Note, that in order to support the handling of variable difficulties, the term $\frac{L}{n}$ is replaced with the aggregated difficulty of the L blocks divided by the sum of the difficulty of all blocks in the current blockchain (see Section 6.4).

Algorithm 9.1 illustrates the approach to generate 'random' numbers, with the required number m and the latest block hash h as inputs. For every required random number (line 2) the concatenation of h (algorithm input) and the random number index are hashed together (line 3). The reason to do this is to generate enough random bits in order to finally build the required random double-precision numbers. The operation in line 3 is secure, because the block header hash is assumed to contain enough randomness. The result of the hash function for a changing input index concatenated with high entropy is again a value with high entropy. As Section 6.5 states, the success rate of the attacker to influence the block header hash is already bounded by λ.

Algorithm 9.1: Calculate random double precision numbers

Input: the latest block hash h, number of required queries m

Output: set R of random numbers

```
1 R ← ∅;
2 for k = 0, k++, k < m do
3     hash = keccak(h, k);
4     byte0 = 63;
5     byte1 = hash[1] | 240;
6     nb = bytes_to_double_precision_nb(byte0, byte1, hash[2], hash[3], hash[4],
        hash[5], hash[6], hash[7]);
7     nb = nb - 1 ;
8     R.push(nb);
9 return R
```

To generate a random double-precision number $n \in [0, 1)$, Saito et al. [62] propose to set the sign bit to 0 and the exponent to 1023. Therefore, the first 12 bits has to be set to 0x3ff. The remaining 52 bits of the double-precision number are filled with random bits. The outcome is a number in the range $[1, 2)$, where finally 1 gets subtracted to get a number $n \in [0, 1)$.

Formula 6.1 can be used as a starting point to get the corresponding aggregated difficulty for a specific random number. Formula 9.3 shows the result of the calculation, where the aggregated probability y is summed up between 0 and a, which are points on the x-axis representing the aggregated weight. This formula is also called cumulative distribution function. Therefore, a has to be between 0 and 1. Formula 9.4 shows the inverse, where the point a can be calculated dependent on the aggregated probability. This formula is also used to sample the random values between 0 and 1 and to get the corresponding aggregated difficulty.

$$y = \int_0^a \frac{1}{(x-1)\ln(\delta)} = \frac{1}{ln(\delta)} * ln(1-a) \tag{9.3}$$

$$a = 1 - e^{y \ln(\delta)} \tag{9.4}$$

In order to get the required block, it has to be searched based on the aggregated difficulty in the blockchain. Algorithm 9.2 was developed for this procedure, which is based on the idea to traverse a tree from left to right in order to sum up all the difficulty values to the point, where the target leaf node represents the desired aggregated difficulty. *curr_tree_number* states the sum of leaf nodes of the subtree, which we are currently inspecting. The loop on line 4 executes as long as we have a subtree of more than 1 leaf node. In the loop we have to distinguish two cases:

- *targetDifficulty >= (aggr_difficulty + left_tree_difficulty).* We know, that our target difficulty exceeds the sum of the aggregated difficulty so far and the highest possible difficulty in the left subtree, therefore we have to branch to the right subtree. The target block number has to be in this specific subtree.
- *targetDifficulty < (aggr_difficulty + left_tree_difficulty).* In this case, our target difficulty is less than the sum of the aggregated difficulty so far and the highest possible difficulty in the left subtree, therefore we branch to the left subtree.

Eventually, the algorithm terminates and we reach the target leaf node. This procedure is important in order to find out the correct position of the node in the blockchain if the

aggregated difficulty is given. After executing the algorithm for each required block, we can proceed with the generation of a FlyClient proof consisting of all these blocks.

Algorithm 9.2: Map aggregated difficulty to block number

Input: Merkle Mountain range tree *mmr*, required block by aggregated target difficulty *targetDifficulty*

Output: Block number n, which corresponds to *targetDifficulty*

```
1  aggr_difficulty = 0;
2  aggr_node_number = 0;
3  curr_tree_number = mmr.leaf_number;
   // traverse tree until leaf node is reached
4  while curr_tree_number > 1 do
5      left_tree_number = calculateLeftTreeNumber(curr_tree_number);
6      node_number =
        calculateNodeNumberSumByLeafNumber(aggr_node_number +
        left_tree_number) - 1;
7      elem = mmr.get_elem_by_number(node_number);
8      left_tree_difficulty = elem.difficulty;
9      if targetDifficulty ≥ (aggr_difficulty + left_tree_difficulty) then
           // branch right
10         aggr_node_number += left_tree_number;
11         left_root_node_number =
            calculateNodeNumberSumByLeafNumber(aggr_node_number) - 1;
12         aggr_difficulty +=
            mmr.get_elem_by_number(left_root_node_number).difficulty;
13         curr_tree_number -= left_tree_number;
14     else
           // branch left
15         curr_tree_number = left_tree_number;
16 return aggr_node_number
```

Algorithm 9.3 shows the procedure to create a proof. All required MMR proof information is included for the given blocks. The algorithm is responsible for calling the recursive defined algorithm 9.4 to traverse the MMR. These procedures are based on algorithm 3 of the FlyClient paper [6], but this is a more generalized version in order to prune duplicate

information. The mentioned paper also gets rid of redundant information, which reduces their proof sizes by 30%, but they have not included the optimized algorithm in their paper.

As an example, Figure 9.3 shows the required information to prove the inclusion of L2 and L6. The algorithm traverses the MMR from left to right and includes every leaf node, where the inclusion has to be proven (blue), and it includes the roots of the subtrees, which do not contain any proof blocks (green). The advantage of this algorithm is that redundant information is never added to the proof independent of the number of required blocks.

Algorithm 9.3: Proof generation

Input: Merkle Mountain range tree *mmr*, required block numbers B

Output: proof P including all necessary MMR proof information

```
1 P ← ∅;
2 P = recursive_proof_generation(mmr.root, B, P);
3 P.push(mmr.root);
4 return P
```

Algorithm 9.4: Recursive Proof generation

Input: current node N, required block numbers B, partial proof P

Output: Partial proof P

```
1  if N is a leaf node then
2      P.push(N);
3      return P;
4  if Contains_any(N.left_node,B) then
5      P = recursive_proof_generation(N.left, B, P);
6  else
7      P.push(N.left)
8  if Contains_any(N.right_node,B) then
9      P = recursive_proof_generation(N.right, B, P);
10 else
11     P.push(N.right)
12 return P
```

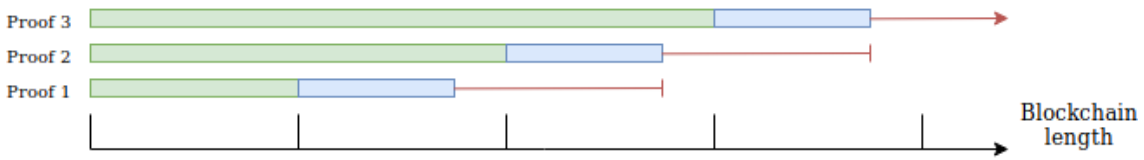

Figure 9.4: Proof generation over time

9.5 Reduce Proof Generation Overhead

The server could generate a new FlyClient proof for every new block. This can result in a performance loss, because the server has to query hundreds of blocks from the internal database for every new block. Another reason to avoid new proofs for every added block is the fact, that a specific proof would not be used by many clients, because with every new block the proof is no longer valid to prove the longest chain. Due to the property, that every client can use the same proof, usage of a specific proof should be maximized.

A possible solution is the limitation of the proof generation procedure to every x blocks, where x is chosen to be 128 in the proof-of-concept implementation. Assuming that a block is generated approximately every 15 seconds, a new proof would be generated every 32 minutes. In order to let all servers generate their proofs at the same block number, proofs are generated for the block number n if and only if the following statement is true: $n \bmod 128 = 0$. Figure 9.4 visualizes this approach, where the markers on the x-axis are block numbers divisible by 128. The FlyClient proofs span all the blocks held in green, whereas the blue blocks are the L blocks, which have to be checked with probability 1. The red line shows the duration of blocks, where the proof is the most recent one. Therefore, the server sends the current proof (FlyClient proof together with the following L blocks) and additionally all blocks, which are generated after the L blocks (blocks, which fall into the red line), in order to allow clients to validate the blockchain.

9.6 Proof Validation

In order to validate a proof, the client has to calculate all required block numbers. To do this, the client needs information regarding the latest block, the sum of the difficulty of the last L blocks and the sum of the difficulty of the corresponding MMR (difficulty stored in the root node). Therefore, the server also has to transmit these data combined with the FlyClient proof. The calculation of the required block numbers is the same as mentioned in Section 9.4 for the generation of the proof. With other words, the client wants to countercheck, if the server has indeed created the proof with the correct 'random' blocks. Algorithm 9.5 can be used to verify the MMR proof.

Because the proof generation algorithm is already a generalized version, which supports pruning duplicate information, this algorithm supports the validation of the correct inclusion of all required blocks in one execution, unlike algorithm 4 of the FlyClient paper. They have not added the more generalized version.

Algorithm 9.5: Proof validation

Input: Proof P, required block numbers B

Output: If P is valid or not

```
1  Nodes ← ∅;
2  root = P.pop();
3  Reverse_elements(P);
4  while P ≠ ∅ do
5      elem = P.pop();
6      if elem is a leaf node then
7          if Nodes ≠ ∅ then
8              hash, difficulty = Calculate_previous_mmr(Nodes);
9              if hash ≠ elem.mmr or !Check_aggregated_difficulty(elem, difficulty)
                then
10                 return false
11         if elem is left node then
12             Nodes.push(Calculate_parent(elem, P.pop()));
13         else
14             Nodes.push(Calculate_parent(Nodes.pop(), elem));
15     else
16         if elem is right node then
17             Nodes.push(Calculate_parent(Nodes.pop(), elem));
18         else
19             Nodes.push(elem);
20     while Nodes.len() > 1 and Nodes[Nodes.len()-2] is left sibling of
        Nodes[Nodes.len()-1] do
21         right_elem = Nodes.pop();
22         left_elem = Nodes.pop();
23         Nodes.push(Calculate_parent(left_elem, right_elem));
24 if Nodes[0].hash ≠ root.hash or Nodes[0].difficulty ≠ root.difficulty then
25     return false
26 else
27     return true
```

Note that the *Nodes* data structure stores nodes, which have to be processed at a later point again. The loop iterates over all elements in the FlyClient proof byte array, which represents elements in the Merkle Mountain Range tree from left to right. The root node is already removed from the array. Therefore, the algorithm only has to deal with intermediate nodes and leaf nodes. The procedures, which have to be executed in every iteration of the loop in the algorithm, are described below in more detail:

1. **Elem is a leaf node (line 6).** The reason for an empty *Nodes* variable is the fact, that the current element is the Genesis block. Only for this block the check of the correct MMR root hash in the block header is omitted together with the difficulty check. Otherwise, the root node of the MMR stored in the leaf node can be restored with the knowledge of elements in *Nodes*. A similar procedure to check the validity is used as stated in *Handling Variable Difficulty* [6]: the calculated MMR root node has to be the same as stated in the block header and the relation of the aggregated difficulty to the total difficulty has to be in a specific range.

   If the leaf node is the left child of a parent, then we know that the next element X has to be the right child, which is also a leaf node. Therefore, we take X out of the array and calculate the hash and difficulty of their common parent node. The resulting parent element is pushed into *Nodes*.

   Otherwise, the leaf node is the right child of a parent and the last element added to *Nodes* is the corresponding left child. Also in this case the parent node can be calculated and pushed into *Nodes*.

2. **Elem is an intermediate node (line 15).** In this case we have to distinguish, if the element is a left or right child of a node in the MMR. If it is the left child, then we push it into *Nodes*. Otherwise, we know that the last element pushed into *Nodes* has to be the corresponding left child, which shares a common parent with the current element. We can use these two elements to calculate the parent node and push it into *Nodes*.

3. **Elem is a leaf or intermediate node (line 20).** In every iteration it has to be checked, if the elements in *Nodes* can be further processed. If the last element is the sibling of the second last element, they can be used in order to calculate their common parent node. The resulting parent node is pushed into *Nodes* afterwards. Then, the check can be repeated as long as *Nodes* has enough elements and the last two elements are siblings.

