UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

An elliptic curve based Proof of Work implementation for sustainable cryptocurrency mining

Autor: Mario Moliner Cano

Directors:	Adriana Moya, Dr. Eloi Puertas
Realitzat a:	Departament Matemàtiques i informàtica
Barcelona,	24 de gener de 2022

Abstract

The aim of this work is to study the foundations of cryptocurrencies and blockchains technologies with a critical mindset in terms of ecosostenibility. We begin with an an overview of how these emerging technologies work to then move onto the mathematical basis that cryptocurrencies build upon. Here we stop to study in-depth the *Elliptic curve discrete logarithm* ECDLP, a classical problem in cryptography. We then study and give a concrete implementation of a novel and more sustainable mining algorithm based in the ECDLP. We compare the results with bitcoin's mining algorithm by providing metrics for our algorithm and checking its feasibility. We then finish this work by giving an interface to this newly developed algorithm an encouraging its deployment in the future.

2020 Mathematics Subject Classification. 94A60, 11Y16, 14H52

Resum

L'objectiu principal d'aquest treball és l'estudi dels fonaments de les criptomonedes amb l'ull posat en l'ecologisme o la manca d'aquest a vegades. Comencem el nostre treball fent un estudi sobre com aquestes tecnologies disruptives funcionen per després moure'ns a estudiar les bases matemàtiques sobre les quals funcionen. Aquí ens detindrem i farem un estudi més en profunditat del problema del *logaritme discret sobre corbes el·líptiques* abreujat ECDLP, un problema clàssic de la criptografia. En acabat estudiarem i donarem una implementació concreta d'un algorisme nou i més sostenible de minat basat en el ECDLP. Donem una comparació dels resultats obtinguts sobre els de l'algorisme de minat que utilitza bitcoin tot donant mètriques i comprovant que sigui factible en un futur. Acabarem el treball donant una interfície d'aquest nou algorisme incentivant la seva futura aplicació.

Abstrakt

Das Ziel dieses Arbeit ist die Studium der Grundlagen von Kryptowährungen mit der Berücksichtigung von Öko-nachhaltigkeit. Zunächst machen wir ein Überblick über der Funktion diesen neuen Technlogien. Dann konzentrieren wir uns auf dem mathematische Hintergrund, nämlich Kryptographie. Wir machen hier eine kleine Pause um tiefer das Diskreter–Logarithmus–Problem der elliptischen Kurven zu studieren. Eine klassische Probleme in der Kryptographie Bereich. Nach dieser Studium geben wir eine Umsetzung von einer neue Algorithmus, die in dem ECDLP basiert ist. Später vergleichen wir die Ergebnisse mit dem Bitcoin Abbauen algorithmus und wir definieren Metrische für das neue Algorithmus. Zum Ende stellen wir zum Verfügung eine Interface und ermutigen wir das Anstellung dieser Algorithmus in die Zukunft.

Acknowledgements

Before anything else, I would like to thank my supervisors Dr. Eloi Puertas and Adriana Moya for their support, ideas and invaluable experience provided during this work. Without them this wouldn't have been possible.

I would also like to thank first my parents for their inconditional support during the whole process together with my friends who also supported and listened my frustrations this last semester.

Contents

1 Introduction **1**
1.1 Objectives . . . 2
1.2 Structure of the work . . . 3

2 Cryptocurrencies and Blockchains **5**
2.1 Fundamentals . . . 5
2.2 Transactions . . . 6
2.3 Blockchain . . . 8
2.4 Consensus . . . 9

3 Cryptography **13**
3.1 Private key cryptograhy . . . 13
3.2 Public key cryptograhy . . . 14
3.2.1 Digital signatures . . . 15
3.3 The Discrete logarithm problem . . . 16
3.4 Elliptic curves . . . 17
3.4.1 The Elliptic curve discrete Logarithm problem (ECDLP) . . . 20
3.4.2 Elliptic Curve Digital Signature Algorithm (ECDSA) . . . 21
3.4.3 Security of Elliptic curve cryptography . . . 24
3.5 Hash functions . . . 27

4 Bread pudding PoW protocols **29**
4.1 PoW and Bread pudding protocols . . . 29
4.2 An ECDLP Bread pudding PoW model . . . 30
4.3 An implementation . . . 35
4.3.1 Details of implementation and problems . . . 36
4.3.2 Efficiency . . . 39

5 Results **43**
5.1 Results . . . 43

5.2 Benchmarks 45
5.3 Discussion 46

6 Conclusions and future work **49**

Bibliography **51**

Appendix **53**

Chapter 1

Introduction

Bitcoin is the biggest cryptocurrency in the world since its appearance back in 2009 [19]. Since its creation bitcoin and other cryptocurrencies have experienced an exponential growth to the point that even governments are taking part in them. One example of this is the state of el Salvador which accepted bitcoin as an official currency back in September of 2021.

