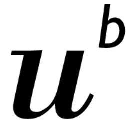

Benchmarking Threshold Signatures for Consensus Protocols

Bachelor Thesis

Julien Brunner

from

Aarau AG, Switzerland

Faculty of Science, University of Bern

23. November 2022

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

Abstract

A blockchain is a technology that allows for the creation of a distributed and decentralized network of parties that want to agree on a common sequence of records called blocks. These blocks are verified without the need of a central authority. Verification is made possible by consensus protocols which enable a set of nodes in the network to align on a common value. Some protocols employ threshold signatures for block validation. Threshold signature schemes involve shares of a private key which are distributed across different nodes. Each node can produce a partial signature on a message m by using his own share. A full threshold signature requires a minimum number of partial signatures. In this thesis, we implement an existing signature scheme, and measure its performance when being deployed in two distinct consensus protocols.

Acknowledgements

First of all I would like to express my gratitude towards Orestis Alpos for the supervision of my thesis and guiding me in the right direction when needed. Furthermore, I want to thank professor Christian Cachin for giving me the opportunity to write a thesis in his research group and to broaden my horizon in an area that I was mostly unfamiliar with before.

Contents

1 Introduction **2**

2 Background **3**
2.1 Byzantine Fault Tolerance in Blockchain . . . 3
2.2 HotStuff . . . 3
2.3 Kauri . . . 4
2.4 BLS Signature Scheme . . . 5
2.5 BLS Multisignatures . . . 6
2.6 Secret Sharing . . . 6
2.7 BLS Threshold Signatures . . . 7

3 Design **9**
3.1 Secure threshold scheme . . . 9
3.2 Star and Tree Topologies . . . 9
3.2.1 Simulate one phase in HotStuff . . . 9
3.2.2 Simulate one phase in Kauri . . . 10

4 Implementation **11**
4.1 Key Generation . . . 11
4.2 Signing and Verification . . . 11
4.3 Multisignatures and Threshold Signatures Creation . . . 13
4.4 Multisignature and Threshold Signature Verification . . . 14

5 Results **15**
5.1 Base Performance . . . 15
5.2 Swapping G_1 and G_2 . . . 16
5.3 The importance to reduce the lagrange coefficients . . . 17

6 Conclusion **19**

Chapter 1

Introduction

Consensus protocols are crucial components of current blockchain technology. They allow for honest nodes in a decentralized network to reach an agreement on a system state in order to transition to the next state. One desired property of such protocols is Byzantine Fault Tolerance (BFT). A consensus protocol implementing BFT maintains reliability despite the presence of some malicious nodes that may falsify information. This is achieved by using common cryptographic components, including signatures creation and verification, as well as hash functions. BFT allows a system with $3f + 1$ nodes, where f denotes the number of malicious parties, to reach consensus (provided that at least $2f + 1$ non-byzantine nodes, i.e. honest nodes, follow the protocol correctly). Some consensus protocols implement BFT by deploying threshold signatures as follows: A designated leader node broadcasts a message (which translates to block in our context) to the rest of the nodes in the network. Upon receiving the broadcast, a node performs a validation procedure using their private key and responds with a partial signature back to the leader. The leader waits for the arrival of sufficient valid responses before combining them into one threshold signature. This threshold signature proves the approval of the message by a sufficient number of nodes. One of the main advantages of threshold signatures is the fact that no more than a single signature is needed to approve the subsequent block. This is crucial due to restrictions in memory space requirements and broadcasting costs in the context of blockchains. In this thesis we will examine the performance of the two distinct consensus protocols HotStuff [2] & Kauri [3] when using the BLS signature scheme [1]. The main aims of this project are:

- Developing the BLS signature scheme in Go.
- Measuring the perfomance of the scheme when being deployed in both, the HotStuff and the Kauri consensus protocol.

In Chapter 2 we introduce essential cryptographic notions that build the background for the following chapters. Chapter 3 summarizes the most important design requirements and choices that were made. We explain most important functions of the BLS scheme and give some technical insights in Chapter 4, before evaluating the BLS signature scheme in Chapter 5.

Chapter 2

Background

2.1 Byzantine Fault Tolerance in Blockchain

Byzantine Fault Tolerance (BFT) is a method in order to reach consensus in a decentralized network. It aims to resolve the Byzantine Generals problem, a logical decision puzzle [19, 20]. It is based on the possibility that generals from the same side leading different armies can experience communication issues when deciding on how to proceed against the adversary. They will win the battle if all generals take the same decision. If communication between the generals is bad or some generals are behaving maliciously, causing some of the troops to attack while others don't, the battle will be lost. Byzantine faults summarize these types of difficulties. In a computer system that consists of several nodes, every single node can be regarded as a general. A computer system implementing BFT has the ability to continue operating even when some nodes may fail or purposefully try to fool it.
In blockchain technology, a certain amount of nodes must validate a transaction, before the transaction is appended to the blockchain. Consensus algorithms are part of every blockchain network. They make it possible that consensus protocols can reach BFT. A large-scale version of the Byzantine Generals Problem affects blockchain networks, and particularly cryptocurrencies. In the following two sections we are going to present different consensus algorithms, namely the HotStuff consensus protocol and Kauri. They are both designed to work on permissioned blockchains. Permissioned blockchains are not publicly accessible and only users that have permission can enter the network, contrary to permissionless blockchain [18]. Bitcoin is the most prominent example of a permissionless blockchain.

2.2 HotStuff

HotStuff is a BFT consensus protocol, where communication is based on a star topology [2]. A replica is acting as the designated leader (here P_0) for each round of consensus. First, the leader proposes a transaction m that he received from a client C to all the other replicas. For a message m to be appended onto the blockchain and the replicas updating their own state machine, the leader needs to gather votes from enough replicas over three phases, the PREPARE-Phase, the PRE-COMMIT-Phase and the COMMIT-Phase.The overall procedure in all three phases is similar:
In each phase the leader makes a proposal to the other replicas. Once the leader receives sufficient valid responses (partial signatures from replicas $P_1, ..., P_6$) for the current proposal, it combines them into a single quorum certificate QC in form of a threshold signature. He sends the quorum certificate QC to the other nodes, which can validate that the quorum certificate QC is correct and enough nodes have approved the proposal from the previous phase.
Finally (in the DECIDE-phase), assuming the leader has received enough valid commit votes, he assembles all valid votes together into a commit quorum certificate, which he broadcast as a final decide message to all the other nodes. Upon the receiving the decide message a replica may execute a state transition and update their own state machine (which represents a copy of their own blockchain). Figure 2.1 gives an overview of HotStuffs topology and communication pattern. Communication in HotStuff is

executed in linear time.