At the end of the algorithm it has to be the case that *Nodes* only consists of one element, the calculated root node. This node has to coincide with the root node of line 2. If the algorithm terminates with *true*, the proof is valid.

9.7 Proof-of-Work Validation

Besides checking the FlyClient proof, also the proof-of-work of the required blocks has to be validated. In order to not re-implement this procedure, the Parity code is used [5]. Because the code is written for the case to validate the blocks sequentially, the proof-of-concept implementation uses the procedure described in Algorithm 9.6 to make use of all four cores of the Raspberry Pi 3B+. The first loop (line 2) iterates over all blocks and divides them according to their epoch numbers. The next loop (line 5) iterates over all required epoch intervals, where the proof-of-work validation cache (line 6) has to be generated. This loop is implemented in a way, such that the work is distributed evenly on every CPU core and if one core has finished its work earlier it will 'steal' more work from other cores. Inside the next loop (line 7) all blocks from the same epoch are checked for valid proof-of-work (line 8). The proof-of-work is valid for all blocks in B if and only if the algorithm terminates and outputs 'true'.

Algorithm 9.6: Proof-of-work validation

Input: Required blocks B

```
1  P ← ∅;
2  foreach b ∈ B do
3  |   epoch = b.number / 300000;
4  |   P[epoch].push(b);
5  foreach E ∈ P do
6  |   ethashManager = ParityEthash();
7  |   foreach e ∈ E do
8  |   |   if ethashManager.validate(e) == false then
9  |   |   |   return false
10 return true
```

After successfully verifying the proof and the proof-of-work of the required blocks, the client can be assured, that the root hash of the proof reflects a valid state of the blockchain within the limits of the defined threat model. If the client obtains different proofs from two connected nodes, he can find the one with a higher aggregated difficulty. This proof represents the valid state, because it is assumed that the client is connected to at least one honest node and the longest blockchain is always the valid one (see Chapter 7).

With the introduced PoC, the client can make use of the FlyClient's approach in order to validate the latest valid block header in a decentralized way. The next chapter uses this implementation for measuring its resource consumption in order to check the feasibility of a deployment on a resource-constrained device.

[5] *https://github.com/paritytech/parity-ethereum/, Accessed: 2019-11-26*

CHAPTER 10

Proof-of-Concept Evaluation

In this chapter the implemented proof-of-concept gets evaluated. The first section shows proof sizes calculated for different block numbers. The next sections list results, which are gathered from proof validations on a Raspberry Pi 3B+ model. The implementation is also able to continue to synchronize the blockchain at an arbitrary block number. This case is also described and measured in terms of used computing resources. The last section gives a conclusion about the results of this chapter.

10.1 Proof Sizes

Figure 10.1 shows the number of required block queries and resulting proof sizes dependent on the length of the blockchain. The parameters are the same as in Figure 5 in the FlyClient paper ($\lambda = 50$, $c = 0.5$). In the paper the number of blocks to check with probability 1 (parameter L) is not fixed in order to try to minimize the proof size. The thesis does not make use of this optimization. In this diagram, L is set to a constant value of 100. The values in this graph are higher, because this proof is built with the security of a non-interactive proof in mind.

To calculate the proof sizes, all block headers are set to 508 bytes, hash outputs to 32 bytes. The size of the difficulty variable in the MMR is set to 16 bytes, which is different from the 8 bytes used in the FlyClient paper. The reason is, that the proof-of-concept encodes the difficulty as an unsigned integer and 8 bytes would already result in an overflow for the aggregated difficulty of the first 7000000 blocks in the Ethereum blockchain. A bigger variable size together with the fact, that L is hard-coded, results in an additional increase in the proof size.

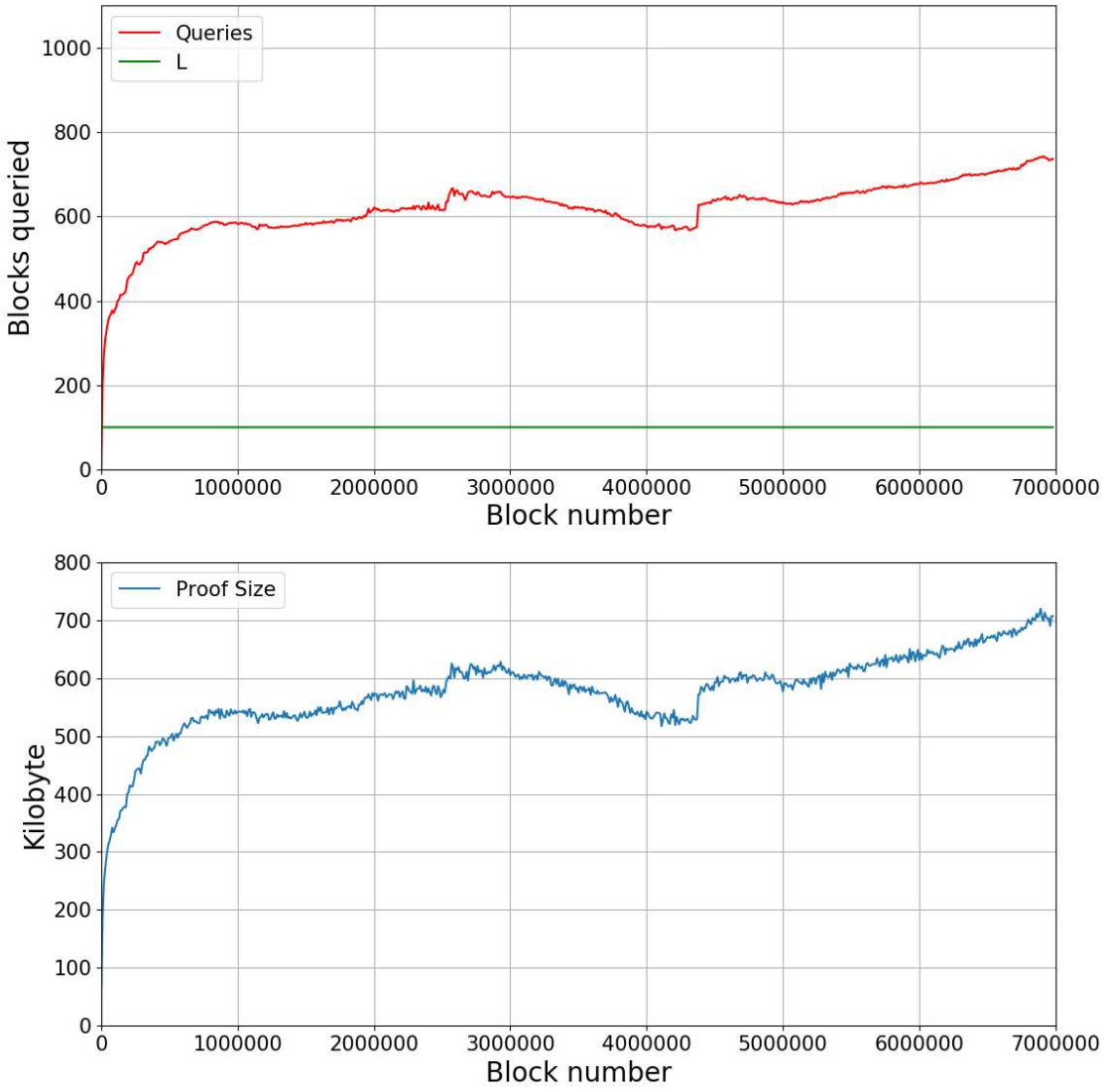

Figure 10.1: Proof sizes and number of required blocks dependent on the block number

10.2 Proof Validation Times

Figure 10.2 shows the time necessary to validate a FlyClient proof and the proof-of-work of all required blocks on a Raspberry Pi 3B+ model. c is set to 0.5, $\lambda = 50$ and L is 100. The FlyClient proof can be validated in less than 1 second for the measured blockchain sizes. Therefore, most of the time is spent on validating the proof-of-work of the Ethereum blocks. As already mentioned in Section 3.2, Ethash is quite resource demanding. Figure 10.2 additionally shows the resulting proof sizes and the required numbers of different epochs, for which the cache has to be generated. The correlation between complete validation time and required epoch number can be graphically observed.

10.3 Reducing Synchronization Time

The previous measurements are conducted with the assumption, that the client has no prior knowledge of any block after the Genesis block. If the client has already synchronized before and wants to update his knowledge, the FlyClient paper also deals with this case and describes a solution in order to not have to validate the complete chain again: if the client already knows that a specific block X is valid, and the client has received a new proof with a Merkle proof including X, then he only has to validate the subchain after X.

The proof-of-concept implements so-called checkpoint blocks, from which the client can continue to synchronize the chain. These checkpoints represent blocks located at the start of epoch intervals in Ethereum and the PoC includes 10 of them in a proof. The first checkpoint block is located on the start of the same epoch interval as the last block of an MMR proof. The second block is located on the start of the previous interval and so on. If a block generation time of 15 seconds is assumed, then 30000 blocks (length of an epoch interval) span a time of about five days. This means, that the checkpoint blocks span a time of 45 days plus the amount of time from the first checkpoint to the current block, which is less or equal than 5 days. The result of this approach is, that a light client can synchronize in a faster way, if its last known block is not older than about 45 days. The newer the block, the faster the synchronization time, because the client could use a newer checkpoint to skip older blocks.

To be more precise, Figure 10.3 shows the interaction between client and server, which is described below:

1. The server generates a proof and includes 10 checkpoint blocks
2. The client has no prior knowledge, therefore he fetches a proof from the server and validates it. The client stores the newest included checkpoint block.
3. Some time went by where the client stays offline. Meanwhile, the server has to generate new proofs, because new blocks are being generated.

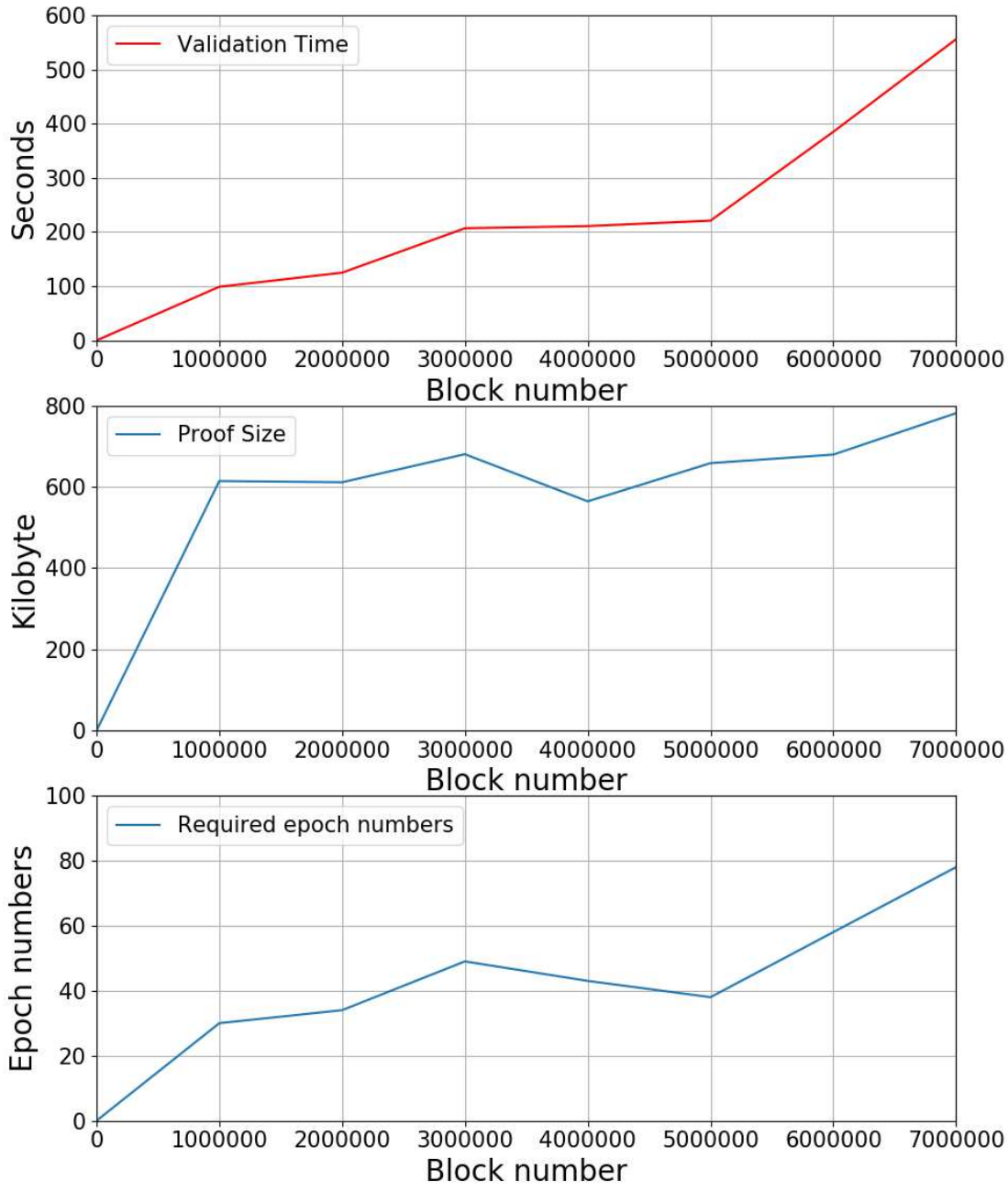

Figure 10.2: Proof validation times, proof sizes, and associated epoch numbers dependent on the block number