According to the Cambridge Center for Alternative Finance, Bitcoin currently consumes around 110 Terawatt Hours per year. This is around 0.55% of the global electricity production and even more than some developed countries such as Finland consume. Without getting into ethical debates of how much a digital cryptocurrency should consume, it is becoming day by day more apparent that eco-friendlier and more sustainable cryptocurrencies should emerge.

These disrupting technologies, given its decentralized nature, usually rely heavily on *hash functions* in order to achieve consensus across all the participants of the network. These functions have the property that they are extremely hard to invert and very fast to compute. Very importantly, they do a big amount of bitwise operation that don't carry any meaning in them after all.

The consensus mechanism used by Bitcoin, known as Proof of Work (PoW) is the stage where cryptocurrencies spend most of their required energy consumption to function. In the last years alternatives have appeared to PoW as a way to achieve consensus. Examples of such are *Proof of stake* (PoS) or *Proof of space* (PoSpace) and they have been already found in successful cryptocurrencies such as *Nano*[1]. These alternatives do solve the problem of energy consumption unfortunately they don't offer the robustness and decentralization that PoW does, and

[1]Nano is a new generation cryptocurrency based in Proof of stake.

they have been seen to carry some vulnerabilities with them [2].

1.1 Objectives

Given that this is a bachelor thesis in Mathematics and Computer science we find ourselves in a perfect position to study cryptocurrencies from both perspectives with an eye in the issue with electric energy consumption:

Studying cryptocurrencies from a mathematical point of view means mostly talking about cryptography, thus we impose ourselves the goal of studying profoundly cryptography with an emphasis in elliptic curve cryptography. This type of cryptography apart from being the basis of cryptocurrencies, will be important given the implementation we will propose later.

As seen previously, Proof of Work based cryptocurrencies have the downside of big energy consumption. For this reason we imposed ourselves also the goal of studying alternatives to the nowadays used PoW algorithms and to try to provide a better, or at least more sustainable alternative to such.

As we found out at the very beginning of this work there are some types of special PoWs, the so called *Bread pudding* Proof of Work. These are Proof of Work that search a sense utility when doing them, hence are more sustainable than the usual ones. After we found out about them, a core goal became to study an actual proposal, its feasibility, and to try to provide an actual implementation of one such PoW. We also had in mind the objective of giving an interface to this implemented PoW so it could be used in future cryptocurrencies easily.

In the quest of one possible Bread pudding PoW to put our atention to, Adriana Moya proposed me a novel paper from 2020 [18]. In this paper the authors Alessio Meneghetti, Massimiliano Sala and Daniele Taufer propose a Proof of Work based in solving one of the two most studied problems in cryptography, The *Elliptic curve discrete logarithm problem* (ECDLP). This problem together with the problem of integer factorization are the classical problems that cryptography builds upon. The authors stated that they implemented the algorithm although in *Magma* a high level computer algebra system[3]. Given our goals of giving a

[2]Three Attacks on Proof-of-Stake Ethereum `https://eprint.iacr.org/2021/1413.pdf`

[3]Magma is a computer algebra system designed to solve problems in algebra, number theory, geometry and combinatorics.

real alternative to the usual PoW's, we defined the clear goal of making an in-depth study of this algorithm. Mathematically speaking, this meant analyzing the efficiency of it and also studying the ECDLP and its robustness. And then programmatically, by trying to give an implementation in a low-level cross compiled programming language, trying to replicate the results that the authors in the orginal paper obtained.

1.2 Structure of the work

This work is divided in four big chapters. In the chapter 2 of this work we will explain how cryptocurrencies and blockchain function. We will study the data structures these technologies use and given its decentralized nature, we will study the mechanisms they utilize for establishing order in the network. As already commented in the motivation we will dive deep in the concept of consensus. This chapter is useful and necessary for giving us the background we will need from the chapter 4 and onwards.

In the chapter 3 we make an overview of cryptography, first with a more informal approach and later introducing the classic cryptographic problem of the *Discrete logarithm problem* before jumping on to the study of elliptic curves. There we formalize the notion of the *Elliptic curve group* in order to finally study the *Elliptic curve discrete logarithm problem* passing by an overview of different signing schemes. This chapter is approached with a mathematical mindset, giving definitions and proofs and a sense of completeness to all the statements we make.