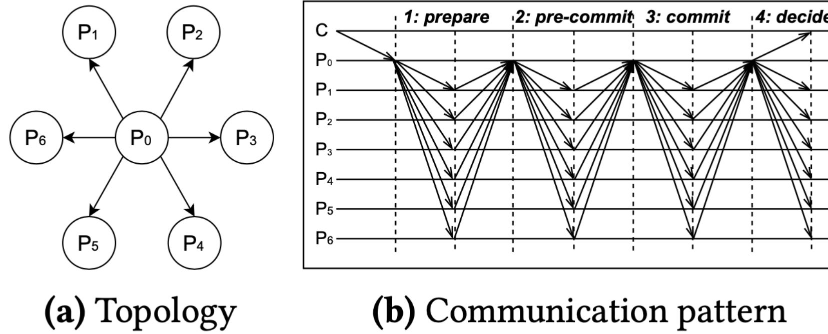

Figure 2.1. HotStuff topology and communication pattern with 7 nodes[2]

In Hotstuffs approach of using a star topology, the system can progress as long as the leader is non-faulty. Additionally, HotStuff ensures to recover within $f + 1$ steps in case of the presence of a malicious leader.

2.3 Kauri

Kauri is BFT communication abstraction and is an extension of the HotStuff consensus protocol [3]. Hotstuff's star topology can rapidly cause some problem to the leader. In HotStuff the leader needs to send, receive and process messages from all other nodes, leading to a potential bandwidth and CPU bottleneck. Instead of communicating over a star topology, in Kauri the communication pattern is a tree. It is designed to improve system scalability and load balancing in HotStuff, which is an issue since the leader works as a central node verifying the signatures of all other nodes. Similar to HotStuff, Kauri works in succession of four phases PREPARE, PRE-COMMIT, COMMIT and DECIDE. Figure 2.2 depicts Kauris tree topology and its communication pattern. The root node (here P_0) acts as the designated leader sends forwards a message m to both his children P_1 and P_2. Both, P_1 and P_2 continue sending m further down the tree by sending the message m to their own children. This procedure continues until the leaf nodes (here $P_3, ..., P_7$) receive the message that was initially sent by the leader P_0. Leaf nodes validate the message m by signing it and propagate it up the tree by sending the signature to their parents. A parent, receiving two signatures, may sign m himself before aggregating all 3 partial signatures to form a multisignature. He then sends the multisignature to his own parent. The procedure continues that way until the the leader receives two multisignatures from both his children. Those two multisignatures contain partial signatures of all nodes that validated the proposal m. He then may then sign the proposal himself and aggregate the two received multisignatures with his own partial signature.

Generally, trees are hard to reconfigure. In case of a failure (e.g. an internal node in the tree is malicious and does not send anything) it's not sufficient to simply switch the leader. A tree is said to be robust if and only if it all internal nodes and the root node are honest. There is a factorial number of trees but only a small fraction of them is robust. We aim to construct a robust tree to achieve consensus. For this we divide all n nodes into m bins, where m represents the fanout of the tree. Each bin contains at least t nodes, where t corresponds to the size of all internal nodes. If we loop over all the bins and assign the nodes in the bin to cover the internal nodes, we are guaranteed to eventually (within $f + 1$ steps) reach a bin without any faulty node and therefore a tree with a robust configuration. There is a catch though: The above strategy only holds in case $f < m$. For the case where $f \geq m$ we start with the same strategy. In case no bin contained only honest nodes, we fall back to a star topology. In the worst case scenario, in Kauri, we will wait $f + 1 + m$ steps until we end up with a robust star topology.

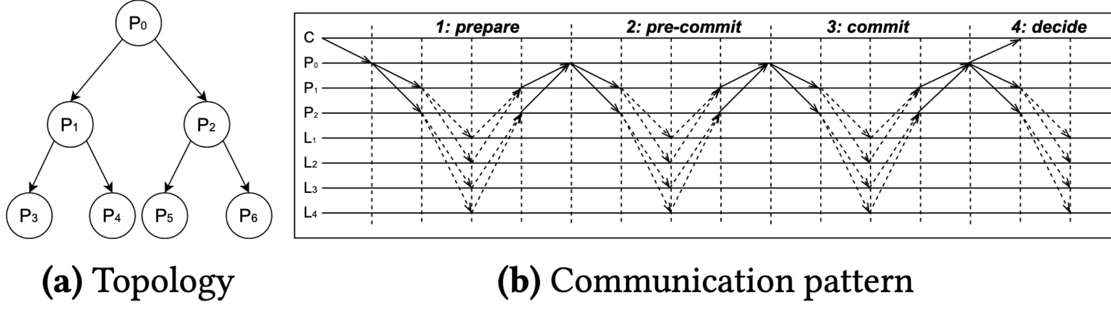

Figure 2.2. Kauri topology and communication pattern with 7 nodes [3]

2.4 BLS Signature Scheme

Boneh-Lynn-Shacham (BLS) is a cryptographic signature scheme [1, 10, 14]. The scheme uses a bilinear pairing function e for the verification of signatures. Signatures are elements of an elliptic curve subgroup. BLS is designed for the usage of different curves and is not restricted to a specific elliptic curve. In our implementation we made use of an elliptic curve called BLS12-381 [15]. The curve was designed in 2017 and used as the foundation for an upgrade on the Zcash protocol [17]. 381 is the number of bits needed to represent coordinates on the curve, whereas the number 12 represents the curves embedding degree. Coordinates of points are elements of a finite field F_q that has prime order q, and q is 381 bits long.

For the signature scheme we make use of the following two subgroups of the BLS12-381 curve:

- $G1 \subset E(F_q)$ where $E : y^2 = x^3 + 4$
- $G2 \subset E'(F_{q^2})$ where $E' : y^2 = x^3 + 4(1 + i)$

G_1 and G_2 are two additive cyclic groups of order r with generators $g_1 \in G_1$ and $g_2 \in G_2$ respectively. The fact that the BLS12-381 actually contains two instead of one curve might be a bit confusing. The reason for this are technical requirements the paring e has to satisfy. G_1 and G_2 need to be distinct groups and must have the same prime-order r. In $E(F_q)$ there is no other subgroup than G_1 that has order r. That's why a subgroup G_2 from a distinct curve $E'(F_{q^2})$ is needed.
It's important to mention that coordinates of points in G_1 are pairs of integers, whereas elements in G_2 are pairs of complex integers. Therefore points in G_2 are more expensive to work with and the take double the amount of memory compared to points in G_1. When dealing with digital signatures, G_1 and G_2 are interchangeable. Thus, we can choose signatures to be elements in G_1 and public keys members of G_2, or the other way around.