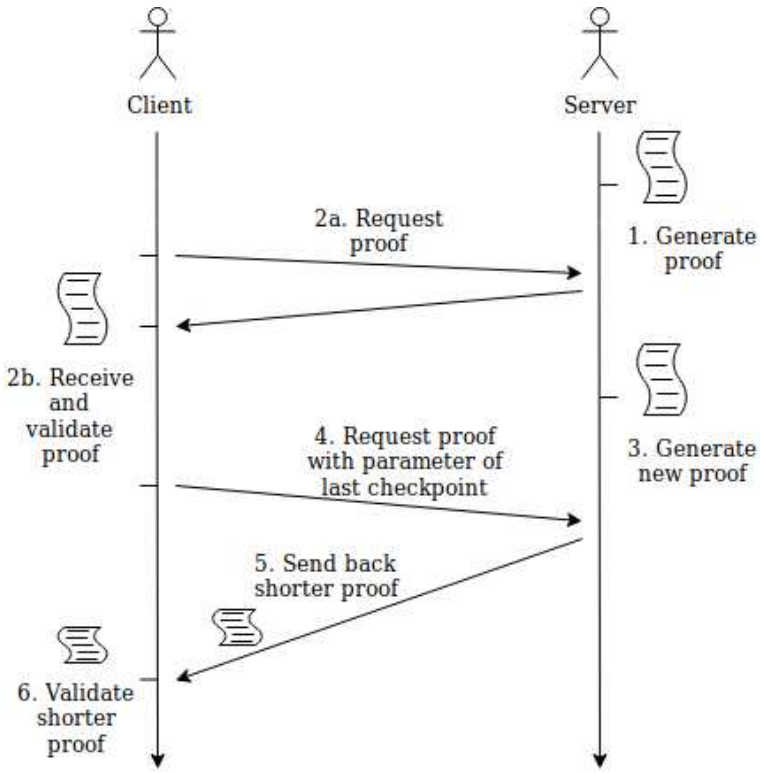

Figure 10.3: Proof validation with and without prior knowledge of an older proof

4. The client goes online and requests a proof of the server. He also includes the last known checkpoint block number into the request.

5. The server sends back the newest proof, but the complete block headers are only included for blocks after the specified checkpoint block.

6. The client verifies the complete proof, but only checks the proof-of-work for new blocks after the provided checkpoint. If the checkpoint was already too old, the client has to validate the proof-of-work of all blocks.

Theoretically, the server could additionally generate a proof, which only includes the subchain between a checkpoint block and the current last block. This would result in a slightly smaller proof. A reason, why the proof-of-concept does not implement this approach, is the fact, that the implementation would be more complex without a real benefit. The last figures have already shown, that the validation time for proofs are negligible compared to the proof-of-work validation. The first and second plot in Figure 10.4 shows the required validation times and proof sizes dependent on the block number, where the synchronization is continued. Again, c is set to 0.5, $\lambda = 50$ and L is 100. Blocks before any checkpoint need the usual validation time. If a checkpoint block

can be used, the validation time drops significantly and with every newer checkpoint the required time drops more and more. Also, the proof size decreases, because headers of blocks occurring before the given checkpoint are no longer transmitted.

The third plot in Figure 10.4 can help to explain, why the proof size does not drop in the same intensity compared to the validation time. The required number of blocks, where a proof-of-work validation is necessary, changes with nearly the same intensity as the proof size in the second plot in Figure 10.4. The only reason for the decrease of the proof size is the fact, that the server does not transmit the full block headers occurring before the given checkpoint. Because the block headers are nearly constant size, the lines are changing with the same relation. The significant drop in validation time can be explained with the necessary generation of the cache files for each required epoch in order to allow a proof-of-work verification for the corresponding blocks (see Section 3.2). The conclusion is, that in order to minimize the validation time, the primary goal is not to minimize the required blocks, for which the proof-of-work has to be checked, but to minimize the number of epochs, where these blocks are located in.

10.4 Trade-off between Proof Size and Proof Validation Time

Because of the conclusion of the previous section, the following question can be asked: how does the choice of L influence the proof size and validation time? Figure 10.5 shows the correlation of these factors with L, which ranges from 100 to 1000. The blockchain has an assumed length of 7000000 blocks. The client has no prior knowledge of any block, therefore he is forced to validate the complete chain. It can be observed, that the proof size increases in a nearly monotonic way with the increase of L, whereas the validation time rises and falls, but stays between 400 and 600 seconds. The motivation to use a bigger L value was the thought, that the more blocks in the end are checked, the fewer blocks in the middle of the chain has to be checked and the less epoch intervals have to be calculated. To conclude the evaluation, a small L value is preferred, which was already used in the previous measurements ($L = 100$). The optimal L value, which minimizes the proof size in the FlyClient paper, is also about 100 and does not change significantly with the blockchain length.

10.5 Summary

The measurements in this chapter have shown, that proof sizes are lower than 1 megabyte and the validation time is below 10 minutes in order to be convinced about the latest valid block header of the Ethereum blockchain with a length of 7000000. Bigger validation speedups can be reached if the last synchronization day lies within the last 45 days. In such a case only a maximum of 100 seconds are required in order to validate a proof. A proof size under 1 megabyte is orders of magnitude lower compared to the enormous amount of data required for a traditional SPV verification in Ethereum 5.4. It can be

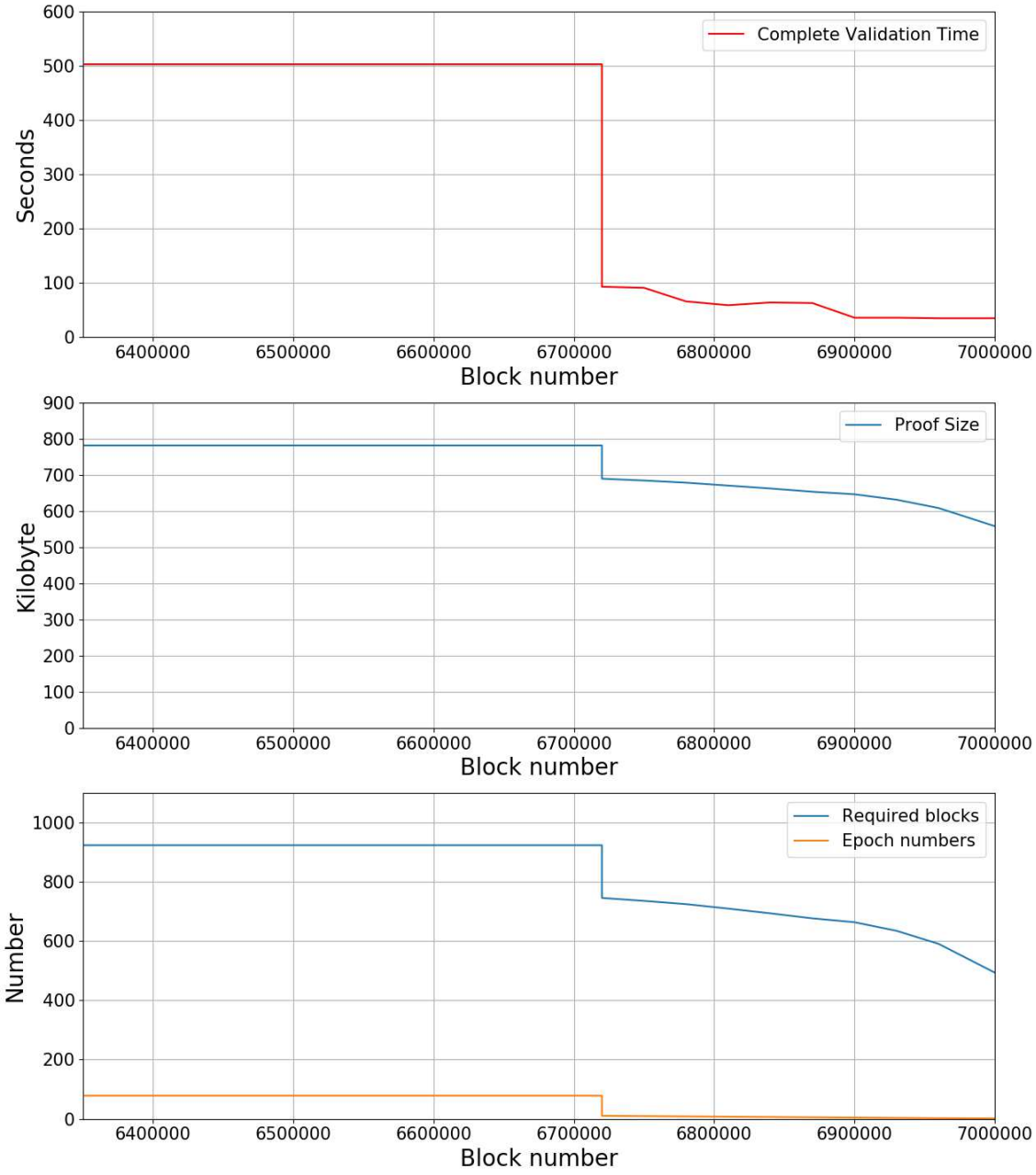

Figure 10.4: Proof validation times, proof sizes and required number of epochs and blocks dependent on the block number, where the synchronization continues

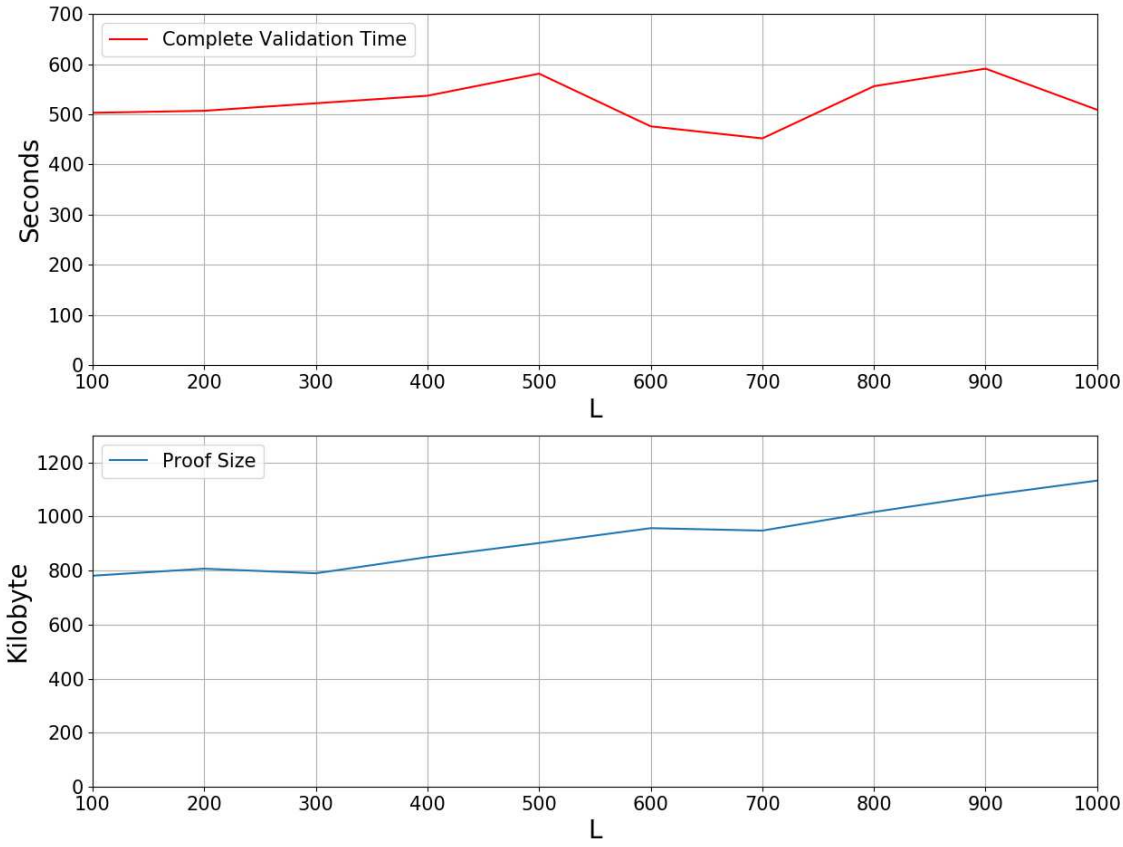

Figure 10.5: Trade-off between proof validation time and proof size dependent on L

concluded, that the implemented light client, which is based on FlyClient, can be deployed on resource-constrained devices. The next chapter outlines concepts, how applications can be powered by the introduced light client in order to efficiently run in a decentralized way.