In chapter 4, we begin by giving mathematical definitions to the concept of *Proof of Work* and its associated refinement *Bread pudding Proof of Work*. We then embrace the PoW model from Alessio Meneghetti, Massimiliano Sala and Daniele Taufer [18] as a possible implementation of one such *Bread pudding Proof of Work*. We start by making an overiew of the protocol, then explaining the implementation we made with its associated caveats and efficiency.

In the chapter 5 we expose the results we obtained from the implementation of the previous chapter. We also compare our results with the metrics that bitcoin PoW has. We then finally discuss the results highlighting the strenghts and weaknesses we found.

Finally, in the last chapter we conclude our work summarizing all that we have seen and giving an overview of the main takeaways from the implementation of the algorithm we have implemented in the previous sections. In this direction we also give the future work that is still left to be done and the questions that remain unanswered.

Chapter 2

Cryptocurrencies and Blockchains

Digital cash solutions also known as cryptocurrencies emerged in 2009 as a alternative to regular FIAT cash after the publication of the infamous paper of Satoshi Nakamoto “Bitcoin: A Peer-to-Peer Electronic Cash System” [19]. In this chapter we first begin by making an overview from a computer engineering perspective how these digital solutions work, we will be basing us in the general model that Satoshi Nakamoto proposed which applies to all posterior blockchain technologies. Across all this chapter we will be refering to cryptocurrencies specifically, because they are a use case of blockchain techonologies but the fundamentals remain the same.

2.1 Fundamentals

All digital cash solutions are based on the idea that instead of relying in a central trusted authority which usually are banks and institutions and are located in servers, we now rely in cryptography as a means of assuring the security of the coin, and not in a central authority.

All digital coins have at least three basic elements:

Private Key, Public Key and **Coin Address** we refer the reader to the chapter on cryptography (chapter 3) in order to gain insight into what the Private key and the Public key represent, in this chapter we will explain the philosophy behind these abstractions.
The Coin address is formed from the Public Key and is what appears as the "recipient" of the sent funds. It would be equivalent to the *IBAN* but in digital cash solutions. The address is usually formed by applying a hash function to the public key. For example in Bitcoin it is calculated as shown below:

$$Base_{addr} = RIPEMD160(SHA256(K_{pub})$$

as we see the the base of the address is 160 bit-long then the final address would be the concatenation of the a checksum calculated using SHA256 hash function and the first 4 bytes of $Sha256(Sha256(Base_{addr}))$ and $Base_{addr}$.[1]

Observe that as always the application of hash functions from a bigger space to smaller (of finite cardinality) can produce collisions. In this case of course it is not different. If two different public keys generate the same Coin address then both users of the coin would be able to spend the money of the other person. This scenario still, is highly unlikely. For instance taking the most used cryptocurrency nowadays, Bitcoin. And thus the most probable of having already collisions we have that as of October of 2021 the Blockchain is 350Gbytes in size and if we assume that the entire blockchain are diferent addresses then we would have roughly 2^{41} adresses. This space, compared with the space of possible addresses is around 2^{160} following the contruction above. Thus the probability of collision can be aproximated using the formula

$$p \approx \frac{n^2}{2m}$$

In bitcoin's case:

$$p \approx \frac{2^{82}}{2^{161}} = \frac{1}{2^{79}}$$

which is negligible and we were still very genereous with the number of Bitcoin accounts.

2.2 Transactions

Transactions in digital cash solutions represent the abstraction of transferring money between parties. The general data structure of a transactions is as follows:

[1]This address construction can vary more or less depending on the version of Bitcoin

Size	Field name	Description
4 bytes	Version	Which version the transaction follows
1-9 bytes	Input counter	How many inputs are included
Not fixed	Inputs	the transaction inputs
1-9 bytes	Output counter	How many Outputs are included
Not fixed	Outputs	The transaction outputs
4 bytes	Locktime	Timestamp for being added to the blockchain

Table 2.1: Transaction data structure of Bitcoin

The inputs and outputs represent the coins which are going to be transferred. And this is managed by what is known as *unspent transaction output*, or UTXO. Intuitively these are funds that are lock to a specific owner (that is a specific Coin address) and then recognized by the network so can be spent in another transaction. The structure of the input and output are as follows.

