VILNIUS UNIVERSITY
MATHEMATICS AND INFORMATICS FACULTY
SOFTWARE ENGINEERING DEPARTMENT

KZG10 Polynomial Commitments in Blockchain Technologies

KZG10 polinomais paremti įsipareigojimai blokų grandinių technologijose

Bachelor Work

Author: 4th course 5th group student
Valdas Rakutis (signature)

Supervisor: partn. doc. Saulius Grigaitis (signature)

Reviewer: dr. Karolis Petrauskas (signature)

Vilnius
2021

Santrauka

Atsiradus „Bitcoin“ kriptovaliutai, blokų grandinių tecnologijos gan greitai išpopuliarėjo. Technologijos naudojamos „Bitcoin“ tinkle turi praktinių trūkumų. Šie trūkumai yra sprendžiami kituose blokų grandinių technologijomis paremtuose protokoluose, pavyzdžiui, „Ethereum“ tinkle. Viena iš šių technologijų problemų yra tinklo mazgų skleidžiamos informacijos teisingumo patvirtinimas. Ši problema sprendžiama įvairiais būdas ir vienas iš jų – kriptografinių įsipareigojimų schemos. Šios schemos jau naudojamos egzistuojančiose technologijose, tačiau „Ethereum“ protokole siūloma pakeisti egzistuojančią schemą į schemą aprašytą KZG10 ir FK20 darbuose. Šios schemos įgyvendinimai, pritaikyti „Ethereum“ protokolui, jau egzistuoja, tačiau šie įgyvendinimai yra nepilni arba įgyvendinti ne su sparčiomis programavimo kalbomis. Šiame darbe aprašoma greičiausia KZG10 schemą įgyvendinanti biblioteka. Palyginus su egzistuojančiu sprendimu parašytu „Ethereum Foundation“ tyrėjų, įgyvendinimas aprašytas šiame darbe pasiekia bent 37% didesnį spartumą (laiko atžvilgiu), lyginant scenarijus planuojamus naudoti „Ethereum“ tinkle.

Raktiniai žodžiai: blokų grandinės, Ethereum, polinomais paremti įsipareigojimai, eliptinės kreivės, kriptografija

Summary

Since the introduction of ”Bitcoin” blockchain technologies have been on the rise and have even entered the mainstream. The design of the ”Bitcoin” protocol has some practical flaws. These flaws are being solved in other blockchains such as the ”Ethereum” network. One of such problems is ensuring that nodes of a network are following the protocol and are not providing invalid data. One of the solutions for this problem are cryptographic commitment schemes. These schemes are already used, but one commitment scheme described in KZG10 is proposed to replace the current schemes in the ”Ethereum” network. There are already implementations of this scheme, but they are not full or are not implemented in performance-oriented programming languages. This work focuses primarily on implementing the fastest commitment scheme library described in the KZG10 and FK20 papers. When compared to the implementation using the Go language done by ”Ethereum Foundation” researchers, this implementation achieves a minimum of 37% performance (time wise) increase in scenarios which are proposed to be used in the ”Ethereum” network.

Keywords: blockchain, Ethereum, polynomial commitments, elliptic curves, cryptography

CONTENTS

Introduction

In the last 10 years blockchain technology use has been rising and entering the mainstream. It can be best seen by looking at crypto-currencies which use this technology. According to ”Investopedia”, the ”bitcoin” crypto-currency started off at a price of 0.0008 to 0.08 US dollars [Inv21]. According to a live crypto-currency tracking service ”coinmarketcap”, currently the two most expensive crypto-currencies - ”bitcoin” and ”ether” cost about 38 thousand and 2.6 thousand US dollars respectively. The total value of only these two crypto-currencies is already over one billion US dollars, while all of the crypto-currencies combined have a combined cap of 1.63 trillion US dollars [Coi21]. Such a huge rise in price over a relatively short time span shows a clear growth trend for such technologies.

Most blockchain networks are decentralised and have a variety of problems due to this reason. Due to there not being a source of truth, consensus algorithms need to exist, which may be abused by bad actors. For example, if the consensus algorithm is simply picking the longest chain, then a 51% attack may be possible. These attacks have cost millions in recent history [SM19]. Many attack vectors is one of the reasons why blockchain technologies are constantly evolving.

An other serious problem is an environmental one. Proof of work based blockchains have a resource intensive design [Tru18], and any way to reduce the energy-footprint can be helpful both for the environment and the public image of blockchain technologies [SBF+20; Tru18]. For example, the ”Ethereum” network is trying to move away from an energy intensive proof of work consensus mechanism to a less energy intensive proof of stake consensus mechanism [SBF+20]. The outlined problems show, that even though there is wide-spread adoption of blockchain based technologies via crypto-currencies, there are still problems that need to be solved.

”Ethereum” network is one of the industry leading crypto-currency blockchain networks. The group now behind the maintenance and further development of this network is called - ”Ethereum Foundation” [Fou21]. This group is constantly trying to update the ”Ethereum” protocol, releasing various white-papers, trying to think of ways to solve existing problems within blockchain technologies.

Part of this work is implementing the KZG10 scheme, as the ”Ethereum Foundation” members have been prototyping and investigating its use to replace state roots or as a method enabling data availability sampling [But20; But21a; KZG10]. The exact scheme is still being actively discussed [But20], but there already are a few implementations, specifically as a proof of concept for the ”Ethereum” network. At least at the start of this research the only implementation which also includes functions required by data availability sampling is written is Go, by ”Ethereum Foundation” researchers.

There is at least one clear problem to be solved with the Go implementation. The implementation depends on a elliptic curve algebra library which implements some of the needed primitive calculation functions. The ”Ethereum Foundation” researchers who wrote the implementation tried using multiple calculation libraries to find the one that performs best, although, not all of them support all needed operations as can be seen from the benchmark results [Pro21b]. The authors’ hypothesis is that they would be able to produce a faster implementation of the commitment scheme,

via the use of a performance-oriented language, focusing on the functions required to enable data availability sampling in the ”Ethererum” network. This implementation could be used for further prototyping by ”Ethereum Foundation” or other researchers.

The way in which the author seeks to achieve this a faster implementation is via a language more suitable for high performance applications. The language of choice is Rust, as it has a zero overhead cost for calling C APIs [Cri15], which is what the underlying elliptic curve algebra library - Herumi MCL has to be used through. The language Go, on the other hand, has a large overhead for calling C APIs [All18]. Additionally, depending on the algorithm Rust can be up to a thousand times faster than Go [Mul20]. The author argues, that these two things in conjunction may help to implement the most performant existing KZG10 implementation which supports data availability sampling.

Even though the main goal is to implement the most performant library, the author is very familiar with the C# programming anguage, while being very unfamiliar with the Rust language and development environments. In order to eliminate one of the unknowns (language), the author seeks to implement a minimal KZG10 scheme in C#.

The implemented library should also be verified to be working correctly. This can be done formally, but as this is an exploratory implementation, the efforts to verify the library formally would be too large. Instead, the author should port (re-implement them) all applicable tests from the go-kzg library and add randomised tests, helping to verify that the library works properly in an informal way. This would ensure that the implemented library is at least as correct as go-kzg.

As the KZG10 polynomial commitment scheme is a complex mathematical construct involving elliptic curve cryptography, part of this work is analysis of what the KZG10 polynomial commitment scheme is and how it works [KZG10]. Implementing the scheme without understanding it is practically impossible, or the final result would not be the most performant KZG10 implementation.

The KZG10 paper is fairly recent and there are not a lot of extensions or explanations of this paper. Therefore the main literature sources are going to be the KZG10 paper itself and some works surrounding the extension of this scheme with additional features, i.e. - data availability sampling.

1 Commitment Schemes

In this context a commitment is a construct where a prover has to provide a value (the commitment) which in a sense locks the value used to build the commitment. Then, once the verifier gets hold of the commitment, the prover provides the value used. That way the verifier is ensured, that the value used to generate the commitment was the one that was provided to them. To be more explicit, any commitment scheme has two stages [Sma+03a]:

1. commit stage
2. reveal stage

During the commit stage, the committer (prover) generates a commitment using a method defined by the given commitment scheme. Essentially a commitment c for a value v is the result of some function $c = f(v)$. Then the prover sends the commitment to all of the participants of the scheme [Sma+03a].