CHAPTER 11

Empowering Light Clients

In the previous chapter it was shown that the proof-of-concept implementation is able to efficiently fetch and validate FlyClient proofs in order to determine the last, valid block header. The next question is the following: how can a hypothetical light client make use of this header in order to empower applications mentioned in Chapter 8. An important goal is to remove the need of a third party. The first section deals with wallet applications, such that users of resource-constrained devices are able to transfer money via blockchain transactions. The second section incorporates the payment channel technique into the hypothetical light client in order to allow users to send and receive money via channels. This procedure allows the blockchain to scale up the transaction throughput. The last section gives a summary, which recapitulates the possible features of a light client, which is built on top of the FlyClient's approach.

11.1 Wallets

Unlike Bitcoin, an Ethereum block header also contains the state root [4]. This means, that in order to check the balance of an account the following steps are sufficient: (1) fetch and validate newest FlyClient proof, (2) check the Merkle proof from the specific account in the state tree up to the state root hash stored in the last valid block header. A wallet application should also have the possibility to list all incoming and outgoing transactions. Algorithm 11.1 sketches a possible procedure. Note that Bünz et al. [6] has already outlined the procedure to check the inclusion of any transaction in the blockchain.

There exists different options how the client can be automatically notified if a new incoming payment arrives:

- The payer sends the transaction with the corresponding Merkle proof and block header directly to the client. The client has to validate this Merkle proof and a

Algorithm 11.1: List transactions

1 The client asks full nodes to send all transactions, where a specified address is the sender or receiver of a payment. Let X denote the set of block headers, where these transactions are included. The client has to retrieve X and Merkle proofs of the transactions.

2 The client validates the Merkle proofs and also a new FlyClient proof. Let y denote the root hash of the FlyClient proof.

3 In order to check if every block header of X is in the blockchain with an MMR root hash of y, the client requests MMR proofs of the headers.

4 The client validates the MMR proofs. If the trust model is satisfied, then no full node can fool the client with a bogus transaction. An evil node can only ignore the request to send back all transactions affecting the specified address. Because the client has to reveal the address of interest for gathering corresponding transactions, privacy concerns have to be considered (see Section 12.7).

new FlyClient proof in order to prove the membership of the given block header in the valid chain.

- The client continuously asks full nodes for new block headers, which are validated and added to his blockchain. In addition, he asks for transactions regarding his address, which can then be validated with the help of Merkle proofs.
- The client continuously asks full nodes for complete blocks, which are validated and added to his blockchain. New transactions are recognized by validating all transactions included in a block.

A simpler alternative would be the creation of a new account for every incoming payment. Because the account would only be used once, the client can easily check the payment by fetching and validating the latest state of the account. If the balance is not zero anymore and the account is only known to the client and the payer, the client can immediately connect the payment to the payer.

In order to send a payment transaction and check the result afterwards, the procedure outlined in Algorithm 11.2 can be used.

The approaches in this section does not only work for Ether, but also for ERC-20 tokens. The only difference is, that not the account gets queried but the state of the corresponding token contract. Also, non-fungible tokens, for example CryptoKitties, can be managed with such a light client wallet (see Section 8.3). Therefore, this approach enables resource-constrained devices, like smartphones, to be used for such applications without the need of a trusted third party.

Algorithm 11.2: Send payment

1 The client retrieves the newest state of his account by checking the Merkle proof of his account against the last valid block header. This is needed because the client has to use the correct nonce in order to create a transaction [4]. He can also check the current balance in order to avoid the creation of a transaction with a value, which exceeds his balance. He also has to check the latest block in order to find out an appropriate gas price.

2 The client sends the new transaction to the Ethereum network.

3 The client proceeds with any of the procedures mentioned earlier to be automatically notified when the transaction has been processed.

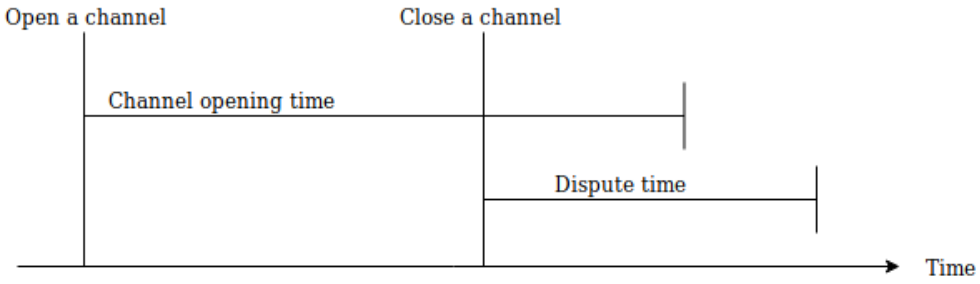

Figure 11.1: Payment Channel over Time

11.2 Payment Channels

In this section a simplified payment channel is described. As already mentioned in Chapter 4, the only interactions with the blockchain are: (1) to open and (2) to close channels and (3) to resolve possible disputes. Figure 11.1 shows the channel handling over time. Imagine two individuals, named Alice and Bob. They want to transfer funds between each other with the help of a channel. At first both have to lock funds in a channel contract ('Open a channel'). The lock is active for a specified time ('Channel opening time'). The next step is to directly exchange messages (without the involvement of the blockchain), which state the new allocation of funds and are signed by both parties. To close the channel before the channel is closed automatically, one of them has to transmit such a message to the blockchain ('Close a channel'). The signatures are required in order to convince the smart contract that a distribution of the funds is desired by both parties. If Alice or Bob sends an old message to the smart contract, the other person has a specific timeframe ('Dispute time') to send a newer message in order to allow the contract to settle the dispute. The reason why this works is, that the smart contract can identify newer messages because of the sequence number in the messages.

The following properties are responsible for the acceptance of a signed message as a valid

payment: (1) the payee can be assured, that the stated amount in the signed message is available as locked funds, and (2) the payee is the only one who has access to the locked funds while the channel is open. In order to not lose funds, the individuals have to make sure to close the channel in a reasonable time before the channel expires. For example, if Alice closes the channel with an old message, Bob has to make sure to convince the smart contract of a newer message before the dispute time is over. The time to get a transaction processed in the Ethereum network has to be considered. Especially in times of a network congestion, which slows down transaction processing, participants have to make sure to send transactions earlier to the network to not miss a deadline like the automatic channel closing or the end of the dispute time.

In order to build a payment channel application with the help of the proof-of-concept implementation, Algorithm 11.3 sketches a possible process flow for a bidirectional channel (see *Duplex payment*, Section 4.1). Alice wants to exchange funds with Bob. It is assumed, that a smart contract implementation already exists, where a channel can be opened and closed and the contract provides functions to enable the dispute handling. The channel-closing procedure can be implemented with the signature-checking functionality of the precompiled ECDSA contract [4]. Furthermore, the end of the dispute time has to be at a later point compared to the automatic channel closing point regardless of the time the channel gets closed.

Algorithm 11.3: Open and close a channel

1. Alice sends a transaction to the channel contract with funds, which get locked for a specific time. She continuously fetches new FlyClient and Merkle proofs to check the state of the contract, because she wants to be assured that her opening-channel-transaction get processed.

2. Bob can check, if Alice has already locked funds by querying the state of the smart contract. The gathered information can be validated with FlyClient and Merkle proofs. If he detects the locked funds, then he will also lock his funds.

3. If both parties have locked funds successfully, they can send signed messages containing new balances directly to each other.

4. During the time, where the channel is open, they can retrieve the newest state of the payment channel smart contract by validating new FlyClient and Merkle proofs. This has to be done in order to detect channel closings. In theory, the participant only has to go online at the time when he wants to close the channel before the channel closes automatically. There are two cases, which can happen: (1) Either the channel was not closed before, then the participant transmits the last signed bid, or (2) the channel was already closed, then the participant is able to call the dispute function on the smart contract if the counterparty transmitted an old (invalid) message.

In contrast to Bitcoin, Ethereum has the advantage that block headers contain the root hash of the current state. In order to identify channel closings, the client only has to check the latest state trie. Unlike Lightning [9], it is not necessary to continuously watch the blockchain for closing transactions if it is assumed that the client can go online once in a while. Therefore, the concept of watchtowers is not needed in this case and the light client can operate without trusting a third party.

11.3 Summary

In this chapter, procedures were outlined how Ethereum applications can be built on top of the FlyClient's approach. At first a wallet application was introduced, which enables light clients to send and receive payments. Then, the payment channel technique was considered and a possible integration was discussed. The features of such a light client are as follows:

- Due to the FlyClient's approach, a resource-constrained device can find out the latest valid block header of the Ethereum chain.
- A wallet application can be developed, which works similar to SPV verification of transactions with its usage of Merkle proofs. The inclusion of the state tree hash, transaction hash and receipt hash in the block header allows asking full nodes for a lot of different information in a way, such that the information can be verified without requiring trust. Examples are the balance or nonce value in order to create valid transactions. Such information can be easily checked for correctness by checking the corresponding Merkle proofs.
- The light client also enables the interaction with payment channels. Because of the assumption that the light client can go online once in a while, there is no need for a complex watchtower setup, where an incentive structure has to be developed in order to not miss a rogue transaction on the blockchain. Because payment channels exist as smart contracts, the state tree hash together with a Merkle proof to the state of such contracts allows to easily verify the current state of the channel.
- As already mentioned, the existence of hash values to different data structures in the block header allows to easily verify information gathered from full nodes. Therefore, the previous mentioned applications are by far not the only possible applications for light clients. With the usage of smart contracts, there exists a great amount of different applications, which can be used on light clients.
- The most important feature of the applications introduced in this chapter is the absence of a trusted third party.

The question arises how adversaries can break such light clients. The next chapter deals with some noteworthy attacks and their consequences.

CHAPTER 12

Attacks

This chapter gives an overview of security threats and their consequences affecting the proof-of-concept implementation or potential applications introduced in Chapter 11. Note that these attacks are only possible if some assumptions of the problem statement and the threat model are relaxed (see Chapter 7). The first and second sections deal with hash and signature functions and how the Ethereum blockchain can break if some assumptions of these functions are violated. Afterwards, an attacker is described with the majority of the mining power and how he can exploit this situation. The next two sections describe cases, where the client cannot connect to other nodes anymore, because of an adversary or of a network defect. Implementation errors in applications are common, therefore the following section deals with consequences of these. Section 12.7 outlines potential privacy intrusions conducted by full nodes, which can learn about usage patterns of light clients. The last section gives a summary about the attacks.

12.1 Breaking Hash Functions

If the underlying cryptographic primitives of Ethereum would be broken, such as the deployed hash functions or the elliptic curve cryptography, then this would be a worst-case scenario with unforeseeable consequences and it is debatable if and how this situation can be mitigated. Nevertheless, some situations are described, which the attacker could exploit.