The pairing function $e : G_1 \times G_2 \longrightarrow G_T$ that allows for the verification of signatures takes as input elements $P \in G_1$ and $Q \in G_2$ and outputs a point from a (multiplicative) group $G_T \subset F_{q^{12}}$. The pairing e ensures that $e(a \cdot P, b \cdot Q) = e(P, b \cdot Q)^a = e(P, Q)^{a \cdot b} = e(P \cdot a, Q)^b = e(b \cdot P, a \cdot Q)$ holds for scalars a and b.

To generate a Private-Public Key pair, we select a random integer in $[1, r - 1]$ to be the private Key sk. The corresponding public key pk is $pk := sk \cdot g_1$. The discrete logarithm problem ensures that it is unfeasible to recover sk given the public-key pk. To sign a message m we first need to map m onto a point in the group G_2 (assuming that we are using G_2 for signatures). One way to accomplish this, is to make use of a method called "hash and check". This method is not doable in constant time and even attacking issues were mentioned. The IETF standard for hashing to curves [13] adopts a superior

approach that is described in the draft for hashing to elliptic curves [5] . We sketch the approach in detail in Chapter 4. Given a hash function h that maps messages onto an element of G_2, we produce a signature on a message m by computing $\sigma := h(m) \cdot sk$.

Given a message m, a signature σ and a public key pk we want to ensure that σ was created by the holder of the sk that corresponds to pk. This is where the pairing function e comes into play. The signature is valid, if and only if

$$e(g_1, \sigma) = e(pk, h(m))$$

is satisfied.

Compared to ECDSA or Schnorr signatures the scheme benefits of shorter signatures (about half the size).

2.5 BLS Multisignatures

A useful property of the BLS signature scheme is that it allows for signature aggregation [1, 10]. It's either possible to aggregate signatures over different messages or we can also aggregate signatures over the same message. The latter is also referred to as a multisignature [6]. For simplicity we are going to consider the case where n nodes $P_1, .., P_n$ all sign the same message m. The nodes compute partial signatures $\sigma_1, ..., \sigma_n$. To create a multisignature we add up all the signatures $\sigma_i \in G_2$ and the corresponding public keys of all parties:

$$\sigma_{agg} = \sigma_1 + ... + \sigma_n$$
$$pk_{agg} = pk_1 + ... + pk_n$$

To verify that σ_{agg} is valid, one only needs to check if $e(g_1, \sigma_{agg}) = e(pk_{agg}, h(m))$ holds. For the verification of a message that was signed by n nodes we only compute 2 pairings, rather than $2n$ that one would have to do if the n signatures would be verified separately. This is a big advantage since pairings are expensive to compute.

When aggregating signatures over the same message m one is susceptible to the rogue public key attack. In this attack, an attacker Eve chooses $\alpha \longleftarrow \mathbb{Z}_r^+$ and computes $pk_{Eve} = g_1 \cdot \alpha - pk_{Bob}$ with $pk_{Bob} \in G1$ being the public key of some unknowing user Bob. Eve can pretend that Bob and himself signed the multisignature $\sigma = h(m) \cdot \alpha$, because

$$e(g_1, \sigma) = e(g_1, h(m) \cdot \alpha) = e(g_1 \cdot \alpha, h(m)) = e(pk_{Eve} + pk_{Bob}, h(m)).$$

is satisfied. The attacker is able to commit Bob to some message m without Bob ever signing m. There are standard defenses against the public key rogue attack. Prove knowledge of the secret key (PKOSK) resolves the problem, which involves a zero-knowledge proof of knowledge. This way the proving party is able to demonstrate knowledge of the secret key, without revealing any information about the secret key itself.

2.6 Secret Sharing

Secret sharing is key ingredient in threshold cryptography. The process of distributing a secret amongst a group of nodes $\{P_1, ..., P_n\}$ is known as Secret Sharing [9, 7, 8]. Each node possesses a different share of the secret. The original secret can be reconstructed if more than a predetermined threshold k of shares are combined together. A secure secret sharing scheme guarantees that the possession of $k-1$ shares reveals no more information than the possession of 0 shares. Several secret sharing algorithm are available. We use the Shamir's secret sharing method, where a dealer $D \notin \{P_1, ..., P_n\}$ shares a secret key x among $\{P_1, ..., P_n\}$. For this, the dealer chooses a random polynomial $p(X)$ of degree $k-1$, s.t. $p(0) = x$ and generates shares $s_i = p(i)$ for all $i \in \{1, ..., n\}$. Node P_1 gets attributed the share s_1, node P_2 is attributed s_2 and so on. In order to recreate the original secret x at least k parties with a set of

indices I need to collaborate together and do lagrange interpolation. They compute

$$x = p(0) = \sum_{i \in I} \lambda_{0,i}^{I} \cdot s_i$$

where

$$\lambda_{0,i}^{I} = \prod_{j \in I, j \neq i} \frac{j}{j-i}$$

are the lagrange coefficients.

Figure 2.3 depicts an example of Secret Sharing. The y-value of the red point represents the shared secret, and the points in blue represent the shares. Since the polynomial has degree 2, three distinct points on the curve are sufficient to recreate the (unique) polynomial interpolating the three points.

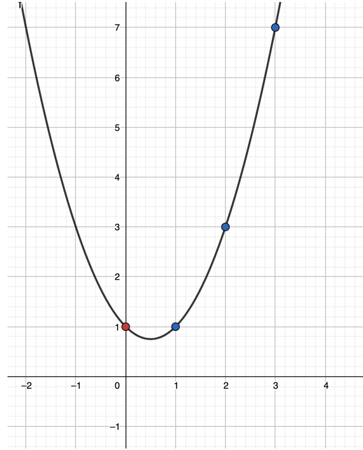

Figure 2.3. An illustration of secret sharing using the polynomial $f(p) = p^2 - p + 1$. The shared secret in red is 1 and the resulting shares in blue are 1,3 and 7.