During the reveal stage the committer (prover) reveals the value v to all of the participants and they can check, that $c = f(v)$ is satisfied [Sma+03a].

1.1 Properties

One of the main properties of a commitment is that having the value of any given commitment, one would not be able to reconstruct the value used to generate the commitment in a trivial way. In more mathematical terms, a commitment c to a value v could be generated by some function $f(v) = c$, which does not have a trivial inverse function $g(c) = v$. This property is further referred to hiding [CF13].

A simple example of this could be a commitment for $n \geq 2$ values v by using their sum:

$$c(v) = \sum_{i=1}^{n} v_i$$

As there are infinitely many ways to provide the same commitment using the function c, there is no way for the verifier to reconstruct the values used. This results in the prover essentially locking their values along with the infinite set of values resulting in the same commitment. Once the values are revealed the verifier can check, that the commitment indeed equals $\sum_{i=1}^{n} v_i$. This is not the only property of a commitment, as it's clear that the prover can still manipulate the values, providing different $v`_i$ values which produce the same commitment. Still, no inverse function can be defined to get the exact original value of any v_i, without knowing the rest of them.

Commitments have an other core property - not only should it be impossible for the verifier to reconstruct the original values from the commitment, but the prover would take a very long time to come up with values, which would result in the same proof being generated. In other words, it should be very hard time wise (impossible to satisfy both this and the hiding property at the same time [Sma+03b]) for the prover, using a commitment function $c(x)$ and values a and b to result in an equal commitment $c(a) = c(b)$. This property is further referred to as the binding property [CF13].

Using the same example of $c(v) = \sum_{i=1}^{n} v_i$, it's trivial to show, that it does not require the prover many operations, to come up with new values for the exact same commitment. Say the prover used values $(v_0, ..., a, b, ...v_n)$ as the values to generate the commitment. Then once time came to provide these values to the verifier, it was more beneficial for the prover to provide $a`$ instead of $a$. The prover can change the value, but still have a valid commitment - they are able to provide $a`$ as a and $b + a - a`$, as $a` + b + a - a` = a + b$. Hence, this function does not satisfy the binding property.

A good example of a commitment, which has both the binding property and the hiding property could be some cryptographic hash function [Sma+03b]. All hash functions are irreversible, giving the same output for different inputs, but not all may be sufficiently hard complex time wise. As an example, collisions for weak hash functions, such as MD5, can be found within 30 minutes [Spy19]. Therefor still, if using a hash function as a simple commitment scheme, one should investigate how long does it take to find collisions, and whether that satisfies the requirements of a scheme.

1.2 Usage

Commitment schemes are used to ensure that values are not changed. A use case for non-changing values could be a lottery. The commitment scheme for such a lottery could look like this:

1. a lottery with participants p is created;
2. every participant p_i chooses their ticket v_i;
3. participants send commitments of their values c_i, generated by some commitment function $f(x)$;
4. the lottery organiser then reveals the winning ticket value w;
5. if any p_i has the such a ticket where $v_i = w$ they can send this to the organiser;
6. the organiser checks whether $f(v_i) = c_i$

The resulting scheme ensures that every participant is unable to change their ticket value post-commitment (during the reveal stage). Although this example seems to explain what commitments are in a nutshell, a very important detail is missing here. What if the lottery organiser is colluding with any of the participants? A collusion could be that the lottery result w is pre-picked and provided to a participant p_i, who would know simply provide the $f(w)$ as their commitment. This small detail points out that for commitment schemes used in systems without a centralised source of truth it's important to examine all possible attack vectors.

2 Vector Commitments

The following section is about a fairly recently formalised commitment type [CF13], its properties and usages.

2.1 Properties

A vector commitment is a commitment with the addition of a vector positional data being committed along with the value. In essence, the prover is able to commit not only the values of a n vector $(v_1, v_2, ..., v_n)$, but also commit it as an ordered sequence. An ordered sequence would mean that the vector $a = (a_1, a_2, a_3)$ and the vector $a` = (a_1, a_3, a_2)$ have different commitments using a vector commitment function $vc(x)$: $vc(a) \neq vc(a`)$. The prover then is able to prove that the i'th point of the vector they used to generate the commitment is indeed v_i. This property is an extension of the binding property, hence it's called position binding [CF13].

2.2 Usage

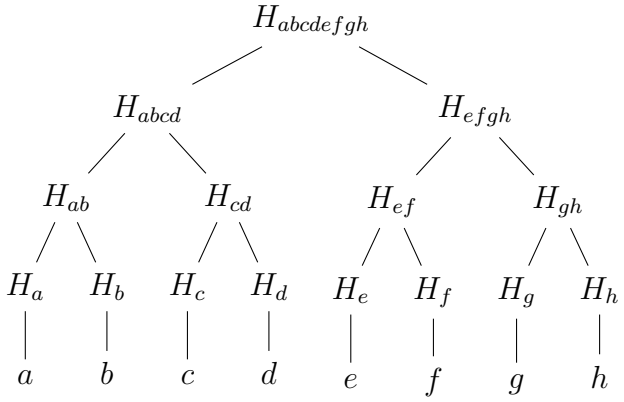
Continuing the lottery ticket where a every participant p_i commits to their ticket v_i, say that the ticket has to commit multiple values where order matters (like a real lottery). First, some definitions:

1. let v_{ij} bet the j'th point value of the i'th participants vector v_i;
2. let the function $c(x, y)$ define the commitment of x being the point value at v_{iy};

then instead of sending the commitment to the ticket v_i, every participant sends a vector of commitments $(v_{i1}, v_{i2}, ..., v_{in})$, where any one of which can verified by the organiser, by using the function c. The verifier can do this until all values are exhausted or an invalid point is found and a win claim can be discarded. The key thing to note here is that this can be equivalent to just committing to the vectors points in an ordered fashion - generating an providing a commitment for every point in a defined order. Generating multiple commitments would require a synchronised communication layer and would force the organiser to respond with some sort of acknowledgements to every commitment. As a vector commitment is a commitment aggregate of the full vector the reliance on a synchronised communication layer is omitted. Therefore committing to a vector in a system without some sort of synchronising layer is better done using vector commitments.

2.3 Merkle Trees

One type of vector commitments are Merkle trees [Kus19a]. A Merkle tree is a type of a binary tree, where any parent node is a hash of its' two children and all leaves are hashes of the vector value. To prove that some value x: is indeed a member of the original value set v at position i, the prover just needs to provide the sub-tree leading from the root to the value x [Szy04]. An example of a merkle tree is shown in the first figure. The values (a, b, c, d, e, f, g, h) are the vector commitment, and all of the H_x are the hashes of the previous two hashes, except the first ones, as they are hashes of the



ex 1. Merkle Tree

vectors points. The commitment to such a vector is all of the hashes: $(H_a, H_b, H_c, ..., H_{abcdefgh})$. A Merkle tree, used as a commitment scheme, provides a way to commit n values using $2 * n - 1$ hashes. In order to check whether any value of a given set belongs in the original value vector, the verifier has to check a $O(\log_2(n))$-sized proof [Kus19b].

Merkle trees have the position binding property therefore also the binding property and they are also hiding, as there is no way to restore the value, once passed through a hash function. Due to their properties, Merkle trees are used as data integrity ensuring constructs in the most popular crypto-currency blockchains, such ”Ethereum” [But20].

In both the ”Ethereum and ”Bitcoin” blockchain Merkle trees are used to verify large numbers of transactions. As one can treat the root of the Merkle tree ($H_{abcdefgh}$ in figure 1) as a commitment, as soon as one transaction in a block is changed, the commitment also changes. As this is essentialy the same action as the proof check, this gives the ability to find the changed transaction in $O(\log n)$ time [Kus19b]. ”Ethereum” uses a modified Merkle tree - a so called Merkle Patricia tree [Tik17]. This modified tree has three roots: one for the transactions of a block, the second root is used to store the state and the third root is the receipt root. The main advantage of using a Merkle tree like structure, is that once an account balance changes, the whole three does not need to be recalculated, only the changed branch.