Size	Field name	Description
32 bytes	Transaction hash	Pointer to the transaction containing the UTXO
4 bytes	Output Index	The index number of the UTXO referenced
1-9 bytes	Unlocking Script size	The lenght of the unlocking-script in bytes
Variable	Unlocking Script	A script that unlock the locking script
4 bytes	Sequence number	disabled

Table 2.2: Input data structure of Bitcoin

Size	Field name	Description
8 bytes	Amount	Bitcoin value in satoshi (10^{-8} bitcoin)
1-9 bytes	Locking-Script size	Locking-Script legth in bytes
Not fixed	Locking-Script	A script defining the conditions needed to spend the output

Table 2.3: Output data structure of Bitcoin

As we can see the table 2.3 and 2.2 we have the parameters Locking script and Unlocking script. They refer in real world usage usually to the signature and public key used to verify the signature. Still more complex and more than one signature method can be employed.

In a real use case where Alice wanted to send to Bob say 0.1 bitcoin we would have as the locking script, usually called a *ScriptPubKey* the public key of the bitcoin sender. Then as the unlocking script we would have, the usually called *scriptSig*, the signature of the transaction with the private key of the sender, that is Alice. Then every bitcoin client would validate transactions by executing the locking and unlocking scripts together. Still statement may sound a little vague but it will become clear in the chapter 3 on cryptography.

As an additional information and not relating to this work note that there's no need to restrain ourselves in these scripting functions. More complex locking and unlocking scripts could be used. This gave birth to smart contracts which are just digital assets getting exchanged based on more complex conditions than the usual signature base model.

2.3 Blockchain

One natural issue that one encounters when designing a currency is to be sure that funds can't be double spent by the same person. This situation with physical money doesn't make much sense but with digital cash it does. Logically the first transaction that appeared is the one that counts and after this one all the successive ones should be incorrect. Thus a cryptocurrency has to be aware of all the transactions that have been made in order to be sure if a coin has been already spent by some participant.

In decentralized currencies as we don't have any trusted authority who does this work for us, the transactions have to be publicly announced and this highlights that we also need a system that creates consensus between all the participants of which transactions have been proposed and accepted by the network.

Blockchain and the more general concept *distributed ledger* [2] are the abstractions that create this shared transaction record across the network. Blockchain for instance refers to the chain that is formed by joining the blocks, which are basically a set of transactions and a timestamp, by the hash of the previous accepted block. Then the next block will include the hash of this block and so on. We shall focus more in the concept of blockchain because the algorithm we present in chapter 4

[2] Blockchain is usually considered just one type of distributed ledger

is thought as a blockchain. The data structure of a block is presented below:

Size	Field name	Description
4 bytes	Block size	The size of the block
80 bytes	Block Header	Header of he block
1-9 bytes	Transaction Counter	The number of transactions that come after
Variable	Transactions	The transactions themselves

Table 2.4: Block data structure of Bitcoin

The blockchain then would be graphically:

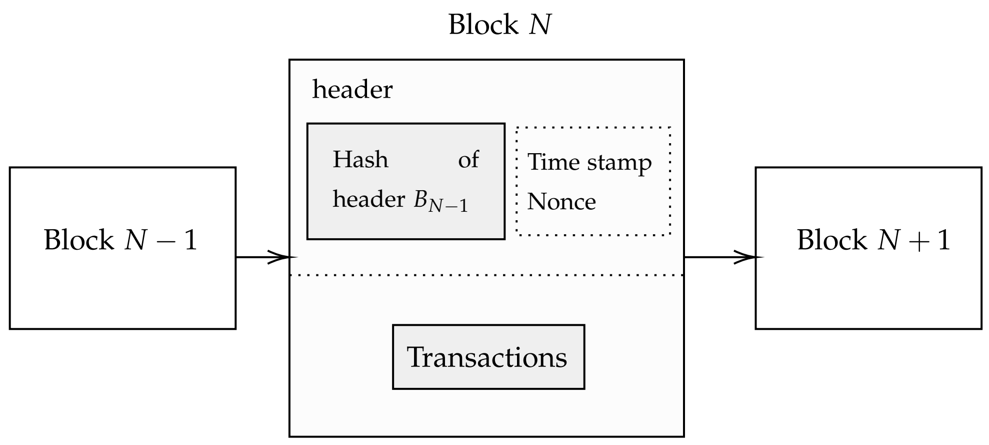


Figure 2.1: A general blockchain structure

Remark 2.1. In a public blockchain[3] all the transactions and PoW solutions are available to everyone, that means a public blockchain can be thought of as a public database. This remark although maybe obvious interests us as we will see later.

2.4 Consensus

In anticipation to deciding which transactions are included in and also who constructs the next block in the blockchain we are in the need of a method to achieve consensus in the network. A Proof of Work (PoW) is usually the approach

[3]A public blockchain has no access restrictions. Anybody with an Internet connection can participate in it. Either by actively mining or passively by sending transactions.

here although other alternatives exist. A Proof of Work is informally a cryptographic proof in which one participant (the prover) proves to others (the verifiers) that some amount of computational effort has been expended, the verifiers should be able to prove this work has been done easily.