Breaking the hash function H can result in violating different defined properties of Section 2.1:

- If the *preimage-resistance* cannot be upheld anymore, the attacker is able to find x for an arbitrary given y as input, such that $H(x) = y$. The attacker could use this power in order to mine blocks. If x is the RLP representation of the block

header and y is the mining target, the attacker can calculate the required block header and set the nonce accordingly. The question arises, if the attacker is only able to calculate random block headers, or if he can calculate block headers, where he can set some bits representing important fields and leave the nonce and other unimportant bits empty in order to generate valid headers. If the attacker can generate valid block headers faster than the honest mining power can mine blocks, then it would be similar to a 51% attack (see Section 12.3 for possible attacks).

If the digest of H is used to generate signatures, then the attacker can forge signatures [18]. The attacker has to find a message m, where $H(m) = e$. He can easily calculate valid values of e, which only needs the curve parameters and the public key of the counterpart as inputs. The result is a valid signature for the message m (see Section 12.2 for possible attacks).

- If the attacker is able to break *second-preimage resistance*, then he is able to find x' for a given x, such that $H(x') = H(x)$. As in the previous example, he can mine blocks, if the requirement to manually set important bits of the header is satisfied. The only restriction is, that he cannot choose the hash output arbitrarily, but only use already generated hashes with known input values.

If the attacker can manually set block header bits and calculate the remaining bits in order to comply with a specific digest, then he can replace specific blocks from the blockchain. Let $B_1 \leftarrow B_2 \leftarrow B_3$ be a chain with three blocks and the attacker wants to replace block B_2, then he generates a partial header $B_{2'}$, which points to B_1 and generates the remaining bits in a way such that the resulting hash is the same as stored in the field *previous block hash* in B_3. The resulting chain $B_1 \leftarrow B_{2'} \leftarrow B_3$ is valid.

12.2 Breaking ECDSA

If an attacker is able to forge signatures, he can create valid transactions. Therefore, funds from arbitrary addresses can be stolen and payment channels can be closed arbitrarily by forging signed messages. Stewart et al. [63] argue, that quantum computers are a potential risk for ECDSA in the future and countermeasures have to be initiated in order to update vulnerable cryptography of blockchain projects to post-quantum cryptography.

12.3 51% Attack

The underlying threat model (see Chapter 7), on which the proof-of-concept implementation is based on, defines the hashrate of a possible attacker as a fraction of the total honest hashing power, $c \in [0, 1)$. This means that an attacker can never reach 50% or more of the total hashing power.

If the attacker has more power compared to the total honest hashing power, then the chain of the attacker will always get longer at some time. Therefore, he can suppress

arbitrary transactions, because it is guaranteed that only his blocks will end up in the longest, valid chain. He can also replace a number of last blocks of the current chain by starting to mine on top of an older block. Because his chain can grow faster compared to the chain of the total honest mining power, his chain will eventually be longer. SPV applications do not verify transactions, therefore the attacker is able to convince such applications of the validity of an invalid chain. SPV can only work with the assumption, that the longest chain is valid.

12.4 Eclipse Attacks

Marcus et al [64] define eclipse attacks as an attack, where the attacker completely controls the victim's access to information about the blockchain. They present an attack that is possible to be executed with only two machines, both of them having a unique IP address. Because Ethereum nodes are only identified by an ECDSA public key, they are able to generate a set of nodes, which then eclipse the victim.

If the proof-of-work of the longest chain is eclipsed from the rest of the network, then the attacker can convince the victim into believing in the validity of a shorter blockchain, because the victim does not know of the existence of a longer chain. The attacker has to generate a chain with valid proof-of-work, from which a proof can be created afterwards. Other attacks include the denial to send back existing Merkle proofs, repression of transactions and the inclusion of invalid transactions, because the light client cannot verify their correctness. A payment channel network cannot run reliable in such a setting. Imagine that an eclipsed entity A has a channel with the attacker or any non-eclipsed entity B. If A cannot observe, that B already closed the channel on the valid longest blockchain with an old signed message, then he does not execute the dispute function on the smart contract and can therefore lose funds.

A mitigation to such an attack could be a check of the sudden difficulty drop in the proof-of-work, because the attacker has to continuously generate new valid blocks in order to not raise suspicion and it is assumed that the attacker only controls a minority of the total hashing power. However, this procedure would have to include heuristics of the difficulty-changing patterns of the past and a detailed understanding of the inner workings of the Ethereum difficulty adjustment algorithm would be necessary. It is not clear that a distinction between a valid difficulty drop and a malicious one can be made in every case.

12.5 No Internet Connection

If the attacker only disconnects the client from the Internet or the client loses the connection because of a technical defect, then the client can easily notice that by observing the absence of any arriving packet at his network interface. This is not a major issue for wallet applications, because funds cannot be stolen. If the client has an open

payment channel at that time, then he has to make sure to go online before the channel automatically closes or when the dispute time ends. Otherwise, the client can lose funds.

12.6 Implementation Errors

Errors can happen on different levels. Atzei et al. [65] provide a taxonomy of vulnerabilities on the Ethereum blockchain, which is divided into Solidity-, EVM- and blockchain-bugs. The most notable attack was the DAO attack, which led to a hard-fork of the Ethereum chain. Because of the severity of such bugs, smart contracts have to be designed carefully in order to make them not exploitable. For this reason formal methods are researched in order to prevent them [66] [67] [68] [69].

Besides payment channel applications, also ordinary wallet applications can make use of smart contracts, which have to be considered while evaluating security risks. A bug in the Parity multisignature wallet contracts resulted in a loss of $ 280 million worth of cryptocurrencies [70].

Implementation errors can also happen in the light client implementation or in the server application, which generates the proof. If not all proof checks, which are stated as necessary in the FlyClient paper [6], are properly executed, then an attacker could exploit this circumstance.

12.7 Privacy Intrusion

An advantage of the FlyClient proof is, that it is not specific to a light client. Therefore, one proof is enough to serve all such clients. Beyond the validation of the latest valid block header, the client has to ask full nodes in order to retrieve further blockchain data. For example, to check the balance for an address, the client has to ask a full node to send back a Merkle proof for the specified address. The full node can infer from the request that the client is probably the owner of the address, because signing transactions can only be done if the corresponding private key is known. If the attacker controls one or more nodes over a longer time, he can probably create movement patterns and know more about the client than the client is willing to tolerate. This privacy intrusion has to be tackled in order for the client to disappear in a sufficiently large anonymity set.

In order to obscure network traces, the client can use Tor [1]. Careful implementation considerations have to be made, because this approach allows to carry out different attacks [71]. There also exists different techniques to strengthen privacy for SPV clients [72] [73].

12.8 Summary

As this chapter suggests, adversaries have a broad spectrum of different attack types. They can look for implementation bugs in the PoC or in the Ethereum nodes Geth and

[1] *https://www.torproject.org/, Accessed: 2019-11-27*

Parity. Bugs can exist in the programming language Solidity, in the EVM specification, or in smart contracts. Protocol specifications have to be hardened to deal with all sort of attacks. Network disconnections also have to be taken into account in order to construct robust applications. A number of assumptions are made in Chapter 7, which have to be carefully evaluated in terms of practicability in order to permit a deployment of an application, which is based on these assumptions.

Other blockchain projects make use of more complex cryptography compared to the introduced concepts. Therefore, the next chapter discusses some implications of this fact. It also mentions a possible proof-of-work change in Ethereum and further fields of applications for the introduced light client.

CHAPTER 13

Discussion and Future Work

The first section deals with the deployed cryptography of Ethereum and the introduced light client. Because the used hash and signature functions are well-established compared to some exotic, complex cryptographic primitives in other cryptocurrencies, the confidence in Ethereum regarding security is presumably higher. Because the Ethereum blockchain will likely not use the Ethash proof-of-work algorithm in the future, the following section gives some notes about possible proof-of-work changes and how these affect the deployment of the light client. The last section suggests some possible fields of applications, which can make use of procedures outlined in Chapter 11. Specifically, three applications in the IoT (Internet of Things) sector are considered in this section: (1) the transfer of sensor messages, whose integrity is of utmost importance, in a smart city context, (2) access control management, and (3) peer-to-peer energy trading.

13.1 Absence of Complex Cryptography

The proof-of-concept implementation, which is based on the FlyClient's approach, and the discussed applications, which can be built on top of the PoC, have to make use of only two cryptographic primitives, which are introduced in Chapter 2: cryptographic hash functions and digital signatures. All other constructs are built on top of these concepts, for example the Merkle tree. Ethereum itself does not make use of other cryptographic primitives aside from the precompiled zkSNARK verification contract [4], which can be used in smart contracts.

An advantage of the absence of privacy preserving transactions (see Section 5.2) is, that resource-constrained devices can easily verify transactions and states with Merkle proofs. In contrast to Zcash, these clients do not have the burden to do a computation-heavy calculation in order to interact with the blockchain. Another advantage is, that we also do not have to make the additional assumption of the correct execution of the one-time trusted setup in order to initialize zkSNARK [74].

The Merkle Mountain Range tree is a construct, where elements are inserted and a binding commitment of constant size in form of a hash value is produced. To prove the inclusion of an element, only the corresponding Merkle proof has to be given. A similar, but arguably more complex, construct is the cryptographic accumulator [75]. It also produces short commitments and the inclusions of elements can be proven with short membership proofs. Accumulators do not make use of hash functions and their properties, because they are built on top of other mathematical assumptions, such as the strong RSA assumption and a concept called *adaptive root assumption*.

13.2 Consensus Change

It seems that Ethereum is going to change the currently deployed Ethash in favor of *ProgPow* [76] [77] [78]. ProgPow stands for Programmatic Proof-of-Work and can be seen as an extension to Ethash. The goal is to help graphic processing units (GPUs) becoming competitive against ASIC (Application Specific Integrated Circuit) devices, because of the argument that GPUs lead to more decentralization compared to the usage of specially designed chips. In order to make GPUs more powerful for mining activities, ProgPow dynamically changes the mining problem regularly to make use of the flexibility property of GPUs. ASICs, on the other hand, do not allow such flexibility and therefore the incentives to develop and use ASICs are minimized. An implementation of the algorithm can be found in [79], whereas the corresponding Ethereum Improvement Proposal (EIP) can be found in [80]. It does not seem, that this change could impact the proof-of-work validation time on resource-constrained devices negatively. Because it is still a variant of proof-of-work, no changes are necessary to the FlyClient's approach and as a consequence no changes are necessary to the introduced light client besides hardcoding the switch-over to the new proof-of-work validation code.

In the future, Ethereum is going to replace its consensus algorithm with *proof-of-stake* (PoS) [81]. In this system everyone with funds can participate in the consensus-building process. A node has to deposit cryptocurrency units in order to become a validator. Validators are responsible to create and vote for valid blocks. If they follow the protocol honestly, then their deposits earn interest and transaction fees. Otherwise, they have to pay penalties and their deposits decrease.

Bünz et al. [6] argue, that FlyClient cannot be deployed on a hybrid PoW/PoS system as-is, because most hybrid approaches use an additional special blocktype called *identity block*, which is issued on a lower rate compared to normal blocks. Every identity block contains a list of new validators and it has to be signed by the current validators in order to transfer the privilege of the generation of new blocks. If the client does not validate all such blocks since the genesis block in order to check the validity, an adversary could pick a block, which is not checked by the client, and fake the chain of identity blocks after this block by generating new fake identities. The FlyClient succeeds in detecting such cheating in a PoW setting, because the adversary has to maintain a valid chain after the fork point with a higher difficulty compared to the chain from the honest miners.

This is not possible, if the security parameter is reasonably bounded by the threat model. On the other hand, generating fake signatures and fake blocks after the fork point in the hybrid setting is not computational expensive. Therefore, the adversary has a high success probability.

The FlyClient paper also argues, that the presented approach can be used in PoS with minimal changes, but the authors did not provide an explanation for this statement. To the best of the thesis' knowledge, also in a PoS setting validator sets can change anytime, and to detect misbehavior every block since the Genesis block has to be checked. The adversary only has to create a short sequence of invalid blocks at some point in order to transfer the sole control to his fake identities. Then, the adversary is the only entity allowed to create new blocks in his blockchain fork. If it is the case that in PoS all blocks have to be checked in order to detect cheating, then the FlyClient's approach is not feasible and therefore also the introduced light client.