3 Polynomial Commitments

The commitments described in the previous sections share one common feature - they don't directly describe the committed value and no observations can be made about the correctness of the underlying value. This section talks about Polynomial commitments, specifically the scheme described in the [KZG10] paper. Note, the following subsections compare polynomial commitments to Merkle trees, as polynomial commitments are one of the replacement candidates for Merkle Trees in the "Ethereum" network [But20].

A polynomial $p(x)$ of degree m with coefficients $a_0, a_1, ..., a_m$ can be defined like so:

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

To commit such a polynomial one could commit the polynomial $p(x)$ using conventional methods, i.e. a Merkle tree. Using a Merkle tree would produce a dynamically sized commitment (based on the polynomial degree m) from the coefficients $(a_0, a_1, ..., a_m)$ [KZG10]. This seems reasonable at first glance, but there's now only one way to verify such a commitment: one would also need to reveal all of the coefficients of the polynomial. Note, most simple commitment schemes share the same behaviour - the committer has to reveal the values used to produce the commitment.

Having to reveal the underlying values is not ideal or even not suitable for some cryptographic applications [KZG10]. For example for secret sharing, where multiple evaluations of the same polynomial $p(x)$ have to be provided at different points without revealing the polynomial. The polynomial commitment scheme described in the [KZG10] paper mostly hides the polynomial: one could still guess the values if the polynomial is very simple - i.e. with coefficients $(1, 2, 3, ..., m)$ or if the degree of the polynomial is very small [Fei20; KZG10].

Another problem with vector commitments and their schemes (and one which the KZG10 scheme tries to solve), is that they usually have a large size, or at least scale linearly with the committed values. Having large commitments in large distributed networks is not ideal as one single bit-flip error invalidates the whole commitment and the larger the commitments are, the more likely the mistake. In addition, all of the commitments have to be stored in many nodes and then distributed between them, therefore commitments with the smallest size would be ideal. The polynomial commitments described in the paper have a size of a single field element of group G of prime order p [KZG10]. In practice, this would mean that irrespective to the data size the commitment would be 48 bytes if for example, the chosen elliptic curve pairing is "BLS12_381".

An other benefit of the the polynomial commitment scheme is that opening has a size time complexity of $O(1)$ while the Merkle tree has an opening time complexity of $O(\log n)$ [KZG10; Szy04]. In addition, using the polynomial commitment scheme described in the [KZG10], it is in fact possible to provide proof for multiple evaluations of the polynomial in one element of the group G. According to "Ethereum" researchers, the properties of this scheme are very attractive for not only zero knowledge proof systems [KZG10], but also as a replacement for a vector commitment (Merkle tree) [But20].

4 KZG10 Polynomial Commitment Scheme

This section describes how the scheme presented in the KZG10 paper works in detail. First, let us define some shorthand notation. Given that G_1 and G_2 are two elliptic curves with the pairing $G_1 \cdot G_2 \rightarrow G_T$, let A and B be the generators of G_1 and G_2 respectively. Then, let $[x]_1 = x \cdot A$ and $[x]_2 = x \cdot B$ where x is any field element from the group F of prime order r.

4.1 Trusted Setup

In order to use the polynomial commitment scheme one first needs a trusted setup with some secret s, from the field element group F. This secret should not be available to neither the prover (committer) nor the verifier. Both the prover and the verifier should have the $[s^i]_1$ and $[s^i]_2$ for every i is in $(0, 1, 2, 3...m-1)$ [KZG10; Tom20]. Building this trusted setup is a challenge by itself. One cannot select a pseudo random group element $[s]_1$ (not randomising the secret, but randomising the group element itself) and then compute $[s]_2$, as one would need know the s. It's also not possible to compute $[s^2]$ even if $[s^1]$ is known, but s is not. This results in some system having to know the secret in order to calculate the group elements $[s^i]_1$ and $[s^i]_2$. To solve this, methods exist which allows a group of parties to create the group elements without any one of them knowing the s, unless they all collude [ZCa18].

4.2 Commitment

Using some properties of elliptic functions the prover can do certain computation with the field elements they are given. For example, they can calculate the multiplication product of any arbitrary value u and a curve point $[s^i]_1$ as $u[s^i]_1 = [us^i]_1$. Using the the polynomial $p(x)$ with coefficients $a_0, a_1, ..., a_m$, $p(x) = \sum_{i=0}^{n} a_i \cdot x^i$ the prover can compute $[p(s)]_1 = [\sum_{i=0}^{n} a_i \cdot s^i]_1 = \sum_{i=0}^{n} a_i \cdot [s^i]_1$ [TAB+20].

In the KZG10 polynomial commitment scheme, the value of C, where $C = [p(s)]_1$ is the commitment to the polynomial $p(x)$ [KZG10]. To recap - a commitment C is the evaluation of the polynomial on the elliptic curve in at the secret point s.

4.3 Properties

As this is still a commitment scheme, the commitments generated using this scheme should still satisfy the basic commitment properties - hiding and biding.

The authors of the KZG10 paper state, that the binding property of the scheme is satisfied using the *t*-SDH assumption, while the binding property is based on the DL assumption. Let's look into what this means. Not satisfying the binding would mean that it would be computationally easy for the prover to find an other polynomial $w(x) \neq p(x)$ which produces the same commitment in this scheme: $([p(s)]_1 = [w(s)]_1$. Assuming that is true, then it would mean that $[p(s) - w(s)]_1 = [0]_1$ which in turn means that $p(x) - w(x) = 0$. $w(x) \neq p(x)$ means that the $z(x)$ produced by

$z(x) = p(x) - w(x)$ is a non-constant polynomial of degree m. As $z(x)$ has a degree m that means it can have at most m values which evaluate to 0 [Fei20].

Given that the prover does not have the value s, the only way they could find a $w(x) = p(x)$ is by making the $z(x) = 0$ with as many values as possible. Given that the degree of the curve p is much larger than the possible zero evaluations of the polynomial $z(x)$, the probability, that s is one of the points chosen to make $w(x) = p(x)$ will be quite small. Just how small that probability would be would depend on the exact parameters of the commitment scheme and the system it's used in, but for large, currently existing schemes the order of the polynomial m is 2^{28} and the curve order is $p \approx 2^{256}$[Fei20]. The probability that the prover has managed to find an other polynomial $w(x)$, that results in the same commitment will be about $m \div p$: $2^{28} \div 2^{256} \approx 2 \cdot 10^{-69}$, which is really quite tiny in practice. Hence, the [KZG10] polynomial commitment scheme is sufficiently computationally binding in practice.

The hiding property of this scheme is satisfied out of the box, as the coefficients of the polynomial cannot be restored from the curve points. Of course, it could be done if the polynomial was very trivial: $(1, 2, 3, ..., m)$ or something similar [KZG10].

4.4 Opening

One of the properties of polynomials is that if $p(x)$ is divisible by a linear factor $x - z$, then it has a zero at z: $p(z) = 0$. This is trivial to show by $p(x) = (x - z) \cdot p`(x)$ for some polynomial $p`(x)$, as this evaluates to $p(x) = 0 \cdot p`(x)$ at z.

If the prover would want to provide proof that $p(x) = y$, they could use a polynomial $p(x) - y$. IF the polynomial evaluates to 0 at z, then let $p`(x)$ be the polynomial $p(x) - y$ divided by a linear factor $x - z$:

$$p`(x) = \frac{p(x) - y}{x - z}$$

Then after transforming the equation:

$$p`(x)(x - z) = p(x) - y$$

Given the scheme defined as it is, there is still no way to open the commitment for the verifier without providing the full polynomial. In order to open the commitments we should remember that there exists a paring $G_1 \cdot G_2 \rightarrow G_T$. As some linear operations are possible (i.e. computing the commitment $C = [p(s)]_1$), we just need to find a way to compute the multiplication product of two polynomials. Let's call the elliptic curve pairing $e : G_1 \cdot G_2 \rightarrow G_T$. We can now have that:

$$e([a]_1, [b]_2) = e(A, B)^{(ab)} = [ab]_T$$

we can further simplify this notation by calling these actions as $[x]_T = e(A, B)^x$. In essence, this means that there is a way to multiply two field elements if they have been committed on different curves, which have a pairing e.