Since the pioneer paper of Satoshi Nakamoto proposed a method for achieving consensus through the use of hash functions, all the blockchain have been following his model. This method is really simple and consists of applying a *hash function* over the block data being added to the database (Blockchain). It is indeed added when this hash output is small enough as stated in the network configuration. The nodes have to play with a variable called *Nonce* inside the block header until this acceptance condition is met.

This process of getting new blocks added in the blockchain is usually named *mining* and the nodes who attempt to mine are called *miners*.

In bitcoin for example it is used the hash function *SHA256* two times searching for a small enough result:

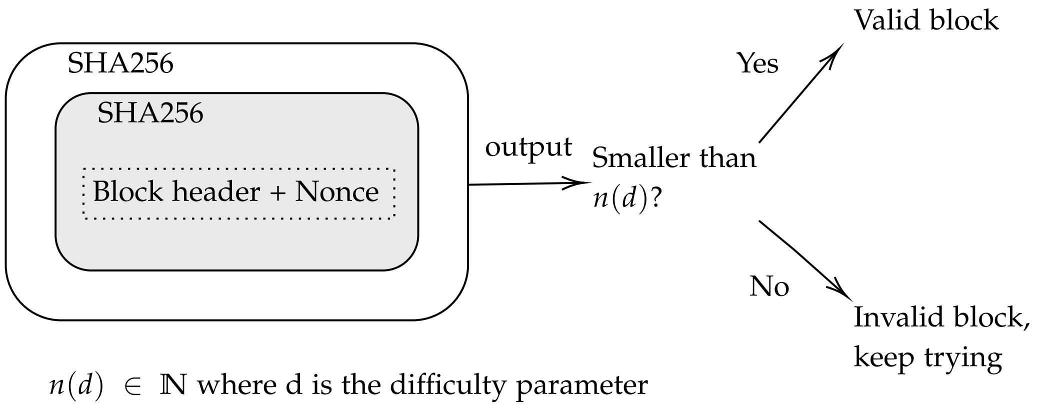

SHA256
SHA256
Block header + Nonce
output
Smaller than $n(d)$?
Yes
Valid block
No
Invalid block, keep trying
$n(d) \in \mathbb{N}$ where d is the difficulty parameter

Figure 2.2: Mining diagram of bitcoin

We outline below how a cryptocurrency operates in order to incorporate a transaction since its creation until the network accepts it:

1. A node broadcasts a new transaction to all the nodes
2. Each node that is mining collects the transactions into a block
3. Each miner tries to solve a Proof of Work

4. A miner finds a nonce which solves the problem and then broadcasts the block to all nodes
5. Nodes accept the block if all the transactions inside it are valid.
6. Nodes start working in a new block following the next accepted.

As can be seen from Table 1.4 the number of transactions that can be included inside a Block is limited and thus the miner nodes have to decide which transactions to include in the next block.

Fees, Incentives

Miners in order to actually want to mine in the original Bitcoin protocol recieve a reward or incentive in the form of the coin itself. The Incentives are usually of two types:

New coin creation: In this case the first transactions in a block is a special one that creates/"mines" a new coin to the miner of that block.

Fees payed by the transactions senders: In this case the nodes that want to get their transactions in the network usually leave a number of inputs bigger than the number of outputs. Then by default the protocol assigns this unallocated difference to the miner of the block containing that transaction. Thus makes sense that the miners usually choose the transactions with higher fees as the transactions to include in the mined block.

Chapter 3

Cryptography

Cryptography comes from the Greek *Kryptos* which means hidden and is the science which studies protocols that prevent third parties or the public from reading private messages.

In the history of cryptography there are two marked periods. The first, before the invention of the computer where cryptography was mostly the application of substitution and translation ciphers. And later, with the invention of computers, where newer and more sophisticated ciphers appeared based in the developing mathematical field of number theory. We will focus more on the later one.

Modern cryptography can be categorized in two main categories: Private key cryptography also known as asymmetric cryptography and public key cryptography also known as symmetric cryptography.

Let us now asume we have two individuals, for example Bob and Alice who want to communicate through a hostile medium. Let's see how each type of cryptography tries solves the problem of communication.

3.1 Private key cryptograhy

In Private key cryptography the premise is that Alice and Bob and just the two of them share a key which we denote as K which can be used to encode messages from the space of possible messages M and also decode messages from space of ciphers denoted C.