In case of BLS the dealer chooses a random polynomial $p \in \mathbb{Z}_r^+[X]$, with r being the degree of G_1.

2.7 BLS Threshold Signatures

The BLS scheme allows for a threshold signatures [1]. A secret key $x \longleftarrow [1, r-1]$ (with corresponding public key $u = g_1 \cdot sk_0 \in G_1$) is shared by a central authority among n nodes using a secret sharing method. Each node i is given a private key x_i and a corresponding public-key $u_i = g_1 \cdot x_i$. For the verification of threshold signatures a node can use the public key u of the shared secret x. In order to create a valid threshold signature, at least $k < n$ nodes have to collaborate together. If signing some message m is required, every node P_i that decides to sign m can publish a partial signature σ_i of the from $\sigma_i = h(m) \cdot x_i \in G_2$. For simplicity, lets assume that users $1, ..., k$ participate and generate partial signatures $\sigma_1, ..., \sigma_k$. Anyone can verify that a partial signature σ_i is valid by checking if $e(g_1, \sigma_i) = e(u_i, h(m))$ holds. When all k partial signatures are valid, the complete signature can be recovered as follows:

$$\sigma \longleftarrow \sum_{i=1}^{k} \sigma_i \cdot \lambda_i \in G_2, \text{ where } \lambda_i = \prod_{i=1}^{k, j \neq i} \frac{j}{j-i} \in \mathbb{Z}_r \text{ are the lagrange coefficients.}$$

Every node can verify that σ is indeed a valid signature by using the public-key v of the shared secret. A node verifies that

$$e(g_1, \sigma) = e(v, h(m))$$

is fulfilled.

In case all the t parties behaved honestly and contributed a partial signature on the same message the above equation holds, because

$$\sigma = \sum_{i=1}^{t} \sigma_i \cdot \lambda_i = \sum_{i=1}^{t} (h(m)_i) \cdot \lambda_i = h(m) \cdot \sum_{i=1}^{t} \sigma_i \cdot x_i = h(m) \cdot x$$

holds.

In contrast to Secret Sharing, the secret key x is not put back together and nodes do never disclose their private key. In the HotStuff consensus protocol, upon receiving sufficient valid partial signatures, the leader creates a threshold signature. He then broadcasts the threshold signature to all other nodes in the network. In Kauri we use multisignatures instead of threshold signatures.

Chapter 3

Design

The key goal is to design an experiment where we measure the leader's workload in HotStuff and in Kauri. Initially, we were looking for an existing implementation of the HotStuff consensus protocol. We were able to find a framework [11] written in Go that would fit our needs. The framework does provide an implementation of HotStuff. The framework also contains parts of the BLS signature scheme including the creation and verification of multisignatures based on the bls12-381 elliptic curve. The framework makes use of the bls12381 package [12]. The package provides useful macros that will be helpful during the process of implementing threshold signatures. It contains a pairing function e for the verification of signatures, point addition in G_1 and G_2 and other useful tools. Parts of the frameworks implementation do not follow the original descripton of the BLS paper [1]. However, since Kauri can be seen as an extension of HotStuff, we decided to work with the existing framework and make some important changes.

3.1 Secure threshold scheme

In the HotStuff framework, a designated leader of a round transmits a message m to all the other nodes in the network. Eventually, the leader will receive partial signatures from several replicas. The leader then verifies that the threshold k of valid partial signatures σ_i is met. The framework is indeed using BLS, but it does employ multisignatures, not threshold signatures. Moreover the implemented multisignature scheme is not secure because there is no check for knowledge of the secret key. Hence, we implemented BLS threshold signatures according to the original BLS paper.

Signatures are points in G_2 and public keys are elements in G_1. Thus, public keys are small and fast to work with compared to signatures in G_2.

3.2 Star and Tree Topologies

In this section we explain how we simulate and measure the leader's workload in HotStuff and in Kauri. For the simulation we employ the BLS signature scheme.

3.2.1 Simulate one phase in HotStuff

We benchmark two different scenarios in HotStuff [2]. A first scenario where the leader creates a QC in form of a threshold signature and a second scenario where the leader deploys multisignatures. In both settings the procedure is similar. We can configurate the number of nodes n that are present in the scenario and set the number of phases that shall be simulated. Since HotStuff itself requires that $f < n/3$, the threshold k is set to $k = n - f$, where $f = \lceil \frac{n-1}{3} \rceil$. In every phase the nodes sign a different message m than in the previous phases. We are ensuring this by changing random bits of the message m_{prev} from the previous phase.

We create a script, where in each phase all n nodes produce a (valid) partial signature σ_i on some message

m. We start measuring time at the point where the leader starts to verify the signatures. For simplicity the leader only verifies the first k signatures σ_i for $i \in [1, k]$. He then produces a (valid) QC either in form of a threshold signatures or a multisignature. After the creation of a QC we stop measuring the time. The networking part, where a leader broadcasts the QC to all other nodes is omitted in the measurements. Furthermore we make other strong assumptions including that there is a check for KOSK when working with multisignatures and assume that all nodes have sent a partial signature. We do not measure HotStuff's reconfiguration time.

3.2.2 Simulate one phase in Kauri

Regarding Kauri, the deployed communication abstraction in form of a tree makes it tedious to work with threshold signatures [3]. For the simulation of a phase in Kauri we can set the depth d of the tree. We work with a binary tree that has fanout 2. A complete binary tree of depth d contains $2^d - 1$ nodes with 2^{d-1} of them being leaf nodes. A multisignature has to meet a threshold $k = n - f$ in order to be valid. We make the assumption that all n nodes (except the leader) sign some message m to produce a partial signature σ_i. Thus, the final multisignature is composed out of $n - 1$ (valid) partial signatures. Every node receiving a signature will still go through the process of validating the signature though. We start at the bottom of the tree, where every leaf node (sitting on level d) signs a message m and computes a partial signature. Their parent node (sitting on level $d - 1$) verifies the signatures from both his children, signs the message m himself and produces a multisignature. Nodes that are part of of the level $d - 2$ will not receive two partial signatures from their children, but two signatures that are already aggregated. Therefore, we implemented a function that takes as an input an array of already aggregated signatures and outputs a multisignature containing a list of all participants and some Point in G_2 that represents the signature. As soon as level $d = 1$ attained, the leader will receive two multisignatures from his two children. That's when we stop the time. The leader checks for correctness of the multisignatures and makes sure that the threshold k is met. He then assembles the two signatures together to create the QC in the form of a multisignature. That's when we stop measuring the the time. This way we are able to estimate the workload for a leader in one phase of Kauri. We make the assumpumtion that there is a check for KOSK and all leaf nodes have sent a partial signature and a all internal nodes a multisignature to their parent node. We are not measuring the protocol's reconfiguration time.