The KZG10 proof is defined as $t = [p`(s)]_1$ for the evaluation $p(z) = y$ (The prover wants

to prove the said equality) [KZG10]. As the commitment is defined as $C = [p(s)]_1$ the verifier simply has to check the proof using [Fei20; KZG10; TAB+20]

$$e(t, [s - z]_2) = e(C - [y]_1, H)$$

As the verifier knows $[s]_2$, they can compute $[s - z]_2$, as it's a linear combination of the group element from the trusted setup and z is just the evaluation point of the polynomial (the verifier had had to ask the prover to provide the proof at point z). The verifier also knows the y, as it is the value claimed by the prover at point z: $p(z) = y$, so they can compute the $[y]_1$ as well.

The natural question is why does the equality condition $e(t, [s-z]_2) = e(C-[y]_1, H)$ proves to the verifier that the prover knows $p(z) = y$. This is because of two properties, correctness and soundness [KZG10]. Correctness would mean that if the prover did follow the protocol, then they are able to provide a proof. Soundness means that they cannot produce a false proof, meaning that they would prove that $p(z) = y`$ when $y` \neq y$ [KZG10].

Correctness is trivial to show by $[p`(s) \cdot (s - z)]_T = [p(s) - y]_T$ [Fei20]. It's the equation $p`(x)(x-z) = p(x)-y$ evaluated at some point s. Therefore, the proof should show to the verifier that the prover has followed the steps of the scheme properly (has not cheated).

Soundness is provable in two parts [Fei20; KZG10]. First, in terms of the polynomial the prover would need to divide $p(x) - y`$ by $x - z$. As $p(x) - y`$ is not zero the prover would not be able polynomial division, as there would be a remainder. The prover could also try to work in the elliptic group itself. They would need to find a $t` = ([p(s)]_1 - [y`]_1)^{\frac{1}{s-z}}$. This is impossible under the *q-strong*-SDH assumption [**dandangoyal2007reducing**; Fei20].

4.5 Proof Aggregation

In the previous sections in was shown that one can prove that a polynomial evaluates to some value at a specific point. When compared to a vector commitment of the polynomials coefficients using a Merkle tree some benefits are immediately apparent. The committer is able to show, that an arbitrarily large polynomial evaluates to a specific value with only one group element [KZG10]. In the implementation that was done as part of this work, the elliptic curve was "BLS12_381" in which the group element size is 48 bytes. In a Merkle tree, one would have to send the whole arbitrarily large polynomial, so the difference is straightforward [But20].

One of the primary things that makes the [KZG10] commitment scheme very attractive, is this: proofs in this scheme can be aggregated - a proof for multiple values can be generated [KZG10]. This proof is essentially an evaluation of a polynomial in any number of points, but still can be stored as one group element.

In order to build a proof for multiple values, one first needs some list of n points. The points would have the values $((z_0, y_0), (z_1, y_1), ..., (z_{n-1}, y_{n-1}))$. Using these points one would need to find a polynomial of degree $k \leq n$, which passes all of these points. This is called the *interpolation polynomial* [TAB+20]. To build the interpolation polynomial, one can use Lagrange interpolation or some other methods. Using Lagrange interpolation [TAB+20], the interpolation polynomial can

be defined as $I(x)$:

$$I(x) = \sum_{i=0}^{n-1} y_i \prod_{j=0, j\neq i}^{k-1} \frac{x - z_j}{z_i - z_j}$$

If we assume that the polynomial $p(x)$ passes though all of these points, then $p(x) - I(x)$ will evaluate to 0 at all $(z_0, z_1, ..., z_{k-1})$. This would in turn mean that the polynomial $p(x)$ is divisible by all of the linear factors $(x - z_0), (x - z_1), ..., (x - z_{k-1})$.
We can use these values to build a construct called a *zero polynomial* [Fei20]:

$$z(x) : \prod_{i=0}^{k-1} x - z_i$$

Using the $z(x)$ we can then compute the *quotient polynomial* [Fei20]

$$q(x) = \frac{p(x) - I(x)}{z(x)}$$

Once we have the quotient polynomial, we are able to define a multiproof for the original values $((z_0, y_0), (z_1, y_1), ..., (z_{n-1}, y_{n-1}))$ in the same was as we would define a proof for a single proof, except we use the quotient polynomial: $t = [q(s)]_1$.

For verification, the verifier would also have to compute the interpolation polynomial $I(x)$ and the zero polynomial $z(x)$. Using these polynomials, the verifier is able to compute $[z(s)]_2$, $[I(s)]_1$. Using these values the verfifier can use the pairing equation mentioned before: $e(t, [s - z]_2) = e(C - [y]_1, H)$ [KZG10]:

$$e(t, [z(s)]_2) = e(C - [I(s)]_1, H)$$

Note, H is the generator of G_1 here, which the verifier can access as it is the $[s^0]_1$ element of the trusted setup. The result is that the prover is able to provide a proof for multiple values in one field element.

5 KZG10 Commitments in Blockchain Technologies

KZG10 polynomial commitments can be used to replace existing vector commitment schemes [But20], where network footprint, and proof size matters. In the second section, it was described, that the main property of vector commitments is position binding. In simplest terms, the position binding property in commitment schemes can be described like so: the committer being unable to change the positions of the values being committed and then being able to provide proof that in the commited vector $(a_1, a_2, ..., a_n)$, the value a_i is the i'th value of the vector. Currently, large distributed blockchain networks such as the "Ethereum" network use Merkle trees as their vector commitment schemes [Tik17]. Merkle trees have a proof size of $O(\log n)$ [Kus19b] and the same complexity [Szy04], just to prove, that some one element is of the vector is indeed at the location that the committer specified. The same exact commitment can be achieved using polynomial commitments. We can define a polynomial $p(x)$, which evaluates to the values that we want to commit for all i, so that $p(i) = a_i$. Such a polynomial can be built in multiple ways, one would be using the aforementioned Lagrange interpolation. Now, the commitment is a constant size element, which position binds *all* of the $(a_1, a_2, ..., a_n)$.

5.1 State Root Replacement

Ethereum foundation researchers are considering to use polynomial commitments as a replacement for state roots (Merkle trees) [But20]. This has it's own problems though. In order to recompute the commitment, all of the values have to be re-evaluated, while using a Merkle tree, it's enough to recompute the single branch where the value has changed. So for updates the difference in computing is significant $O(n)$ versus $O(\log n)$. "Ethereum" researcher V. Buterin suggests solving this by splitting the state into caches [But20].

The ideal vector commitment for the "Ethereum" network is still an open problem [But21b]. The polynomial commitment scheme described in the KZG10 paper has some useful properties - proof aggregation, constant size commitments [KZG10]. On the other hand it requires a trusted setup (generating values using a shared secret, without revealing the secret). The scheme is also computationally heavy for state which is constantly updated [But20].

It is clear, that this scheme has some benefits and some drawbacks which need to be addressed. According to "Ethereum" researchers, the polynomial commitment scheme is still the closest one to being ideal [But21b]. Due to it possibly being used in the "Ethereum" network, researchers have begun implementing prototypes of KZG10.

5.2 Data availability sampling

An other usage of the KZG10 polynomial commitment scheme in blockchain technologies, would be so called data availability sampling or DAS for short [But21a]. The goal of data availability sampling is to provide an efficient way to prove that some data is published and downloadable (available) on a distributed network.