Thus the encoding/encrypting function can be seen as the following:

$$e\colon K \times M \to C$$

Where the domain are the pairs of possible keys with the possible messages to encode. The range are the possible ciphertexts which the enconding function results on. Similarly the decoding/decrypting function can be seen as:

$$d \colon C \times K \to M$$

This functions must share the property that they are one the inverse of the other one. That means intuitively that the decoding function applied over the encoding function with the same key must return the same encoded message:

$$d(k, e(k, m)) = m \text{ for all } k \in K \text{ and all } m \in M$$

It is important to comment that the security layer in this encryption scheme resides exclusively in the key which should be only known by the encoder and decoder. It has to be asssumed that the methods of encoding (and thus decoding) are known by thirds. This principle is known as *Kerckhoffs principle*.

3.2 Public key cryptograhy

In Public key Cryptography or PKC we have as before the space of possible keys K, the space of messages M and the space of ciphertexts C. The difference with the previous type of protocols is that now an element in the space of keys is actually a pair of keys (k_{priv}, k_{pub}). The first one is called the private key and the second one is the public key. The operating mode with these keys is the following:

1. For each public key k_{pub} there's its corresponding $e_{k_{pub}} \colon M \to C$ encryption function
2. For each private key k_{priv} there's its corresponding $d_{k_{priv}} \colon C \to M$ decryption function
3. If $(k_{priv}, k_{pub}) \in K$ then $d_{k_{priv}}(e_{k_{pub}}(m)) = m$ for all $m \in M$

As we can imagine from the three conditions above, the ecryption function given a key should be easy and fast to compute but getting the inverse without the private key should be really hard. This is known as a *one-way trapdoor function* [11] pag. 63. Generally a one way function is an invertible function which is easy to compute but has an inverse which is very difficult to compute. The term difficult here is not rigorous but the idea is that computing the inverse should be complex enough, requiring a computation equal to the age of the earth for example.

Secure PKC's are built using one-way trapdoor functions. where k_{priv} acts as the backdoor for the encoding function d_{pub} and similarly k_{pub} acts as the backdoor for the decoding function k_{priv}.

As a note the reader might be surprised to know that there's still no proof on the existance of such one way functions as described earlier and in fact proving the statement is equivalent to proving the famous $P = NP$ problem. [11]

3.2.1 Digital signatures

Digital signature refers to the process of giving authenticity to a message sent across the network. That is, proving that the message has not been sent by someone else faking the sender identity.

The usual procedure when working with signatures is as follows:

1. The sender produces the signature S of a document M using a signing algorithm
2. The receiver applies a verification algorithm which given M and S returns TRUE if the message is authentical and FALSE otherwise

Logically only the sender has to be able to produce valid authentical messages refering to him.

Digital signatures are usually approached using asymmetric cryptography. Necessary conditions that a secure digital signature scheme must have are the following:

1. Given k_{pub} an attacker can't determine the corresponding k_{priv} nor produce another key that gives the same signatures as k_{priv}
2. Given k_{pub} and a list of signed documents $D_1, ..., D_n$ with their corresponding signatures $S_1, ..., S_n$, an attacker cannot feasibly determine a valid signature on any document D that is not in the list

Some comment about the second property, it refers to the intuitive fact that if we have some documents and its signatures the attacker cannot determine any pattern in order to deduce signatures of other documents. Very similar documents should not produce very similar signatures.

Digital signatures as we have presented them here are a very big simplification of what real life solutions look like. As we should study them in a rigorous mathematical way the formal way we have presented them is already good.

Still we would like to comment that usually signatures are applied not to the documents entire data, because of the big size they might have. Usually a in between step is added where a *digest function* such as a hash function is applied. The reason for this is the output being much smaller than the input. As we saw with the problem of collision in address creation in cryptocurrencies the problem of collision is usually negligible.

3.3 The Discrete logarithm problem

The discrete logarithm problem is a mathematical problem which arises in many situations and can be used to our advantage in secure cryptographic systems.

The first publication in PKC was due to Diffie and Hellman and is based in the discrete logarithm problem.

Let's start our study of the discrete logarithm problem in a formal mathematical way:

Definition 3.1. *$\mathbb{Z}/n\mathbb{Z}$ where $n \in \mathbb{N}$ denotes the ring equiped withthe natural sum and multiplication of the integers modulo n.*

From now on we shall refer to $\mathbb{Z}/n\mathbb{Z}$ as simply $\mathbb{Z}_n$

Lemma 3.2. *Given p a prime number then $\mathbb{Z}_p$ is a field and we denote it $\mathbb{F}_p$.*

Lemma 3.3. *Let p be a prime, then the multiplicative group of $\mathbb{F}_p$ is cyclic of order $p-1$.*

Note that this actually implies Fermat's little theorem. We give no proof to this lemma's as they were seen already in the bachelor classes.