13.3 IoT Applications

Ethereum is not limited to applications described so far, such as wallets. This section deals with various other applications, which can make use of an Ethereum light client on resource-constrained devices.

In [82] Reilly et al. argue that their Ethereum light client can be used for the communication of critical data with a strong focus on integrity in a smart city context. Their client is able to create transactions and broadcast them to full nodes. In order to find out about the current state of the transactions, the client has to query full nodes. It seems, that the client does not validate any blocks of the blockchain, therefore it has to trust full nodes completely. The reason, why they use a blockchain, is that (1) other participants can validate transactions issued from these light clients and can be assured that the integrity of the message is preserved because of the included signature and (2) the blockchain is publicly accessible, which allows them to view all transactions of a specific light client. The FlyClient's approach can help in this case, because their light client additionally gains the ability to verify all information gathered from full nodes without relying on third-party trust.

Access control management of IoT devices can also benefit from blockchains. Novo [83] introduces a proof-of-concept, where a smart contract defines access rules. These rules are enforced by IoT devices on doors. The actual devices have to be connected to a management hub, which acts like a bridge to evaluate the current state of the smart contract and sends the result to the devices. The management hub is necessary, because the IoT devices are not powerful enough to validate the blockchain. Also in this case, the FlyClient's can help to make management hubs redundant. Obviously this is only possible, if the devices are powerful enough to communicate directly with full nodes and to validate FlyClient proofs.

A possible field of application for blockchains is the energy sector. Andoni et al. [84]

systematically reviewed research projects and startups in this sector. A specific use case is peer-to-peer (P2P) energy trading for end-customers. A particular project in their survey uses a Raspberry Pi 3 as a smart meter together with the Ethereum blockchain in order to make energy trading between residents possible.

Such projects can also benefit from the PoC implementation because of the possibility of smart meters and other devices to validate the blockchain efficiently in order to interact with smart contracts. A simple utilization of the proof-of-concept in a P2P energy trading setup would be as follows:

- Every resident is equipped with a Raspberry Pi 3B+, which is connected to the smart meter for incoming electricity. If the resident is also a power producer, then the smart meter also handles outgoing electricity.
- The RPi uses the proof-of-concept implementation to validate the newest block header and to interact with a payment channel smart contract.
- On top of the implementation, the resident uses a payment channel network with all other residents to send and receive funds efficiently.
- A software has to run on the RPi, which determines the lowest price of energy and buys electricity from other residents. If the user is a power producer, then he wants to sell the surplus for a price, which maximizes his profit.

CHAPTER 14

Conclusion

This thesis recapitulated functional principles of Ethereum, the FlyClient's approach and relevant cryptographic primitives. In order to take scalability considerations into account, the payment channel technique was described. Then, the thesis evaluated existing applications on top of Ethereum and arguments were given that either these applications have to use a computational-heavy full node to power them, or they must trust information gathered from third-party services. The next step was the introduction and development of a light client proof-of-concept implementation based on using FlyClient, which can be deployed on a Raspberry Pi 3B+. Performance measurements were conducted for this proof-of-concept. Afterwards, the thesis discusses how the previously mentioned existing applications can benefit from light clients. Next, different types of attacks and the possibilities for adversaries against the presented light client approach were described. The simplicity of FlyClient's approach regarding the deployed cryptographic primitives, consensus changes of Ethereum and other possible fields of applications of the light client were discussed.

The conclusion of the thesis is the following: if Ethereum adopts the Merkle Mountain Range tree hash commitment in its block header structure, then efficient light clients can be built, which do not rely on third-party trust. The light client has considerably lower CPU/RAM/bandwidth and storage requirements compared to clients using the traditional SPV verification. Therefore, this allows a large amount of different applications to run on resource-constrained devices in a completely decentralized environment without the need of third party services. The thesis sketched the following procedures: (1) a wallet application, (2) transferring money over payment channels, (3) applications in the context of IoT. These applications are possible to construct, because the light client is able to efficiently fetch and validate the latest valid block header. This header contains the root hash values of the transaction, receipt and state data structures. The applications' required information can be validated with Merkle proofs against these root hashes.

Therefore, a large amount of other smart contract applications can be developed, which can run on this light client.

Ethereum is an interesting new way for developing and interacting with decentralized applications, therefore it would be of general interest to observe the future of light clients in this network and if the Ethereum community will adopt the FlyClient's approach.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, Dec 2008. Accessed: 2019-05-24.

[2] Y. Sompolinsky and A. Zohar, "Bitcoin's security model revisited," *arXiv preprint arXiv:1605.09193*, 2016.

[3] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Advances in Cryptology-EUROCRYPT 2015*, pp. 281–310, Springer, 2015.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger." https://ethereum.github.io/yellowpaper/paper.pdf, 2014. Accessed: 2019-05-24.

[5] B. et al., "White paper." https://github.com/ethereum/wiki/wiki/White-Paper, 2019. Accessed: 2019-05-24.

[6] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "Flyclient: Super-light clients for cryptocurrencies." Cryptology ePrint Archive, Report 2019/226, 2019. https://eprint.iacr.org/2019/226.

[7] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, and E. Gün, "On scaling decentralized blockchains," in *3rd Workshop on Bitcoin and Blockchain Research, Financial Cryptography 16*, 2016.

[8] Y. Kwon, H. Kim, J. Shin, and Y. Kim, "Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash?." arXiv:1902.11064, 2019.

[9] J. Poon and T. Dryja, "The bitcoin lightning network," 2016. Accessed: 2019-05-24.

[10] "Raiden network - high speed asset transfers for ethereum." http://raiden.network, 2019. Accessed: 2019-05-24.

[11] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016.

[12] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd ed., 2014.

[13] "Which cryptographic hash function does ethereum use?." https://ethereum.stackexchange.com/questions/550/which-cryptographic-hash-function-does-ethereum-use<Plug>CocRefresh, 2016. Accessed: 2019-06-10.

[14] "What are the key differences between the draft sha-3 standard and the keccak submission?." https://crypto.stackexchange.com/questions/15727/what-are-the-key-differences-between-the-draft-sha-3-standard-and-the-keccak-sub, 2014. Accessed: 2019-06-10.

[15] N. D. Bhaskar and D. L. K. Chuen, "Chapter 3 - bitcoin mining technology," in *Handbook of Digital Currency* (D. L. K. Chuen, ed.), pp. 45 – 65, San Diego: Academic Press, 2015.

[16] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87* (C. Pomerance, ed.), (Berlin, Heidelberg), pp. 369–378, Springer Berlin Heidelberg, 1988.

[17] "Merkle mountain ranges." https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, 2016. Accessed: 2019-05-25.

[18] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, pp. 36–63, Aug 2001.

[19] "Sec 2: Recommended elliptic curve domain parameters." http://www.secg.org/sec2-v2.pdf, 2010. Accessed: 2019-06-11.

[20] "Elliptic curve digital signature algorithm." https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm, 2019. Accessed: 2019-06-11.

[21] "Secp256k1." https://en.bitcoin.it/wiki/Secp256k1, 2019. Accessed: 2019-06-11.

[22] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*, pp. 139–147, Springer, 1992.

[23] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols," in *Secure Information Networks*, pp. 258–272, Springer, 1999.

[24] B. et al., "Ethash." https://github.com/ethereum/wiki/wiki/Ethash, 2018. Accessed: 2019-05-25.

[25] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *23rd ACM Conference on Computer and Communications Security (ACM CCS 2016)*, Oct 2016.

[26] "Solidity." https://solidity.readthedocs.io/en/v0.5.8/, 2019. Accessed: 2019-05-25.

[27] "Openzeppelin is a library for secure smart contract development." https://github.com/OpenZeppelin/openzeppelin-solidity, 2019. Accessed: 2019-06-14.

[28] F. Victor and B. K. Lüders, "Measuring ethereum-based erc20 token networks," in *Financial Cryptography and Data Security* (I. Goldberg and T. Moore, eds.), (Cham), pp. 113–129, Springer International Publishing, 2019.

[29] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges." arXiv preprint arXiv:1904.05234, 2019.

[30] "Ethereum chain data size growth." https://etherscan.io/chartsync/chaindefault, 2019. https://etherscan.io/chartsync/chaindefault.

[31] "Ethereum average block time chart." https://etherscan.io/chart/blocktime, 2019. https://etherscan.io/chart/blocktime.

[32] M. Jourenko, K. Kurazumi, M. Larangeira, and K. Tanaka, "Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies." Cryptology ePrint Archive, Report 2019/352, 2019. https://eprint.iacr.org/2019/352.

[33] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than lightning," 2017. Accessed: 2017-03-22.

[34] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability." Cryptology ePrint Archive, Report 2018/472, 2018. https://eprint.iacr.org/2018/472.

[35] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks." Cryptology ePrint Archive, Report 2017/823, 2017. Accessed:2017-09-26.

[36] G. Avarikioti, F. Laufenberg, J. Sliwinski, Y. Wang, and R. Wattenhofer, "Towards secure and efficient payment channels," *CoRR*, vol. abs/1811.12740, 2018.

[37] "Bitcoin developer reference: Find technical details and api documentation.." https://bitcoin.org/en/developer-reference#block-chain, 2019. Accessed: 2019-05-30.

[38] "Blockchain." https://bitcoin.org/en/blockchain-guide#proof-of-work, 2019. Accessed: 2019-05-30.

[39] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 459–474, IEEE, 2014.

[40] K. Wüst, S. Matetic, M. Schneider, I. Miers, K. Kostiainen, and S. Capkun, "Zlite: Lightweight clients for shielded zcash transactions using trusted execution." Cryptology ePrint Archive, Report 2018/1024, 2018. https://eprint.iacr.org/2018/1024.

[41] V. Costan and S. Devadas, "Intel sgx explained." Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[42] A. Poelstra, "Mimblewimble." https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf, 2016.

[43] "Running an ethereum node." https://docs.ethhub.io/using-ethereum/running-an-ethereum-node/, 2019. Accessed: 2019-05-27.

[44] A. Kiayias, A. Miller, and D. Zindros, "Non-interactive proofs of proof-of-work." Cryptology ePrint Archive, Report 2017/963, 2017. Accessed:2017-10-03.

[45] S. D. Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain," in *Italian Conference on Cyber Security (06/02/18)*, January 2018.

[46] L. Bahack, "Theoretical bitcoin attacks with less than half of the computational power," 2013. Accessed: 2017-11-23.

[47] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source," 2015. Accessed: 2015-10-25.

[48] S. Goldwasser and M. Sipser, "Private coins versus public coins in interactive proof systems," in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, (New York, NY, USA), pp. 59–68, ACM, 1986.

[49] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86* (A. M. Odlyzko, ed.), (Berlin, Heidelberg), pp. 186–194, Springer Berlin Heidelberg, 1987.

[50] "How is jaxx able to support over 70 currencies in its mobile app?." https://www.reddit.com/r/jaxx/comments/aq9jtq/how_is_jaxx_able_to_support_over_70_currencies_in/, 2019. Accessed:2019-05-27.

[51] "Myetherwallet: how does it work (offline/online)." https://www.reddit.com/r/MyEtherWallet/comments/7qgyn1/how_does_it_work_offlineonline/, 2018. Accessed:2019-05-27.

[52] "Coinbase: What is it and how do you use it?." https://www.investopedia.com/tech/coinbase-what-it-and-how-do-you-use-it/, 2019. Accessed:2019-05-27.

[53] "To which remote ethereum nodes does metamask plugin send signed transactions? and are they exposed to denial of service attacks?." https://ethereum.stackexchange.com/questions/13362/to-which-remote-ethereum-nodes-does-metamask-plugin-send-signed-transactions-an, 2019. Accessed:2019-05-27.

[54] "Raiden network 0.100.3 documentation." https://raiden-network.readthedocs.io/, 2019. Accessed: 2019-05-30.

[55] "Publi project launch: Raiden light client sdk and dapp." https://medium.com/raiden-network/public-project-launch-raiden-light-client-sdk-and-dapp-140a546c63a0, 2019. Accessed: 2019-05-30.

[56] W. Warren, "Canonical weth." https://blog.0xproject.com/canonical-weth-a9aa7d0279dd, 2017. Accessed: 2019-06-01.

[57] "Microraiden: A payment channel framework for fast & free off-chain erc20 token transfers." https://raiden.network/micro.html, 2019. Accessed: 2019-05-30.