A trivial way to achieve data availability sampling would be for all of the clients to download all of the data, so they directly know what is available and what is not. Using the KZG10 scheme, this can be achieved indirectly, saving network resources [But21a]. The key thing to remember here - the KZG10 commitment commits a vector, that always describes a polynomial. Due to the commitment being made on a polynomial, general polynomial properties can be used. This turns out to be useful as one can commit data as a polynomial that evaluates to the data.

Given a data array of length n: $D_0, D_1 ... D_{n-1}$, one can build a polynomial $P(x)$ that evaluates to the data in the given index positions: $P(0) = D_0$, $P(1) = D_1$, ... ,$P(i) = D_i$, ... , $P(n-1) = D_{n-1}$ [But21a]. Now, for data availability sampling, one property of polynomials can be used. A polynomial of degree k any distinct k evaluations of the polynomial can be used to reconstruct the polynomial. Therefore, one can reconstruct the original data if they would extend the polynomial $P(x)$ to n more evaluations [But21a] (the total size of the committed polynomial would be $2n$).

According to ”Ethereum” founder and researcher V. Buterin, this solves a large problem. To check whether data is available, a client would have to download 50% of the data instead of 100%. The latter, has a problem, where when the client has almost all of the data, say 90%, they’re dependent on the last 10% of the data to not contain a missing chunk. Using polynomials for data availability sampling allows the ”Ethereum” network be scalable [But21a].

The part where the KZG10 scheme could be applied is that the polynomial can be committed, and obtain all of the KZG10 scheme commitment properties. One could generate openings for all of the $2n$ outputs of the polynomial, which would result in a data block not being able to be invalid - the data block to be either valid or unavailable [But21a].

6 Implementation

Implementation of the KZG10 polynomial commitment scheme is the part in achieving the goals set out for this work. The main criterion set out for the implementation is that it should be performant. Originally, the author planned to implement KZG10 both in C# and Rust. In the end the C# implementation was used for prototyping in order understand the specifics of the scheme better. On the other hand, the Rust implementation was the one that focused on performance.

A lot of the comparison later involves Herumi MCL and Kilic BLS libraries. At least benchmarks of pairings indicate, that Kilic BLS is much slower than Herumi MCL (in the authors' tests about twice as slow on the same machine). Benchmarks of both libraries can be found in their respective "Github" pages. Therefore, the assumption from this point is that the Kilic BLS library is slower than Herumi MCL.

6.1 Language Choice

The author initially chose C# in order to not have the compound complexity of learning a language and understanding non-trivial algorithms at the same time.

Rust was chosen as the low level language for the performance oriented implementation. This is because it offers low level language tools such as manual memory management (if needed - the compiler usually handles the memory management), zero-overhead C API interaction [Cri15]. The latter is important, because the author has chosen to use Herumi MCL as the library of choice for elliptic curve operations, which is implemented in C. In comparison to other languages, Rust performs very similarly to C++ and C [Tea21], while providing additional safety features. For instance, one of the main tools that the Rust language offers is the borrow checker. In essence, it allows for the Rust compiler to make memory safety guarantees without needing solutions with high performance overhead, such as garbage collection. The Rust compiler ensures, that a memory region can only have one writer at the time, while having multiple readers. As a result of the borrow checker fixture, the programmer using Rust is ensured by the compiler of certain features that their code has.

Additionally, just like programs written in C or C++, Rust programs can be compiled into library files (such as DLLs). This results in library type programs written in Rust being reusable and importable into higher level languages, which in turn allows programmers to write performance-impacting code in Rust and then use the resulting library in higher level languages. This was an additional reason, for the authors' choice not to implement the fully optimised version in C# as it would be outperformed by rust, and could be just imported as a library itself.

In addition, the Rust language is constantly evolving, adding new features and staying idiomatic. As a result of this it's a very pleasant experience to code in this language: the 2020 Stack Overflow survey listed Rust as the most loved language [Sta20]. The author thinks this is noteworthy as low level languages are notoriously hard to work with.

6.2 Libararies

In order to implement the polynomial commitment scheme described in the KZG10 paper, one first needs a elliptic curve algebra library. The Herumi MCL library offers all of the elliptic curve operations needed by the KZG10 scheme, is written in C and can be imported and used in Rust with zero overhead using the C API [Cri15].

The author will be comparing their implementation with the implementation of one of the ”Ethereum” researchers’. The said implementation was done using the Go language and Herumi MCL as the elliptic curve algebra library [Pro21a]. This implementation was chosen for comparison, because at that point in time, this was the only library which implemented the functions required to enable data availability sampling in the proposed ”Ethereum” protocol.

In addition, due to some layers of abstraction provided by the Rust language, it wouldn’t be hard to exchange Herumi MCL with another library which has the required elliptic curve operations to implement the KZG10 scheme.

For benchmarking the Rust implementation, the author used a library called ”criterion”. This library runs the same test multiple times, providing statistically significant results. As an example, even though individually, the benchmarked functions cost about a minute of combined execution time, due to all of the benchmark iterations, running the benchmarks can take up to an hour.

6.3 Data Structures

A polynomial as a data structure can be represented as an vector of coefficients. A vector in programming is best represented as an array, hence a polynomial in the Rust implementation can be defined as:

```
#[derive(Debug, Clone)]
pub struct Polynomial {
    pub coeffs: Vec<Fr>
}
```

One important derivation here is the Clone trait. It allows for easy cloning of the data structure when needed, but does not support implicit cloning defined by the Copy trait. As polynomials can have any degree (of course the coefficients should be able to fit in computer memory), copying them might be a very costly operation. To prevent accidental copying when passing the polynomial as a parameter, only the Clone trait is implemented.

In the same way, all of the elliptic group primitives are defined with a Clone trait, but without the Copy trait. Even though their size is large compared to usual primitives (usually integer is 4 or 8 bytes) - the size of the Polynomials coefficients (Fr type) is 32 bytes. In this way the author, as a programmer would have to think about the performance every time they copied the value, as it would have to be explicit.

An other data structure defined was the Curve structure.

```
#[derive(Debug, Clone)]
pub struct Curve {
    pub g1_gen: G1,
    pub g2_gen: G2,
    pub g1_points: Vec<G1>,
    pub g2_points: Vec<G2>,
    pub order: usize
}
```

This structure is essentially used to represent the trusted setup [KZG10] defined in the 4.1 section: $[s^i]_1$ and $[s^i]_2$ for every i in $(0, 1, 2, 3...order - 1)$. The field `g1_points` stores the $[s^i]_1$ and `g2_points` stores the $[s^i]_2$.

This data structure is even larger - the order of it has to be as large or larger than the degree of polynomials being committed. The elliptic group elements defined by G1 and G2 are much larger that the polynomial coefficients - every group element is 48 bytes, hence copying this structure implicitly by accident would be a very expensive operation. That's why it also has a Clone trait instead of a Copy trait.

6.4 KZG10 Scheme

6.4.1 Trusted Setup

In the KZG10 polynomial scheme, the first step is the trusted setup [KZG10]. If this library were to be used in the real world, the trusted setup would be a dependency in order to start using this scheme. However, in order to test a way to build a emulated trusted setup with a known secret is required. This setup is implemented in the `Curve::new` function. As this setup is used throughout the testing of the library, it is important to verify the correctness of the outputs at least to some extent. The correctness is verified informally by these tests:

- curve_new_sets_g1_gen_to_correct_val
- curve_new_sets_g2_gen_to_correct_val
- curve_new_first_g1_point_is_generator
- curve_new_first_g2_point_is_generator
- curve_new_g1_points_should_have_exact_values_given_specific_params

6.4.2 Commitment

The commitment for a polynomial with coefficients $(a_0, a_1, ..., a_{n-1})$ in this scheme [Fei20; KZG10] is calculated like so:

$$C = \sum_{i=0}^{n} a_i \cdot [s^i]_1$$

This is implemented in the function `Polyniomial::commit`. The size of the commitment is a single G1 element as defined in the scheme [KZG10] and the implementation of the whole calculation is directly delegated to the Herumi MCL library. Polynomial commitments are also covered by tests:

- polynomial_commit_should_have_specific_value_given_exact_inputs
- proof_loop_works_with_random_points

The latter test uses a random random point for proofs used to verify that the committer knows the polynomial.