Chapter 4

Implementation

In this chapter, we give a technical insight to all the most important functions of the BLS signature scheme.

4.1 Key Generation

The BLS scheme uses cyclic subgroups of G_1 and G_2 of prime-order r [1]. We denote with g_1 and g_2 the generators of G_1, respectively G_2. All nodes in the network have knowledge about these parameters. In the actual implementation we make use of a cryptographically secure random number generator. The $KeyGen$ function takes as inputs a threshold parameter k and a integer n that represents the number of nodes in the network. It outputs a set that contains n private key shares $sk_1, ..., sk_n$, a set with the corresponding public keys $pk_1, ..., pk_n$ and the public key pk_0 of the shared secret sk_0.

Algorithm 1 Key Generation

1: **procedure** KEYGEN(k, n)
2: **if** $k > n$ **then**
3: **return** invalid threshold parameter
4: $sk_0 \leftarrow \mathbb{Z}_r^+$ // r is the order of G_1 and G_2
5: $pk_0 \leftarrow sk_0 \cdot g_1 \in G_1$
6: $f \leftarrow$ select a random polynomial of degree $k-1$, s.t $f(0) = sk_0$
7: **for** $i = 1, ..., n$ **do**
8: $sk_i \leftarrow f(i)$
9: $pk_i \leftarrow sk_i \cdot g_1$
10: $Sk \leftarrow \{sk_1, ..., sk_n\}$
11: $Pk \leftarrow \{pk_1, ..., pk_n\}$
12: **return** Sk, Pk, pk_0

The algorithm first checks if the threshold parameter k does not exceed n. Otherwise it is not possible to create valid threshold signatures. After assigning the secret sk_0 a random number in $[1, r-1]$ and computing the associated public key pk_0 (pk_0 is needed for verification purposes of threshold signatures), the secret sk_0 is shared among n nodes. For distribution of the sk_0 we make use of Shamir's secret sharing method. Each node i gets assigned a private-public key pair sk_i and pk_i.

4.2 Signing and Verification

Upon receiving a message m from the leader, a node i first hashes m into a bit-string denoted by m'. He feeds m to the SHA-256 hash function h_1, calculating $m' = h_1(m)$. This results in a bit-string of length 256.

To produce a digital signature on m', mapping m' onto a point in G_2 is required. We shall call the function mapping bit-strings onto Points in G_2 $hashToCurve$. There exist several ways to accomplish this, with different levels of security and efficiency [10]. In the bls12381 library there's an implementation of such a hash function satisfying the IETF standards for hashing onto elliptic curves [13]. A broad description of the procedure looks as follows: We start by hashing $m' \, mod \, q$ to create a point $s \in F_q$. It ensures that the resulting output is distributed uniformly. We then use a special function, called the SWU map, that guarantees to convert the field point s into an elliptic curve point p. The point p is not part of $E'(F_{q^2})$ though but is a point on some other elliptic curve that is isogenous to $E'(F_{q^2})$. We then translate p to some point $r \in E'(F_{q^2})$ by using another function (3-isogeny). We are not quite done yet, since r is a point in $E'(F_{q^2})$, but what we want to end up with is a point in the subgroup $G_2 \subset E'(F_{q^2})$. In a last step, cofactor clearing is used to map $r \in E'(F_{q^2})$ to an element in G_2. In other words, the function $hashToCurve$ is a composition of several functions [5]. The same procedure can be applied to map bit-strings to G_1. This is necessary if we want signatures to be in G_1.

The Signing method described in Algortihm 2 takes as inputs a message m and the private key sk_i of some node i. The method outputs a signature structure σ that contains the signature itself and in addition to that the nodes index i. The index is needed for verification purpose to reveal the signers identity in order to access his public key.

Algorithm 2 Signing

1: **procedure** SIGN(m, sk_i)
2: $\quad m' \leftarrow h_1(m)$ // m' is a 256-bit string
3: $\quad c \leftarrow hashToCurve(m') \in G_2$
4: $\quad s \leftarrow c \cdot sk_i \in G_2$
5: $\quad \sigma \leftarrow \{Signer : \, i, \, Signature : \, s\}$
6: $\quad$ **return** σ

A time-consuming part of the Signing algorithm is when $c \in G_2$ is multiplied with the secret-key scalar sk_i. Multiplication of a Point $P \in G_2$ with a scalar n can be regarded as the repeated addition of Points in G_2, i.e. $n \cdot P = \underbrace{P + P + ... + P}_{n \, times}$. Since simple addition of the same point P for large n can be really time-consuming, there are several techniques that speed up the computation time of scalar multiplication on some elliptic curves. In our implementation we use a variant of the Gallant-Lambert-Vanstone (GLV) method [16][4], which we are not going to discuss here. Scalar multiplication in G_1 and especially in G_2 remain expensive though.

For the verification that a signature σ on some message m is valid, we transform the message into a 256-bit string and hash the string onto a point $c \in G_2$. Using the public-key pk of the signing node i, we check if the pairings $e(g_2, \, \sigma.Signature)$ and $e(pk, c)$ have the same value. If they do then the verifying node can be sure that it's a valid signature signed by node i. If not, the verification method will output "false".

Algorithm 3 Verification

1: **procedure** VERIFY(σ, m)
2: $\quad m' \leftarrow h_1(m')$
3: $\quad c \leftarrow hashToCurve(m') \in G_2$
4: $\quad pk = \sigma.Signer.PublicKey \in G_1$
5: $\quad$ **return** $e(g_1, \, \sigma.Signature) \stackrel{?}{=} e(pk, c)$

In line 5 of the Verification algorithm, two pairings e need to be evaluated to check if $e(g_1, \, \sigma.Signature) = e(pk, c)$ holds. Calculation of pairings consist of two parts. One of them being the so-called Miller loop and the other one final exponentiation [10]. They are expensive in terms of

computational costs, but we make use of a trick to reduce the costs of final exponentiation. We denote with e' the pairing function before applying the final exponentiation. Normally we would now check if $e'(g_1, \sigma.Signature)^x = e'(pk, c)^x$ for some large number $x = (q^k - 1)/r$. Instead of computing two Miller Loops and additionally two final exponentiations, we use the fact that G_T is a multiplicative group, reorganise the equation and check if $(e'(-g_1, Signature) \cdot e'(pk, c))^x = 1$ is fulfilled. We could negate any of the four inputs of e'. Due to properties of pairings, negating an input value is equivalent of taking the inverse of an element in G_T. Using this technique we get rid of one out of two final exponentiations. The implementation in the bls12381 library follows exactly the same procedure as above.