Definition 3.4. *Let G be a group, we say $g \in G$ is a primitive root of G if for all $x \in G$ there exists $h \in \mathbb{N}$ such that $g^h = x$.*

Remark 3.5. If G is a cyclic group then it has a primitive root by definition of cyclic group.

Definition 3.6. *Let g be a primitive root for $\mathbb{F}_p$ where p is prime and let h be a non zero element of $\mathbb{F}_p$. We denote as the Discrete Logarithm Droblem (DLP) the problem of finding an exponent $x \in \mathbb{N}$ such that*

$$g^x \equiv h \mod p$$

Thus the discrete Logartithm problem is the problem of finding the *index* in Group theoretic terms. Assuming we are in $\mathbb{F}_p$ we can then define a function log_g depending on its associated primitive root g assigning, given an element x in $\mathbb{F}_p$, the index of that point.

$$log_g : F_p \to \frac{\mathbb{Z}}{(p-1)\mathbb{Z}}$$

The following calculation should give the reader the impression that the name of such function has some meaning behind it:

let $a, b \in \mathbb{F}_p$; $log_g(a) + log_g(a)$ by definition is the natural number $h = h_a + h_b$ such that $g^{h_a} = a$ and $g^{h_b} = b$ thus multiplying we have $g^{h_a}g^{h_b} = ab$ so $g^{h_a+h_b} = ab$ and thus $h_a + h_b = log_g(ab)$. We have proved then that:

$$log_g(ab) = log_g(a) + log_g(b) \tag{3.1}$$

Resembling the distinctive property of the logarithm.

Of course we don't have to limit ourselves to $\mathbb{F}_p$, indeed we can further generalize the discrete logarithm problem to a Group G. In general the only condition that seems we have to ask is the group to be cyclic. Still if a Group G weren't cyclic we could define a discrete logarithm problem basing us just in the Base point and multiples of it.

As we will see next we can study geometrical objects such as elliptic curves with the structure of a group and work in the DLP over that group.

3.4 Elliptic curves

Definition 3.7. *We define an elliptic curve as the set of solutions to a equation of the form* $Y^2 = X^3 + AX + B$ *satisfying* $4A^3 + 27B^2 \neq 0$

Remark 3.8. By imposing that $4A^3 + 27B^2 \neq 0$ we force to the curve to not have singular points, ie. the curve is smooth.

Observe that this definition is very general as we don't restrict the coefficients A, B of the equation in any set and we also don't specify where the solution should reside in, we will see where this coefficients should reside in a constructive way.

Remark 3.9. If $P = (p_1, p_2)$ is a point in the Elliptic curve then $Q = (p_1, -p_2)$ also lies in the elliptic curve

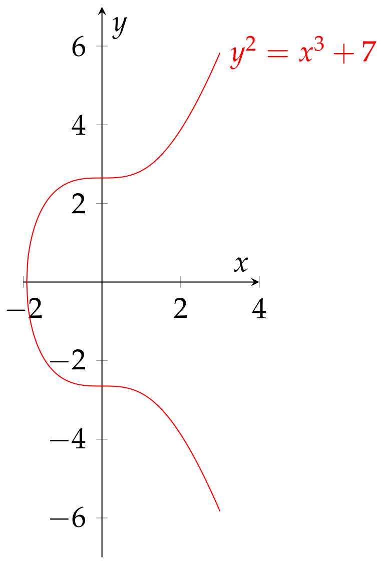


Figure 3.1: Elliptic curve $y^2 = x^3 + 7$ over $\mathbb{R}$

We shall now give to this set a addition-like operation. The way we construct such "+" operation is geometrical:

Let $P, Q \in \mathbb{R}^2$ such that $P \neq Q$ we can consider the segment joining these two points. We construct the sum point M of these two by finding the third intersection point of this line with the elliptic curve and then making a reflection in the X-axis. That is multiplying the y-component by -1.

With this sum operation defined and the very general definition of elliptic curve given several problems arise:

1. If the operation is well defined
2. If the operation is closed
3. What happens if we want sum $P + P$
4. What happens if we want to sum two X-axis reflected point $P = (p_1, p_2)$ and $Q = (p_1, -p_2)$

Lets study each one of the problems in order to refine first the sum "+" operation given and second the kind of algebraic structure the set of solutions and coeficients have to be in.