6.4.3 Proof

Let's remember that the proof for the polynomial is $t = [p`(s)]_1$ [KZG10], where $p`(x)$ is the quotient polynomial built from the value that the verifier asked to provide the proof for and the polynomial $p(x)$.

In the code this is implemented in the function `Polynomial::gen_proof_at`. As seen from the function signature, the result (proof) has the size of a single elliptic group element as defined by the scheme. In order to implement this function, a function for polynomial division had to be implemented, as the Herumi MCL library does not have such a function. Just like the commitment, the proof generation is delegated to the Herumi MCL library function - `mclBnG1_mulVec`.

This functionality is covered by these tests:

- polynomial_generate_proof_at_should_have_a_specific_value_given_exact_inputs
- proof_loop_works_with_random_points

The proof loop test in this case acts to simulate the verifier asking for different proofs and perhaps to catch some edge case unhandled by the code.

6.4.4 Proof Verfication

The final step of this scheme is for the verifier to check whether the provided commitment and proof are valid. This is achieved via elliptic curve pairing [KZG10; TAB+20]:

$$e(t, [z(s)]_2) = e(C - [I(s)]_1, H)$$

In the implementation this behaviour can be found in the function `Curve::is_proof_valid`. One noteworthy thing here is that even though the trusted setup along with the whole scheme is emulated in tests, the data structures used to store information and the functions defined for these data structures are separated akin to the real-world scenario. As an example, the verifier is assumed to have access to the `Curve` (trusted setup) data structure and can operate using it, hence the proof validity being defined as a function of the `Curve` data structure.

These are the tests that cover proof validation functionality:

- curve_is_proof_valid_should_return_true_when_same_parameters_used_for_gen_are_passed
- proof_loop_works_with_random_points

6.5 FK20

The KZG10 polynomial commitment scheme can be optimised using mathematical methods. One of the ways to do so has been described in the ”Kate Amortized” paper [FK20] (note, ”Kate” is a common name for the KZG10 commitment scheme). This paper defines optimisation for this scheme using Fast Fourier Transforms (FFT for short). Without getting into the details why this works, the result is that the computational complexity of certain operations using polynomials can be reduced to $O(N \cdot \log N)$ instead of $O(N^2)$ [But19]. For small N these optimisations are barely noticeable, and may even perform worse as they introduce some overhead complexity, but for large N the difference should be apparent.

As in practice the polynomials committed using this scheme would be quite large [But20; But21a] it makes sense to optimise it for large N values. Due to the said reason, the Protolambda implementation of the KZG10 scheme with FK20 optimisations only provides the benchmarks of the FFT based actions [Pro21b], as they indicate the real world performance the best. The author argues that in order to better see the real world performance, one should use performance oriented languages as they would provide the fairest and least misleading comparison to existing alternative schemes.

6.5.1 Additional Data Structures

In order to use the optimisations defined by the FK20 paper, some additional data structures are needed. First, the `FFTSettings` structure, which stores some pre-calculated values, which are used for the FFT calculations, as the name `FFTSettings` suggests:

```
pub struct FFTSettings {
    pub max_width: usize,
    pub root_of_unity: Fr,
    pub exp_roots_of_unity: Vec<Fr>,
    pub exp_roots_of_unity_rev: Vec<Fr>
}
```

An other structure called `FK20Matrix` is also used to store pre-computed values:

```
pub struct FK20Matrix {
    pub curve: Curve,
    pub x_ext_fft_files: Vec<Vec<G1>>,
    pub fft_settings: FFTSettings,
    pub chunk_len: usize,
}
```

These structures are intended to be used as singletons, as the values within them only need to be computed once, and don't need to change.

6.5.2 Multiple Proof Generation

One of the main features of the FK20 based implementation is generating multiple proofs efficiently [FK20] ($O(N \cdot \log N)$ complexity instead of $O(N^2)$ [But19]). The multiple proof generation has been implemented in the function `FK20Matrix::dau_using_fk20_multi`. It is covered by multiple tests:

- dau_using_fk20_multi_generates_exact_values_given_known_inputs
- fk20_multi_proof_full_circle_fixed_value
- fk20_multi_proof_full_circle_random_secret

6.5.3 Multiple Proof Verification

Having generated multiple proofs, they have to be verified. This is implemented in the function `FK20Matrix::check_proof_multi`. It's covered by these tests:

- fk20_multi_proof_full_circle_fixed_value
- fk20_multi_proof_full_circle_random_secret

6.6 Optimizations

Optimising the calculations mathematically is not trivial in this case, as one would need to have a very deep understanding how the elliptic curves and fast Fourier transforms work, but this could be an interesting research subject for a masters degree. There are some areas of code which could be optimised without such knowledge. There are multiple parts in the code, which allocate new memory instead of reusing it. For example in the `FK20Matrix::fft_g1` function the calculations are carried out on array copy of G1 field elements. Each element takes up 48 bytes so the allocation could be costly for large arrays. The same could be done in the `FK20Matrix::fft_g1_inv` function.

An other location, where code could definitely be optimised, is the `FK20Matrix::_fft_g1` function. This function recursively manipulates given values, until a certain threshold is met. At the point of this threshold the algorithm is swapped out with a simpler one. The threshold could be fine tuned based on experiments with data sets similar to real ones. Although, at this point in time, the exact commitment scheme is still being discussed [But21b], so it would be hard to know the exact details of the data.

As the division action is one of the most costly operations on current computers, all division has been swapped to right shifting.

6.7 Verification

If a cryptographic library were to be used in real production environments, it should also be formally verified as an impact of an insecure cryptographic library can be severe. Formal verification is a complex process on it's own [Bje05]. As this is an exploratory implementation, the efforts to verify the library formally would be too large.

Even though formal verification is too burdening at this stage of the KZG10 library implementation, one can verify the correctness informally. For example, one can ensure that tests that other libraries use are ported to their implementation. All applicable tests from the Go implementation have been ported to the implemented library.

There are also tests which use specific values to test mathematical edge cases in dependent libraries. For example the function `build_protolambda_poly` builds a polynomial with a large range of coefficients. The coefficients include such cases like -1 (in computer memory this would be the maximum value in a memory cell or a register). Such values help to ensure, that the libraries dependencies are not providing incorrect values and help identify differences (mistakes) in algorithms if any.

Furthermore, various infrastructure elements - functions which don't implement the KZG10 algorithms, but are needed as helpers are tested. For example, the function `is_power_of_2` is covered by a test `get_next_power_of_two_returns_correct_values}`.

Lastly, as the this library is a commitment scheme implementation, there are tests which test the full commit-reveal cycle with randomised values. This ensures that the behaviour of the library is invariant to the variables used.

7 Benchmarks

The most important part of this research is the performance benchmarks between the Protolambda KZG10 implementation using Go and the authors' implementation using Rust. This section presents the relevant benchmarks and analyses them.

The authors' code contains multiple additional benchmarks, such as a benchmark for KZG10 commitments, proof generation and proof checking. These benchmarks are not the main focus, as the FK20 paper describes optimised ways to generate and check proofs. The optimisations are achieved using a mathematical method called fast Fourier transforms [But19]. Hence, the benchmarks provided further focus in the FFT actions, which are also the focus of benchmarks in the Go implementation.

The benchmarks provided by Protolambda seem to be valid across different platforms. The author has run the Go benchmarks on multiple machines, achieving very similar results to the ones originally provided by Protolambda (virtually no difference, sometimes the performance would be worse, sometimes better, but always within a few percent). It appears to not be bound by computational resources, but more by some other, perhaps input-output operations. Due to said reason, the author decided to use the benchmark numbers provided by Protolambda as the source of truth for further comparison. Additionally, the Rust benchmark code is included in the source code, therefore anyone doubting the results is able to run their own tests.