4.3 Multisignatures and Threshold Signatures Creation

A multisignature is a structure composed out of a set of participants and the signature itself which is a point in G_2. The method described in Algorithm 4 takes as input a set of partial signatures.

Algorithm 4 Create Multisignature

```
 1: procedure MULTISIGNATURE(signatures)
 2:     if length(signatures) == 0 then
 3:         return nul
 4:     s_agg ← O   // point at infinity (neutral element in G_2)
 5:     participants ← {}
 6:     for σ ∈ signatures do
 7:         s_agg ← s_agg + σ.Signature
 8:         append σ.Signer to participants
 9:     σ_agg ← {Participants : participants, Signature : s_agg}
10:     return σ_agg
```

In case the input is not empty, the algorithm loops over all the partial signatures and follows the described procedure in Section 2.5 to form a multisignature. The variable participants is a bitfield structure made up of an array of bytes and an integer that represents the number of nodes that participated in the creation of the multisignature. To check if a node i is a participant, we simply check if the bit at i-1 is set to 1. The knowledge about the number of participants is important when it comes to the question if the multisignature contains a quorum of certificates. Furthermore a list of all participants is needed to build a public key for the purpose of verification.

In Kauri all the internal nodes and especially also the leader will receive two multisignatures and have to form another multisignature. We do not show the pseudocode for that functionality here, but the idea shares a lot of similarities with Algorithm 4. In addition to that, instead of verifying if the length of the input is equal to 0, the leader verifies if the length of the signatures reach the threshold k before computing a quorum certificate.

The procedure to create a threshold signature is different. After verifying every single partial signature using Algorithm 3, in HotStuff the leader calls the method Threshold Signatures and inputs a set of (distinct) partial signatures. He outputs a single threshold signature $\sigma_{threshold}$. Identities of the participants are not revealed by $\sigma_{threshold}$. The function in Algorithm 5 describes the creation of a threshold signature.

First, we need to ensure that the threshold of k partial signatures is met. We then create a set of indies, containing all indices i of the nodes that have sent a valid partial signature to the leader. This set is needed when it comes to the part where we compute the lagrange coefficients λ_i for all $i \in I$. Because of r being prime, the multiplicative inverse of $(j - i)$ in the ring $\mathbb{Z}_r$ (often written $\mathbb{Z}/r\mathbb{Z}$ or $\mathbb{F}_r$) can be computed with the extended euclidean algorithm. An implementation of the algorithm can be found inside Golangs math library. The calculation of $\sigma_{threshold}$ follows strictly the procedure from section 2.7.

Algorithm 5 Create Threshold Signature

1: **procedure** THRESHOLDSIGNATURE($signatures$)
2: **if** $length(signatures) < k$ **then**
3: **return** threshold barrier of at least k signatures is not met
4: $I \leftarrow \{\}$
5: **for** $partialSignature \in signatures$ **do**
6: $I \leftarrow I \cup \{partialSignature.Signer\}$
7: $\sigma_{threshold} \leftarrow O$ // point at infinity (neutral element in G_2)
8: **for** $partialSignature \in signatures$ **do**
9: $i \leftarrow partialSignature.Signer$
10: $\lambda_i \leftarrow \prod_{j \in I \setminus \{i\}} \frac{j}{j-i} mod\, r \in \mathbb{Z}_r$
11: $\sigma_{threshold} \leftarrow \sigma_{threshold} + \lambda_i \cdot partialSignature.Signature$
12: **return** $\sigma_{threshold}$

4.4 Multisignature and Threshold Signature Verification

The Verification of a multisignature σ_{agg} on some message m described in Algorithm 6 is similar to the verification of a single signature. The difference is that we add up all public keys $pk_i \in G_1$ of nodes i that participated in the creation of σ_{agg} before verifying σ_{agg}.

Algorithm 6 Multisignature Verification

1: **procedure** VERIFYMULTISIGNATURE(σ_{agg}, m)
2: $m' \leftarrow h_1(m')$
3: $c \leftarrow hashToCurve(m') \in G_2$
4: $pk_{agg} \leftarrow O$ // point at infinity (neutral element in G_1)
5: **for** $p \in \sigma_{agg}.Participants$ **do**
6: $pk_{agg} \leftarrow pk_{agg} + p.PublicKey$
7: **return** $e(g_1,\, \sigma_{agg}.Signature) \stackrel{?}{=} e(pk_{agg}, c)$

It's important to make a remark about deploying multisignatures in Kauri. Upon receiving a quorum certificate in form of a multisignature from the leader, a node is not only checking wether $e(g_1,\, \sigma_{agg}.Signature) \stackrel{?}{=} e(pk_{agg}, c)$ holds, but also if at least k different nodes did contribute in the creation of σ_{agg}.

Verifying if a threshold signature $\sigma_{threshold}$ on a message m is valid is easier. Any node can call the function $VerifythresholdSignature$ with inputs $\sigma_{threshold}$ and a message m.

Algorithm 7 Threshold Signature Verification

1: **procedure** VERIFYTHRESHOLDSIGNATURE($\sigma_{threshold}$, m)
2: $m' \leftarrow h_1(m')$
3: $c \leftarrow hashToCurve(m') \in G_2$
4: **return** $e(g_1,\, \sigma_{threshold}) \stackrel{?}{=} e(pk_0, c)$

The signature $\sigma_{threshold}$ can be verified using the public key pk_0 of the shared secret sk_0, because a valid threshold signature on m is of the form $\sigma_{threshold} = hashToCurve(m') \cdot sk_0$.

Chapter 5

Results

In this chapter, we compare the leader's computational effort when either using a tree or a star communication topology. Moreover we discuss the reasons behind the results and state possible improvements. We conducted our experiments on a MacBook Pro with an Intel Core i5 CPU and 16GB of RAM.