[58] "Microraiden: Blockchain setup." https://microraiden.readthedocs.io/en/latest/tutorials/blockchain.html, 2019. Accessed: 2019-05-30.

[59] "Cryptokitties." https://www.cryptokitties.co/technical-details, 2019. Accessed: 2019-05-30.

[60] "Erc-721 non-fungible token standard." https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md, 2019. Accessed: 2019-05-30.

[61] "0x docs." https://0x.org/docs, 2019. Accessed: 2019-07-01.

[62] M. Saito and M. Matsumoto, "A prng specialized in double precision floating point numbers using an affine transition," in *Monte Carlo and Quasi-Monte Carlo Methods 2008* (P. L' Ecuyer and A. B. Owen, eds.), (Berlin, Heidelberg), pp. 589–602, Springer Berlin Heidelberg, 2009.

[63] I. Stewart, D. Ilie, A. Zamyatin, S. Werner, M. Torshizi, and W. Knottenbelt, "Committing to quantum resistance: A slow defence for bitcoin against a fast quantum computing attack." Cryptology ePrint Archive, Report 2018/213, 2018.

[64] Y. Marcus, E. Heilman, and S. Goldberg, "Low-resource eclipse attacks on ethereum's peer-to-peer network." Cryptology ePrint Archive, Report 2018/236, 2018. https://eprint.iacr.org/2018/236.

[65] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," Oct 2016. Accessed: 2016-11-08.

[66] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts." arXiv:1802.08660, 2018. Accessed:2018-03-12.

[67] D. Annenkov and B. Spitters, "Towards a smart contract verification framework in coq," 2019.

[68] I. Sergey, A. Kumar, and A. Hobor, "Temporal properties of smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pp. 323–338, 2018.

[69] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Safevm: A safety verifier for ethereum smart contracts," 2019.

[70] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum." arXiv:1901.01292, 2019.

[71] A. Biryukov and I. Pustogarov, "Bitcoin over tor isn't a good idea," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 122–134, IEEE, 2015.

[72] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber, "On the privacy provisions of bloom filters in lightweight bitcoin clients," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 326–335, ACM, 2014.

[73] K. Kanemura, K. Toyoda, and T. Ohtsuki, "Design of privacy-preserving mobile bitcoin client based on γ-deniability enabled bloom filter," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pp. 1–6, IEEE, 2017.

[74] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture." Cryptology ePrint Archive, Report 2013/879, 2013. `https://eprint.iacr.org/2013/879`.

[75] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains." Cryptology ePrint Archive, Report 2018/1188, 2018. `https://eprint.iacr.org/2018/1188`.

[76] "What is ethereum's progpow and how is it impacting miners?." `https://finance.yahoo.com/news/ethereum-progpow-impacting-miners-110027936.html`, 2019. Accessed: 2019-06-12.

[77] "Ethereum's progpow mining change approved again, but timeline unclear." `https://www.coindesk.com/ethereums-progpow-mining-change-approved-again-but-timeline-unclear`, 2019. Accessed: 2019-06-12.

[78] "Funding approved for audit of ethereum's progpow mining proposal." https://www.coindesk.com/funding-approved-for-audit-of-ethereums-progpow-mining-proposal, 2019. Accessed: 2019-06-12.

[79] "Progpow - a programmatic proof of work." https://github.com/ifdefelse/ProgPOW, 2019. Accessed: 2019-06-12.

[80] "Eip 1057: Progpow, a programmatic proof-of-work." https://eips.ethereum.org/EIPS/eip-1057, 2019. Accessed: 2019-06-13.

[81] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017.

[82] E. Reilly, M. Maloney, M. Siegel, and G. Falco, "An iot integrity-first communication protocol via an ethereum blockchain light client," 05 2019.

[83] O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot," *IEEE Internet of Things Journal*, vol. PP, pp. 1–1, 03 2018.

[84] M. Andoni, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum, and A. Peacock, "Blockchain technology in the energy sector: A systematic review of challenges and opportunities," *Renewable and Sustainable Energy Reviews*, vol. 100, pp. 143 – 174, 2019.

APPENDIX A

Terminology

ASIC	ASIC stands for Application Specific Integrated Circuit. These hardware chips are built in order to execute only one specific application, but to perform well. An example is a Bitcoin mining ASIC, which outperforms CPUs and GPUs by calculating SHA-256 faster than these devices.
Block	A data structure, which contains two different types of data: (1) transactions and (2) various other important data in form of the block header.
Blockchain	It is a chain of blocks, where every block references an earlier block. The consensus algorithm is important in order to generate a chain, where a block is only referenced by exactly one block and miners build on top of the longest chain.
Block header	It is part of a block. It contains information regarding consensus-building and can contain cryptographic commitments, such as hash pointers to transactions.
Cryptocurrency	Currencies, which are built on top of a blockchain, are also called cryptocurrencies. For example, Bitcoin is a cryptocurrency. Ethereum is also often called a cryptocurrency despite the fact that it is a more generalized version of a currency with smart contract capabilities.
Client	In this thesis clients are an abbreviation of light clients.
ECDSA	ECDSA stands for Elliptic Curve Digital Signature Algorithm. It is a signature algorithm, which uses properties of a mathematical construct called elliptic curve in order to generate key pairs. A goal is to make computation of the private key infeasible if the adversary has only access to the public key, signed messages, and the curve parameters.

Epoch	In Ethereum blocks are divided into epochs. Every 30000 blocks in the chain belong to the same epoch.
ERC-20	It is a token standard, which defines interface methods. In order to comply with this standard, the token developer has to implement the corresponding contract methods. A goal of this standard is the uniform token handling via other smart contracts or wallet applications.
ERC-721	It is similar to the ERC-20 standard. The difference is, that this standard is used for non-fungible tokens, where the owner address is stored in the contract for every single token. On the other hand, in the ERC-20 standard addresses are mapped to amounts of tokens.
Ethash	Ethash is the currently deployed proof-of-work algorithm as part of the consensus protocol in Ethereum, with a focus on the memory-hardness property. The goal is to discourage the development and usage of Ethash ASICs.
Ether	The built-in currency in the Ethereum network is called Ether and it is used to pay for transaction fees, i.e. gas.
EVM	The Ethereum Virtual Machine is responsible for state transitions, which are triggered by transactions. In practical terms, it can be seen as the computer which executes smart contracts.
IoT	IoT stand for Internet of Things. These are small computers, which are typically resource-constrained and have access to the Internet. In the context of this thesis, a Raspberry Pi 3B+ is considered as an IoT device.
Fee	In order to pay for transactions in the Ethereum network, the user has to specify a gas price. The used gas of the transaction multiplied with the gas price corresponds to the Ether value the user has to pay for the transaction fee.
Full node	A full node is a type of node, which hosts the complete blockchain. Sometimes it is only called node throughout the paper, which means the same thing. The important differentiation to the light client is the storage and processing capabilities for the complete chain.
Gas	Every execution step within a transaction (in order to send balances to other addresses or to call smart contract operations) has a specific gas cost. The transaction fees depend on the used gas.
Hashrate	Hashrate is the unit used to measure speed of calculating hashes for proof-of-work mining purposes.

Light Client	This is a client, which does not host the complete blockchain. Reasons can be that (1) it does not have enough CPU/RAM resources to validate the complete chain, (2) it is not connected to a fast network connection to validate the chain in time, (3) it has a limited cellphone contract, such that downloading the complete chain is infeasible. SPV (simple payment verification) is an efficient blockchain validation approach to gain trust in the chain without relying on a third party.
Miner	This type of node also takes part in the consensus-building process.
Node	In the context of cryptocurrencies and other peer-to-peer protocols, a node is an independent computer, which is connected with other computers via a peer-to-peer protocol over the Internet.
Nonce	It is the block header field, which a miner can alter to brute-force a desired block hash. In addition, the number of transactions, which every address has already conducted, is stored in the state tree in a variable called nonce.
On-chain	This term means that the interaction between parties is conducted via blockchain transactions.
Off-chain	This term has the opposite meaning of on-chain. In this case the interaction between parties happens without the use of transactions. An example to illustrate the difference is a payment channel. Channel opening and closing has to be done on-chain, whereas the exchange of signed messages is done directly between the participants, id est off-chain.
Payment channel	It is a construct, where funds can be transferred between parties with the direct exchange of signed messages. The blockchain is only used to open and close a channel between participants and to settle disputes.
Proof-of-work	PoW is a technique for proving that a certain amount of computational effort was expended, it is often based on hash functions.
Root hash	In this thesis it refers to the hash value of certain data structures. In a Merkle tree or Merkle Mountain Range tree it denotes the root value of the tree.
RLP	RLP stands for Recursive Length Prefix. It is a serialization method used in Ethereum, which is able to encode arbitrarily structured binary data [4].
Server	In this thesis server refers to nodes, which host the complete blockchain.

Smart Meter	In the context of this paper a smart meter is an electric meter, which also offers other functionalities like recording power usage per time and providing access to different statistics of the collected data.
Smart Contract	Smart contracts are programs, which can be deployed on the Ethereum network. They keep their state in variables, which can be altered by functions. Transactions are used in order to invoke these functions.
State channel	This construct allows exchanging states between parties with the direct exchange of signed messages. Compared to payment channels this technology is a more generalized one, where parties can reach an agreement on the state of arbitrary variables.
State root	It is a field in the Ethereum block header, which denotes the root hash of the data structure containing the complete, current state of Ethereum.
Third-party trust	In the context of this paper, this means that a light client has to trust external servers for the validity of blockchain information. The reason is, that the client is not able to verify the chain itself.
Token	A token can be seen as a similar concept to a currency. Tokens are built on smart contracts, which enable users to have a specific amount stored in a contract variable. Users can send their own amount of tokens to other users with transactions. The result of such a transaction is, that the specific amount of tokens of the payer is reduced by a value of x, and the amount of tokens possessed by the payee is increased by x. Such tokens can get valuable if they are restricted by a total supply. This concept can be used to represent different things, like shares of a company or an in-game currency.
Transaction	Ethereum transactions are similar to Bitcoin transactions, which are used in order to send cryptocurrency units from one address to another. In Ethereum such transactions are also responsible to invoke smart contract methods and to create smart contracts.
Transaction root	It is a field in the Ethereum block header, which denotes the root hash of a data structure containing all transactions in the specific block.

APPENDIX B

Proofs

Theorem 1. *Let m be a valid Merkle Mountain Range tree, then the sum of the intermediate nodes and one root node is one less than the sum of the leaf nodes.*

Proof. Let A_n denote the following assumptions: (a) the current MMR has n leaves and the number of the other nodes is $n-1$, (b) the root node has exactly two child nodes, but no parent node, (c) intermediate nodes have exactly one parent node and two child nodes, (d) the depth of the right branch of an intermediate or root node is always less than or equal to the left branch, (e) new leaf nodes have to be inserted to the right of all existing leaf nodes and the sequence of these nodes cannot be changed in any point in time. Obviously, no circles are allowed in a Merkle Mountain Range tree, leaves have exactly one parent node and no child nodes, and the tree is one connected component. If these assumptions are fulfilled, we say that the MMR is valid. A_n is used as the induction hypothesis.

Consider the base case A_2. There exists one root node, which has the first leaf node as its left child and the second leaf node as its right child. This is a valid MMR, because it satisfies A_n.

For the induction step from A_n to A_{n+1}, which deals with the addition of a new leaf, we have to distinguish two cases:

(1) If the current MMR has a leaf number of k, where $\exists l \in \mathbb{N}_0, k = 2^l$, then we say that the MMR is full and we have to add a new root node, where the left child is connected to the root node of the old MMR (the old root node is an intermediate node now) and the right child is the new leaf node. Because we only added one leaf and one root node, assumption (a) is still true. (d) is true, because the subtree on the right of the root node consists only of a leaf node, where the depth cannot be higher compared to the existing subtree on the left. All other assumptions are still true, hence the resulting MMR is valid.