The Go implementation supports multiple elliptic curve algebra libraries. One of them is Herumi MCL, which was also used in the authors' Rust implementation. An other library used - Kilic BLS, which is a elliptic curve algebra library written natively in go.

One issue with Herumi MCL, is that it's implemented in C (issue for Go). To operate this library in other languages, one needs to use the C API bindings. The Go language is notoriously slow when it comes to consuming C APIs [All18]. Hence, for a more comprehensive comparison, the author will compare their implementation with both the Go implementation using Herumi MCL as the underlying elliptic curve algebra library and the Kilic BLS library.

7.1 FFT DAS Extension

The first and second benchmarks (seen in tables one and two respectively) compare the time complexity of

`FFTSettings.DASFFTExtension`

implemented in Go with the authors' Rust implementation in the function

`FFTSettings::das_fft_extension`

This function is used to prepare data for data availability sampling [But21a]. This is a scenario, which might be used in "Ethereum" network. Further, the author presents a table of benchmarking results.

In all of the tables the scale value is the variable used to decide how much large should the data set be. Each increment of the scale variable increases the amount of data to process by two-fold:

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	6943	1421	5522	0.2046665706
5	16354	2681	13673	0.1639354286
6	36731	5932	30799	0.1614984618
7	79517	12817	66700	0.1611856584
8	187907	29108	158799	0.1549064165
9	448536	64748	383788	0.1443540764
10	933580	147840	785740	0.1583581482
11	1842469	321590	1520879	0.1745429638
12	4351600	704940	3646660	0.1619955878
13	9669196	1559400	8109796	0.1612750429
14	21868114	3513600	18354514	0.1606722921
15	44655458	7470600	37184858	0.1672942197

Table 1. MCL/Go vs MCL/Rust

2^{scale}. The absolute difference (absolute diff column) is the difference in processing time between the Go and Rust implementations - negative numbers would mean that the Go implementation has a smaller time complexity, while the value being positive would mean that the Rust implementation has a smaller time complexity. Lastly, the final column (relative diff) is the the performance of Rust code in Relation to the Go code.

Running these benchmarks multiple times yields roughly the same results: the Rust code takes about 16% of the time that the Go implementation takes (see the first table). It seems, that this is also scale independent. The author would argue that the pattern would continue for larger scales. This seems like a promising result, as the difference is non-negligible and does not seem to be explainable by variations in the data (running the benchmarks multiple times yields the same result).

As Go performance is easily hindered by C API calls [All18] and the Herumi MCL library is written in C/C++ and usable through it's C API, it's also important to exchange Herumi MCL with some native Go library for elliptic curve algebra. In this case, the protolambda implementation also supports Kilic BLS as the underlying library for such actions. The results of such comparison can be seen in the seen in the second table.

The difference between these two implementations is not as big. Still, the Rust implementation using Herumi MCL takes 60% of the time that the Go implementation takes for large scales. The author argues, that the 40% difference still quite a significant. One thing that was noticed, is that these computations do not seem to be CPU bound - staying at around 15-20% usage thought the benchmarking process. This could imply that the trivial optimisations mentioned in the 6.6 Section, could speed up the Rust implementation performance even more.

So far, the data seen in the figures seems to indicate, that the implementation written in Rust might be generally faster than the one written in Go.

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	6436	1421	5015	0.2207893101
5	9426	2681	6745	0.2844260556
6	14649	5932	8717	0.4049423169
7	27099	12817	14282	0.4729694823
8	50896	2910	21788	0.5719113486
9	108707	64748	43959	0.5956194173
10	231713	147840	83873	0.6380306672
11	516664	321590	195074	0.6224354706
12	1169011	704940	464071	0.603022555
13	2569475	1559400	1010075	0.6068944045
14	5421951	3513600	1908351	0.6480324149
15	11377382	7470600	3906782	0.6566185437

Table 2. Kilic BLS/Go vs MCL/Rust

7.2 FFT Fr

The next important benchmark tests data conversion to a polynomial, which evaluates to said data - polynomial interpolation. This is the second phase of preparing data for data availability sampling [But21a]. This behaviour is described in the 5.2 section of this work. The compared functions here are:

```
FFTSettings.FFT in Go
```

```
FFTSettings::fft in Rust
```

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	12176	908	11268	0.07457293035
5	27535	1985	25550	0.07209006719
6	66410	4462	61948	0.0671886764
7	140656	9684	130972	0.06884882266
8	287225	21415	265810	0.07455827313
9	646755	48218	598537	0.07455373364
10	1404484	101900	1302584	0.07255333631
11	2735691	214910	2520781	0.07855784882
12	6183629	464080	5719549	0.07504978064
13	14290446	990300	13300146	0.0692980471
14	28762232	2141570	26620662	0.07445771246
15	41157878	5060100	36097778	0.1229436561

Table 3. MCL/Go vs MCL/Rust

The results seen in the third table are even more significant than previous benchmarks. The Rust implementation seems to only take up 7% of the time, that the Go implementation takes (over 14 times faster). As the code correctness is not verified formally, this could be an indication of an issue within the Rust implementation. The author argues that this is unlikely, because the functionality is covered by multiple tests, which are also implemented in the Go library. Instead,

the author argues, that this is the result of Rust being faster than Go [Mul20], and the many C API calls that have to be made within this algorithm. At the very base of the algorithm, there are very many small actions (simple arithmetic) over the whole data set of size 2^{scale} using calls via the C API. Rust has zero overhead, while Go can take up as much as 171ns for a single call [All18]. This is further supported by the next benchmark (table four), where the Go implementation uses Kilic BLS as the underlying algebra library.

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	12176	908	3083	0.2275119018
5	27535	1985	6398	0.2367887391
6	66410	4462	14983	0.2294677295
7	140656	9684	30762	0.2394303516
8	287225	21415	65865	0.2453597617
9	646755	48218	149666	0.2436680075
10	1404484	101900	315988	0.2438452408
11	2735691	214910	666139	0.2439251392
12	6183629	464080	1447791	0.2427360423
13	14290446	990300	2900941	0.2544946458
14	28762232	2141570	6189642	0.2570538356
15	41157878	5060100	10382764	0.3276659045

Table 4. Kilic BLS/Go vs MCL/Rust

Table four is the benchmark comparison of the same action as in table three, except the library used in the Go implementation is Kilic BLS. Here, the results are more conservative than in table three. On the largest scales, there is a three times difference in time complexity between the Rust implementation and the Go implementation. This further supports the hypothesis, that the culprit is C API call overhead as Kilic BLS is slower than Herumi MCL in general.

7.3 FFT G1

The final two benchmarks seen in tables five and six compare the performance of

`FFTSettings.FFTG1 in Go`

`FFTSettings::fft_g1 in Rust`

These functions generate all proofs for a given polynomial for the DAS scheme. The fifth table shows that the Rust implementation takes around half of the time of Go implementation. This comparison shows a few interesting points - one is that the difference is not as big in previous comparisons (although it is still substantial), and the gap between these two implementations seems to be closing by around 1% per one difference in scale. Note, the actual data set is getting larger in a non linear way - every one increase in scale, means a double data set. There are a lot of non-aggregate calls to the Herumi MCL library in this function, but it appears that in this functions case the algebraic actions cost more (time wise) than the C API overhead in Go.