5.1 Base Performance

We measure the leader's workload in three different scenarios. A first scenario reproduces a phase when deploying Kauri's tree communication pattern. In Kauri the leader forms a Quorum Certificate (QC) in form of a multisignature. In Scenario2 and Scenario3 we aim to simulate a phase in the HotStuff Consensus Protocol. Thus, we reproduce a star communication pattern where the leader forms a QC. This QC has once the form of a multisignature and once the form of a threshold signature. For all three scenarios we simulated 100 phases with different blocks. We are interested in measuring the average time it takes the leader to execute one phase. We simulate networks consisting of 7, 15, 31 and 63 nodes with the respective threshold parameter k being 5, 10, 21 and 42.

Scenario	Topology	$n = 7$	$n = 15$	$n = 31$	$n = 63$	Possible in
Scenario1	**tree (multisig)**	3.9	5.6	8.7	15.2	Kauri
Scenario2	**star (multisig)**	6.7	12.7	26.0	52.4	HotStuff
Scenario3	**star (threshold)**	6.8	13.3	27.8	58.7	HotStuff

Table 5.1. Computational expense for the leader deployed in different communication topologies. Time is measured in ms and represents the average time for one phase.

A few interesting things can be extracted from Table 5.1. In Scenario2 and Scenario3, the leader's workload grows approximately linear by a factor of 2 with respect to the amount of nodes in the network. In Hotstuff's star topology the leader verifies k partial signatures and creates either a multisignature (Scenario2) or a threshold signature (Scenario3). Thus, for verifying all partial signatures, the leader needs to evaluate $2k$ pairings. For the creation of a multisignature signatures the leader computes k additions of signatures in G_2. In Scenario1 the leader only verifies two multisignatures before combining them and output a new multisignature himself. The verification process of both multisignatures involves the evaluation of four pairings and a total of $n - 1$ public key additions in G_1. The computation of all parings is the most expensive part when using the BLS signature scheme. In Scenario1, the leader benefits of the fact that no matter the size of the network the amount of pairings to compute remains four. This leads to a comparable small workload for the leader in Scenario1 compared to Scenario2 and Scenario3. This fact gets especially visible for bigger networks.

There is a small gap in terms of time difference between Scenario2 and Scenario3 where the leader follows the HotStuff consensus protocol. Table 5.1 shows us that the creation of a threshold signature

takes more time than the creation of a multisignature. The main reason for this is the computational effort when multiplying the lagrange coefficients with the partial signatures in order to create a threshold signature. The overall time spent on the computation of this product can be extracted from Table 5.2. This small extra effort when creating threshold signatures pays of when it comes to the verification of the signature. Table 5.3 shows the average time for the verification of a multisignature and a threshold signature. While verification costs stay the same when dealing with threshold signatures, no matter the amount of nodes present in the network, it does not with multisignatures. Verification of a multisignature involves hashing a sha256 bit-string to an element in G_2 (this costs don't change no matter the size of the network), but also the addition of k public keys in G_1. The latter is the reason why multisignatures get more expensive the bigger the network.

operation	$n=7$	$n=15$	$n=31$	$n=63$
evaluation of 2k pairings	4.9	9.4	19.5	39.7
hashing onto curve (k times)	1.6	3.1	6.5	13.3
mul. of lagrange coeffs. with par. Signatures	0.2	0.5	1.1	3.0

Table 5.2. Time taken (in ms) for different operations needed for creating threshold signatures.

Table 5.2 shows the most expensive operations in terms of time when using the BLS signature scheme for creating threshold signatures.

operation	$n=7$	$n=15$	$n=31$	$n=63$
verification multisignature	2.2	3.1	5.3	9.6
verification threshold signature	1.3	1.2	1.3	1.2

Table 5.3. Time (in ms) needed for the verification of multisignatures and threshold signatures.

5.2 Swapping G_1 and G_2

An important change which we did not introduce initially, is to swap G_1 and G_2. Signatures are now elements in G_1 and public keys are elements in G_2. This should most probably leave to a decrease in terms of computational costs when multiplying partial signatures with scalars (e.g. threshold coefficients), since arithmetic operations in G_1 are cheaper than in G_2. The addition of public keys is now more expensive.

Scenario	Topology	$n=7$	$n=15$	$n=31$	$n=63$	Possible in
Scenario1	**tree (multisig)**	3.6 (-7.7%)	5.4 (-3.6%)	8.8 (+1.1%)	15.9 (+4.6%)	Kauri
Scenario2	**star (multisig)**	5.9 (-11.9%)	12.0 (-4.8%)	25.6 (-1.5%)	50.8 (-3.1%)	HotStuff
Scenario3	**star (threshold)**	5.8 (-14.7%)	12.6 (-5.3%)	27.8 ($\pm$0%)	55.3 (-5.8%)	HotStuff

Table 5.4. Average of the leaders workload when being deployed in the HotStuff and Kauri communication pattern, when signatures are in G_1 and public keys in G_2. Time is measured in ms and represents the average time for one phase. Number in parentheses corresponds to the difference from the results obtained in Table 5.1.

Table 5.4 summarizes the leaders workload for the three scenarios with signatures being points in G_1 and public keys are elements of G_2. Comparing these results with the results obtained in Table 5.1, we do not notice a drastic change in performance.

Table 5.5 shows the most expensive operations when dealing with the creation of threshold signatures. Hashing arbitrary sha256 bit-strings to points in G_1 is cheaper compared to G_2. This is proven when comparing Table 5.2 and Table 5.5. Furthermore, we observe that the total costs of multiplying signatures in G_1 with the lagrange coefficients is cheaper than doing the same in G_2.

operation	$n = 7$	$n = 15$	$n = 31$	$n = 63$
evaluation of 2k pairings	4.8	10.2	22.3	42.8
hashing onto curve (k times)	0.9	1.8	3.9	7.7
mul. of lagrange coeffs. with par. Signatures	0.2	0.4	0.9	1.9

Table 5.5. Time taken (in ms) for different operations needed for creating threshold signatures in G_1.

operation	$n = 7$	$n = 15$	$n = 31$	$n = 63$
verification multisignature	2.0	3.3	5.6	10.1
verification threshold signature	1.1	1.3	1.2	1.3

Table 5.6. Time (in ms) needed for the verification of multisignatures and threshold signatures in G_1

We can not see that changing the order of G1 and G2 has a drastic impact on the performance in the three scenarios. This is the real efficiency of our model.

5.3 The importance to reduce the lagrange coefficients

In practice it is not realistic to not reduce the lagrange coefficients modulo r or, equivalently, to choose a subgroups order r which is too big. Despite the fact of not being very realistic, we still tried it for the three scenarios because it shows how efficiency changes if one chooses a field with longer representation. It worsens the efficiency when working with threshold signatures as can be seen in Table 5.7.