(2) We have the case, that the current MMR is not full. Therefore, a new leaf is added in the branch on the right of the root node, but the root node stays the same. Assumption (c) is only satisfied by adding exactly one intermediate node, because the new leaf node has to be connected somehow to the existing branch. The question arises how some intermediate nodes have to be altered in order to get a valid MMR again without violating assumption (d). The procedure to insert the new leaf node is as follows:

1. Delete all intermediate nodes in the right branch of the root node.
2. The new leaf node is added on the right of the existing leaf nodes.
3. Every two leaf nodes are connected to an intermediate node. It can happen, that the rightmost leaf node is not getting connected with an intermediate node in this step.
4. Every two existing intermediate nodes are connected to a new intermediate node. If it is the case, that the rightmost intermediate node is not connected to a new intermediate node in this step, then the intermediate node gets connected with the rightmost leaf node if and only if the rightmost leaf node is still unconnected.
5. Every two existing intermediate nodes in the highest level gets connected to a new intermediate node. Every time the level has an odd number of nodes, the rightmost node has to be connected to a new intermediate node together with an unconnected node (a node without a parent) in lower levels if and only if such a node exists. This step is repeated as long as there is only one node left without the possibility to connect to a node from a lower level. This node is then connected to the root node of the MMR.

The outlined procedure satisfies (a), because we only added a new leaf node and an additional intermediate node. Otherwise it is not possible to create a subtree with this amount of leaf nodes without violating (c). Assumption (b) is still true, because only the structure of the subbranch of the root node was modified. (c) is true because of the construction where every two nodes are connected to an intermediate node and these intermediate nodes are also connected to new ones up to exactly one remaining intermediate node which was then connected to the root node. (d) is fulfilled in the right branch of the root node, because the construction always started on the left and a connection between nodes of different levels could only occur, if the right node was on a lower level. Because of case (2) we know that, after adding the new leaf, the right branch has the same or less amount of leaf nodes, therefore the same or a lower depth is possible. The outlined procedure inserts intermediate nodes in way, such that the width of the branch gets maximized, therefore the depth is optimal for the given amount of leaf nodes. Therefore, (d) is also true for the complete MMR. (e) is true, because the new leaf node was inserted on the rightmost position. Hence, the resutling MMR is valid.

□

APPENDIX C

Implementation Details

This chapter describes implementation details of the proof-of-concept application. The complete source code can be found on Github [1]. The repository contains the light client and light server implementation illustrated in Figure 9.1 in order to generate and validate FlyClient proofs and to validate the proof-of-work of Ethereum block headers.

The Merkle Mountain Range tree could be implemented with the help of pointers, which connects the corresponding nodes with each other. An insertion could be done by adding new nodes and altering some pointers. Traversing operations would consist of following the pointers to the target nodes. As already mentioned in Section 9.2, the server application makes use of another approach for dealing with the MMR data structure. The tree is implemented as an array of bytes, where every node consists of a 32 bytes hash and a 16 bytes difficulty number. Insertions are done on the right end exclusively, because the MMR is an append-only data structure. Traversing the array can be done by calculating the position of the target node (for example the children of a specific node), because the MMR has properties which can be used for such procedures. In this chapter some methods are explained in more detail, which the implementation makes use of. Therefore, these code snippets are valid Rust code.

C.1 Calculate Different MMR Properties

From Theorem 1 we know that the sum of intermediate nodes and root node is one less than the sum of leaf nodes. If we have x leaves, we would have a total of $2x - 1$ nodes. Therefore, the following code snippet shows, how we can compute the total node number, if we have the number of leaves given:

```
1 fn leaf_to_node_number(leaf_number: u64) -> u64 {
```

[1] *https://github.com/patrick0210/FlyClient_Ethereum_Prototype, Accessed: 2019-28-11*

```
    (2 * leaf_number) - 1
}
```

If we want the inverse, the following snippet calculates this for us:

```
fn node_to_leaf_number(node_number: u64) -> u64 {
    (node_number + 1) / 2
}
```

If we have the number n of leaves given for a subtree with root node X, then the following code can calculate the number k of leaves of the left child of X. This is possible, because of the MMR property that the left subtree has k leaves, where $k = \max\{2^l : \{l \in \mathbb{N}_0 | l \leq \log_2(n)\}\})$. In other words, the leaf number on the left subtree is always a power of two and it has the same number or more leaves compared to the right subtree.

```
fn get_left_leaf_number(leaf_number: u64) -> u64 {
    if leaf_number.is_power_of_two() {
        leaf_number / 2
    } else {
        leaf_number.next_power_of_two() / 2
    }
}
```

We can also calculate the depth of the MMR with the number of leaves as input. Every MMR, which has k leaves, where $\exists l \in \mathbb{N}_0, k = 2^l$, has exactly a depth of l. Is the leaf number not a power of two, then an additionally depth has to be added to the depth of the next smaller number, which is a power of 2. In the following snippet, the binary logarithm is calculated by summing up the leading zeros in order to calculate the depth:

```
fn get_depth(leaf_number: u64) -> u32 {
    let mut depth = 64 - leaf_number.leading_zeros() - 1;
    if !leaf_number.is_power_of_two() {
        depth += 1;
    }
    depth
}
```

C.2 Element Insertion in MMR

Figure C.1 shows the MMR tree and the corresponding memory representation for the insertion of the first four elements. Elements are only added at the end, and only a logarithmic number of nodes has to be altered in order to obtain a valid MMR again. Note that in the transition of one to two and two to three elements only a new root has to be defined, which connects the old tree with the new element. This procedure can always be done if the old leaf number is a power of two. Otherwise, for example in the case of the transition of three to four elements, the current root node has to be deleted and the new element with a new intermediate node and a new root node has to be added. As already mentioned, the changes are logarithmic in the number of the nodes, therefore the byte array is an efficient representation of an MMR.

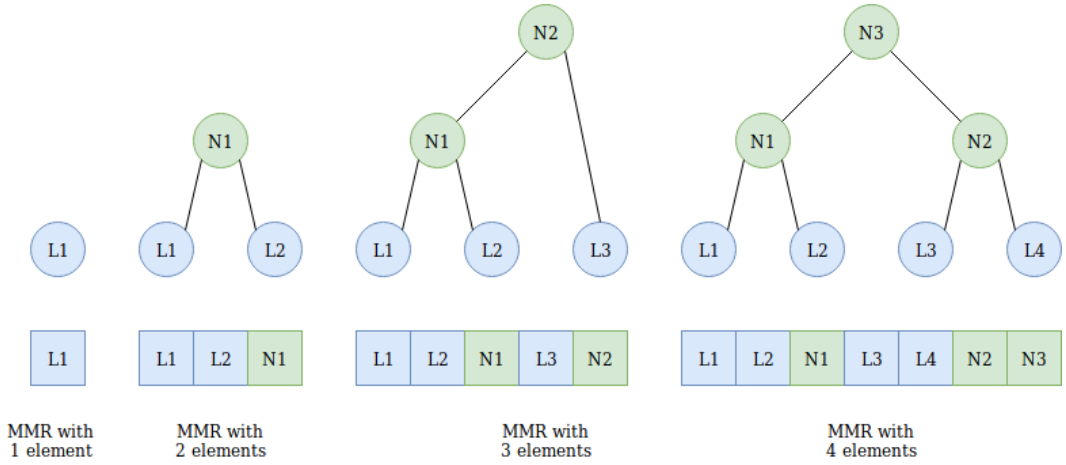

Figure C.1: Insertion of elements into an MMR

The next code snippet shows the insertion of an element into the MMR byte array. The new element only consists of a hash and difficulty number (line 1). The first loop (line 13) is responsible to traverse the tree by always using the right branch as long as the leaf number of the right subtree is not a power of two. This means, that the algorithm looks for the correct subtree, where the new element has to be inserted. The loop also deletes the root node of every subtree it traverses and stores the left subtree root nodes in a temporary nodes variable. At the end of the traversal it also stores the left sibling of the new element. Then the new element is added to the byte array. The last step is the loop in line 28, which hashes the new element with the left sibling to get a new node and write it to the byte array, then it uses this new node and hashes it with the last node element from the temporary variable and stores it again in the byte array. This procedure gets repeated as long as we have more than one stored node in the temporary nodes variable. The last node is the root node of the tree, which also gets stored in the byte array.

In short, the first loop of the algorithm removes a specific path. Then, the new element gets inserted. Finally, the second loop hashes nodes together and stores them in the byte array to recreate a new path in order to get a valid MMR again.

```
1  fn append_leaf(&mut self, hash: H256, difficulty: u128) {
2      let mut new_elem = MmrElem {
3          hash,
4          difficulty,
5          position_in_datastore: None,
6      };
7
8      let mut nodes_to_hash = vec![];
9      let mut curr_tree_number = self.leaf_number;
10
11     let mut aggr_node_number = 0;
12
13     while !curr_tree_number.is_power_of_two() {
```

```
14          self.datastore.remove_last_elem().unwrap();
15
16          let left_tree_number = curr_tree_number.next_power_of_two() / 2;
17          aggr_node_number += left_tree_number;
18          let right_tree_number = curr_tree_number - left_tree_number;
19
20          let left_root_node_number = get_node_number(aggr_node_number) - 1;
21          nodes_to_hash.push(self.datastore.get_elem(left_root_node_number));
22          curr_tree_number = right_tree_number;
23      }
24      nodes_to_hash.push(self.datastore.get_root_elem());
25      self.datastore.write_elem(&mut new_elem).unwrap();
26      nodes_to_hash.push(new_elem);
27
28      while nodes_to_hash.len() > 1 {
29          let curr_right_elem = nodes_to_hash.pop().unwrap();
30          let curr_left_elem = nodes_to_hash.pop().unwrap();
31          let hash = hash_children(&curr_left_elem, &curr_right_elem);
32          let difficulty = curr_left_elem.difficulty + curr_right_elem.
    difficulty;
33          let mut new_intermediate_node = MmrElem {
34              hash,
35              difficulty,
36              position_in_datastore: None,
37          };
38          self.datastore
39              .write_elem(&mut new_intermediate_node)
40              .unwrap();
41          nodes_to_hash.push(new_intermediate_node);
42      }
43      self.leaf_number += 1;
44  }
```

C.3 Get children by node number

Because the tree is implemented as a byte array instead of pointers, another approach has to be used in order to retrieve child nodes. Let x denote the node, whose children have to be determined. Because every element has a size of exactly 48 bytes, the MMR node position of x (line 5) and the sum of all nodes (line 6) can be computed from their byte position in the array by dividing by 48. The loop in line 9 executes as long as the current subtree has more than two elements, otherwise it is not possible anymore that the subtree contains a node, which has two child nodes, and the algorithm panics. The loop in line 9 in combination with the if-statements on line 28 and 31 are responsible for branching in the right direction in order to find the children of x.

If the if-statement on line 14 evaluates to true, then we have reached the root node of the left subtree of node x. The root node of the left child's subtree and the root node of the right child's subtree can then be computed and returned.

```
1  fn get_children<T: Datastore>(
```

```
    &mut self,
    mmr: &mut MerkleTree<T>,
) -> (Box<MmrElem>, Box<MmrElem>) {
    let elem_node_number = self.position_in_datastore.unwrap() / 48;
    let mut curr_root_node_number = mmr.datastore.get_storage_size() / 48;
    let mut aggr_node_number = 0;

    while curr_root_node_number > 2 {
        let leaf_number = node_to_leaf_number(curr_root_node_number);
        let left_tree_leaf_number = get_left_leaf_number(leaf_number);
        let left_tree_node_number = leaf_to_node_number(
    left_tree_leaf_number);

        if (aggr_node_number + curr_root_node_number) == (elem_node_number
    + 1) {
            let leaf_number = node_to_leaf_number(curr_root_node_number);
            let left_tree_leaf_number = get_left_leaf_number(leaf_number);
            let left_tree_node_number = leaf_to_node_number(
    left_tree_leaf_number);

            let left_node_position = aggr_node_number +
    left_tree_node_number - 1;
            let right_node_position = aggr_node_number +
    curr_root_node_number - 2;

            let left_elem = mmr.datastore.get_elem(left_node_position);
            let right_elem = mmr.datastore.get_elem(right_node_position);

            return (Box::new(left_elem), Box::new(right_elem));
        }

        if elem_node_number < (aggr_node_number + left_tree_node_number) {
            // branch left
            curr_root_node_number = left_tree_node_number;
        } else {
            // branch right
            curr_root_node_number = curr_root_node_number -
    left_tree_node_number - 1;
            aggr_node_number += left_tree_node_number;
        }
    }

    panic!("This node has no children!");
}
```