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	1900117	852670	1047447	0.4487460509
5	4524605	2090200	2434405	0.461962978
6	10362223	5073100	5289123	0.4895764162
7	25088604	12159000	12929604	0.48464235
8	57022201	28021000	29001201	0.4914050933
9	125807653	63987000	61820653	0.5086097584
10	281827965	144840000	136987965	0.5139305462
11	624146434	319550000	304596434	0.5119792129
12	1379302210	704190000	675112210	0.510540761
13	2921717660	1538400000	1383317660	0.5265395836
14	6351155786	3332800000	3018355786	0.5247548812
15	13513868449	7173600000	6340268449	0.5308324576

Table 5. MCL/Go vs MCL/Rust

Scale	Go (ns)	Rust (ns)	Absolute diff (ns)	Relative diff
4	4592074	852670	3739404	0.1856829833
5	11496429	2090200	9406229	0.1818129786
6	28781767	5073100	23708667	0.1762608946
7	64030299	12159000	51871299	0.1898944748
8	148535217	28021000	120514217	0.1886488643
9	305014341	63987000	241027341	0.2097835787
10	760028615	144840000	615188615	0.1905717721
11	1684792328	319550000	1365242328	0.1896672929
12	3745748396	704190000	3041558396	0.1879971438
13	7411640027	1538400000	5873240027	0.2075653964
14	12106541411	3332800000	8773741411	0.2752891918
15	21183031053	7173600000	14009431053	0.3386484201

Table 6. Kilic BLS/Go vs MCL/Rust

The results in the sixth table compare the same functions as in table five, but the underlying library in Go is Kilic BLS instead of Herumi MCL. As mentioned in the analysis of the fifth table, the overhead of calling the C API is much smaller in this function, when compared to the time cost algebraic actions themselves. As Kilic BLS is slower than Herumi MCL, in this case removing the overhead still does not outperform the Go/Herumi MCL variant. The difference between the computation on the largest data set using Herumi MCL and Kilic BLS (last entries in table 2 and six) appears to be about 20%.

Results

1. Commitment schemes in general were analysed and presented in this work.
2. The author detailed how the KZG10 polynomial commitment scheme works and how it could be used in blockchain technologies, focusing on the ”Ethereum” network.
3. Multiple libraries for the KZG10 commitment scheme were implemented:
   (a) C# implememntation;
   (b) Rust implementation.

   The C# implementation was used in order to help the author understand the KZG10 scheme better, as it is a complex cryptographic commitment scheme.

   The Rust implementation was oriented towards performance, and was compared with an implementation written by ”Ethereum Foundation” researchers in Go.
4. The implementations in both languages have been informally verified to be working correctly by porting tests from existing libraries and by using randomised commit-reveal cycle tests.
5. The Rust implementation of the KZG10 scheme was benchmarked with equivalent benchmarks as the existing Go implementation, producing insightful results. When using a language which has no overhead for calling the C APIs, for the scheme parts, which perform a lot of calls via C API, a significant performance increase was noticed. At its worst, the Go implementation performed over 14 times slower than the Rust implementation, while at best performing at just over 37% slower than the Rust implementation. This is further supported by the fact that in some scheme parts where a lot of small calls to algebra library are made, an implementation using a slower algebra library outperformed an implementation using a faster one (as seen in tables 3 and 4).

8 Conclusion

The goal set out for this work was achieved - an implementation of the KZG10 polynomial commitment scheme using Rust suitable for data availability sampling, that performs better than Go implementation. As seen in the benchmarks, language choice can have a large impact on performance. The authors of the Go implementation tried to circumvent the language performance issues when it comes to C APIs [All18] by using a library written directly in Go. That did not produce the desired result in all of the functions, as the Go library performs some calculations slower than the one written in C [Pro21b].

Based on the results, the author argues, that programming language choice should be one of the primary factors when trying to implement complex, time-efficient algorithms. When choosing a language, one should first evaluate the performance of the languages themselves, as that on its own may provide significant results. Another part of this evaluation should be the investigation of what type of libraries are going to be used. If the libraries have a special way of communication (such as calling it via a C API) and if the number of calls to the library is large and cannot be aggregated easily, then one should also consider using a language with the best performing communication.

References

[All18] Sean T. Allen. Gophercon 2018 - adventures in cgo performance, 2018. https://about.sourcegraph.com/go/gophercon-2018-adventures-in-cgo-performance/.

[Bje05] Per Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1–es, 2005.

[But19] Vitalik Buterin. Fast fourier transforms, 2019. https://vitalik.ca/general/2019/05/12/fft.html.

[But20] Vitalik Buterin. Using polynomial commitments to replace state roots. https://ethresear.ch/t/using-polynomial-commitments-to-replace-state-roots/7095, 2020.

[But21a] Vitalik Buterin. Data availability sampling phase 1 proposal. https://hackmd.io/@vbuterin/das, 2021.

[But21b] Vitalik Buterin. Open problem: ideal vector commitment, 2021. https://ethresear.ch/t/open-problem-ideal-vector-commitment/7421/27.

[CF13] Dario Catalano and Dario Fiore. *Vector commitments and their applications*. 2013, pp. 1–2.

[Coi21] CoinMarketCap. Today's cryptocurrency prices by market cap. coinmarketcap.com, 2021.

[Cri15] Alex Crichton. Rust once, run everywhere, 2015. https://blog.rust-lang.org/2015/04/24/Rust-Once-Run-Everywhere.html.

[Fei20] Dankrad Feist. Kate polynomial commitments, 2020. https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html.

[FK20] Dankrad Feist and Dmitry Khovratovich. Fast amortized kate proofs, 2020. Checked 2021-05-25.

[Fou21] Ethereum Foundation. About the ethereum foundation. https://ethereum.org/en/foundation/, 2021.

[Inv21] Investopedia. Bitcoin's price history. https://www.investopedia.com/articles/forex/121815/bitcoins-price-history.asp, 2021.

[Kus19a] John Kuszmaul. Verkle trees. *Verkle Trees*:2–3, 2019.

[Kus19b] John Kuszmaul. Verkle trees. *Verkle Trees*:1, 2019.

[KZG10] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Polynomial commitments. *Tech. Rep*, 2010.

[Mul20] Michiel Mulders. When to use rust and when to use go, 2020. https://blog.logrocket.com/when-to-use-rust-and-when-to-use-golang.

[Pro21a] Protolambda. Kzg10 commitment scheme in go, 2021. Checked 2021-05-25.

[Pro21b] Protolambda. Kzg10 commitment scheme in go (benchmarks), 2021.

[SBF+20] Johannes Sedlmeir, Hans Ulrich Buhl, Gilbert Fridgen, and Robert Keller. The energy consumption of blockchain technology: beyond myth. *Business & Information Systems Engineering*, 62(6):599–608, 2020.

[SM19] Sarwar Sayeed and Hector Marco-Gisbert. Assessing blockchain consensus and security mechanisms against the 51% attack. `https://doi.org/10.3390/app9091788`, 2019.

[Sma+03a] Nigel Paul Smart et al. *Cryptography: an introduction*, vol. 3. McGraw-Hill New York, 2003, pp. 363–364.

[Sma+03b] Nigel Paul Smart et al. *Cryptography: an introduction*, vol. 3. McGraw-Hill New York, 2003, p. 365.

[Spy19] SpyCloud. How long would it take to crack your password? `https://spycloud.com/how-long-would-it-take-to-crack-your-password/`, 2019.

[Sta20] StackOverflow. Most loved, dreaded, and wanted languages, 2020. `https://insights.stackoverflow.com/survey/2020`.

[Szy04] Michael Szydlo. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 541–554. Springer, 2004.

[TAB+20] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. Cryptology ePrint Archive, Report 2020/527, 2020. `https://eprint.iacr.org/2020/527`.

[Tea21] Benchmarks Game Team. Which programs are fastest?, 2021. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html`.

[Tik17] Sergei Tikhomirov. Ethereum: state of knowledge and research perspectives. In *International Symposium on Foundations and Practice of Security*, pp. 206–221. Springer, 2017.

[Tom20] Alin Tomescu. Kate-zaverucha-goldberg (kzg) constant-sized polynomial commitments. `https://alinush.github.io/2020/05/06/aggregatable-subvector-commitments-for-stateless-cryptocurrencies.html`, 2020.

[Tru18] Jon Truby. Decarbonizing bitcoin: law and policy choices for reducing the energy consumption of blockchain technologies and digital currencies. *Energy research & social science*, 44:399–410, 2018.

[ZCa18] ZCash. Parameter generation. `https://z.cash/technology/paramgen/`, 2018.

Appendix no. 1
KZG10 scheme implemenetation in Rust

https://github.com/UndeadRat22/kzg10-rust

Appendix no. 2
KZG10 scheme implementation in C#

https://github.com/UndeadRat22/kzg10-poly-commit