Scenario	Topology	$n = 7$	$n = 15$	$n = 31$	$n = 63$	Possible in
Scenario1	**tree (multisig)**	3.9	5.7	8.5	15.0	Kauri
Scenario2	**star (multisig)**	6.7	12.9	27.0	51.7	HotStuff
Scenario3	**star (threshold)**	11.9	46.0	185.8	797.0	HotStuff

Table 5.7. Computational expense for the leader deployed in different communication topologies, when the lagrange coefficients are not reduced modulo r or r is too big. Time is measured in ms and represents the average time for one phase. Signatures are in G_2.

Scenario	Topology	$n = 7$	$n = 15$	$n = 31$	$n = 63$	Possible in
Scenario1	**tree (multisig)**	3.3	4.9	8.0	14.3	Kauri
Scenario2	**star (multisig)**	5.4	11.3	22.8	45.7	HotStuff
Scenario3	**star (threshold)**	8.0	25.6	107.0	461.1	HotStuff

Table 5.8. Computational expense for the leader deployed in different communication topologies, when lagrange coefficients are not reduced modulo r or r is too big. Time is measured in ms and represents the average time for one phase. Signatures are in G_1.

Scenario1 and Scenario2 from Table 5.7 and 5.8 do not differ by a lot from the results obtained in Table 5.1 and Table 5.4. But there's a drastic change in the performance when the leader deploys threshold signatures. Furthermore, we extract the proof that it's up to 50 percent cheaper if signatures are elements in G_1 rather than G_2 when dealing with the creation of threshold signatures. One of the reasons behind this is shown in Table 5.9. If we do not reduce the lagrange coefficients by mod r or r is too big and the threshold k is large, the lagrange coefficients get larger and larger. Therefore the resulting multiplication of the (large) lagrange coefficients with the partial signatures is expensive compared to the multiplication with small coefficients. With big lagrange coefficients the difference when dealing with partial signatures in G_1 and G_2 can not be overseen. When using multisignatures, we do not expect a

big change since creation of an multisignatures involves k additions of partial signatures (either in G_2 or G_1) and verification involves k additions of public keys (either in G_1 or G_2).

operation	$n=7$	$n=15$	$n=31$	$n=63$
multiplication (in G_2)	5.7	32.2	158.7	742.3
multiplication (in G_1)	2.6	14.3	82.3	412.5

Table 5.9. Time taken for the multiplication of partial signatures with lagrange coefficients when using threshold signatures. Time is measured in ms and represents the average time for one phase.

Chapter 6

Conclusion

Consensus protocols are key components in today's blockchain technology. The HotStuff consensus protocol and Kauri both make use of digital signatures. We have implemented the BLS signature scheme based on the elliptic curve BLS12-381. We made use of the signature scheme to evaluate the leader's computational effort in HotStuff and in Kauri.

Kauri's tree topology aims to improve system's scalability and load balancing, which is a problem in HotStuff. Our results confirm the fact that Kauri is able to improve on both these issues. In a network consisting of 63 nodes, in Kauri the leader will spend $15.2ms$ on average per phase, whereas in HotStuff the leader spends $52.4ms$ per phase. When working with threshold signatures in HotStuff, the time needed for the leader per phase is even slightly higher ($58.7ms$). We furthermore present results that may serve as a proof that the most expensive operations for the leader in the BLS scheme (based on BLS12-381), when using threshold signatures, are the evaluation of pairings and hashing bit-strings to the elliptic curve. We observed that the leader's performance increases in HotStuff, if signatures are in G_1 and public keys in G_2, and not the other way around.

However, our findings have to be considered carefully since we did not measure the network's transmission delay that occurs in real deployments of HotStuff and Kauri. Furthermore, we made the strong assumption that all nodes would compute and send (correct) signatures to the leader. We are therefore not taking into account the time needed for the reconfiguration of the protocol. Further improvements may include the implementation of a reconfiguration protocol and therefore move towards a more realistic scenario. Another important change would be to implement a mechanism that verifies knowledge of the secret key (KOSK) when using multisignatures. KOSK offers protection against the rogue public key attack.

Bibliography

[1] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[2] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[3] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 35–48, 2021.

[4] Steven D Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of cryptology*, 24(3):446–469, 2011.

[5] Riad S Wahby and Dan Boneh. Fast and simple constant-time hashing to the bls12-381 elliptic curve. *Cryptology ePrint Archive*, 2019.

[6] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.

[7] Christian Cachin. Distributed cryptography. *Cryptographic Protocols, FS 2021*.

[8] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[9] Shamir's secret sharing. Available at: `https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing`.

[10] Edgington Ben. Bls12-381 for the rest of us. 2022. Available at: `https://hackmd.io/@benjaminion/bls12-381#BLS12-381-For-The-Rest-Of-Us`.

[11] HotStuff. Available at: `https://github.com/relab/hotstuff`.

[12] BLS12-381. Available at: `https://pkg.go.dev/github.com/ethereum/go-ethereum/crypto/bls12381`.

[13] IETF standards: Hashing to elliptic curves. Available at: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16`.

[14] BLS digital signature. Available at: `https://en.wikipedia.org/wiki/BLS_digital_signature`.

[15] BLS12-381: New zk-snark elliptic curve construction. Available at: `https://electriccoin.co/blog/new-snark-curve/`.

[16] Armando Faz-Hernández, Patrick Longa, and Ana H Sánchez. Efficient and secure algorithms for glv-based scalar multiplication and their implementation on glv–gls curves (extended version). *Journal of Cryptographic Engineering*, 5(1):31–52, 2015.

[17] Zcash. Available at: https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md.

[18] Permissioned Blockchain. Available at: https://www.investopedia.com/terms/p/permissioned-blockchains.asp.

[19] Byzantine Fault Tolerance. Available at: https://www.fool.com/investing/stock-market/market-sectors/financials/cryptocurrency-stocks/byzantine-fault-tolerance/#:~:text=Byzantine%20Fault%20Tolerance%20is%20a,a%20group%20of%20Byzantine%20generals.

[20] Byzantine Fault Tolerance in Blockchain . Available at: https://www.naukri.com/learning/articles/byzantine-fault-tolerance-in-blockchain/#s2.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Aarau, 23.11.2022

Ort/Datum

Unterschrift