We begin considering the second problem. We can consider the line joining P with itself as a limit like situation, thus the tangent line makes sense and so we consider it as the line to search the new intersection with the elliptic curve and then make the reflexion with the X-axis.

In the third poblem we have that the line associated is vertical ie. $X = c$, c a constant and thus the third point does not exists because we substituing in the equation of the elliptic curve we have $Y^2 = c^3 + Ac + B$ which has just two solutions. The solution to this issue is to add an extra point in the plane O resembling the infinity of projective spaces such that O is the the solution of this sum. The question now turns: How is a generic plane point P added with O? If we imagine it as the infinity lying in (p_1, ∞) then we can construct the line joining the two, $X = p_1$ and finally the other intersecting point which is $(p_1, -p_2)$ after as defined above we reflect it in X-axis giving $P + O = Q = (p_1, p_2) = P$. With this construction, the sum of any point P with the point O is equal to P. So O is the identity element of this sum operation.

For the first problem we study the new intersection point explicitely:

Let $P = (x_p, y_p), Q = (x_q, y_q)$, $P \neq Q$ two points in the set of solutions of the elliptic curve. We have that the slope of this segment joining the two is $s = \frac{y_q - y_p}{x_q - x_p}$. In order for $(x_q - x_p)^{-1}$ to be well defined, the set of solutions **has to be a field**. If we consider the line containing P and Q as $y = sx + d$ then substituting in the elliptic curve equation gives:

$$(sx + d)^2 = x^3 + ax + b \tag{3.2}$$

We also know that the solution to this polynomial must be X_p, X_q, X_r where R is the third point and x_r its x-component. Thus 3.2 can be rewritten as:

$$(x - x_p)(x - x_q)(x - x_r) = x^3 + x^2(-x_p - x_q - x_r) + x(x_p x_q + x_p x_r + x_q x_r) = 0$$

Comparing the coefficients of x^2 we arrive to the explicit expression of R:

$$\begin{cases} x_r = s^2 - x_p - x_q \\ y_r = y_p + s(x_r - x_p) \end{cases}$$

From the explicit expression we have that the operation is well defined if and only if the set of the coefficients is contained in the set (and field) of the solutions.

From now on we will refer to elliptic curves as elliptic curves over a field $\mathbb{K}$ in the sense that the coeficients A, B from the Weierstrass equation belong to the field and also the set of solutions belong there.

Theorem 3.10. *The set of the solutions E of an elliptic curve together with the operation "+" as defined in the paragraphs above forms a Group.*

Proof. By the construction of the sum operation we have the existence of inverses because given P we have that $-P$ belongs to the curve. Also, the commutativity and the existence of a identity element are immediate by construction. The closeness of the operation over the field has been seen explicitly. It reamins to be proved just that the operation has the associativty property. For the sake of brevity a proof is not posed here but the reader can find one in [9]. □

Definition 3.11. *Let $p \geq 3$ a prime. An elliptic curve over $\mathbb{F}_p$ is an equation of the form*

$$E : Y^2 = X^3 + AX + B \textit{ with } A, B \in \mathbb{F}_p \textit{ satisfying } 4A^3 + 27B^2 \neq 0$$

The set of points on E with coordinates in $\mathbb{F}_p$ is the set

$$E(\mathbb{F}_p) = \{(x, y) : x, y \in \mathbb{F}_p \text{ satisfy } y^2 = x^3 + Az + B\} \cup \{O\}$$

Theorem 3.12. *Let E be an elliptic curve over $\mathbb{F}_p$ and let P and Q be points in E($\mathbb{F}_p$) then (E($\mathbb{F}_p$), +) is a group.*

Proof. Just a special case of 3.10 considering the coefficients and solutions to be in F_p. □

3.4.1 The Elliptic curve discrete Logarithm problem (ECDLP)

In this subsection we shall study again the discrete logarithm problem but now restraining ourselves to the group $E(\mathbb{F}_p, +)$. Remember that at the end of section 3.3 we saw that a necessary and suficient condition for having a well defined Elliptic curve discrete logarithm problem was to have a cyclic group.

Definition 3.13. *Let E be an elliptic curve over the finite field $\mathbb{F}_p$. The Ellitptic Curve Discrete Logarithm Problem (ECDLP) over P is the problem of finding an integer n such that $Q = nP$ for a given $Q \in E(\mathbb{F}_p)$.*

By analogy with the general discrete logarithm problem of section 3.3 we denote this integer n by:

$$n = log_P(Q)$$

and we call n the elliptic discrete logarithm of Q with respect to P.

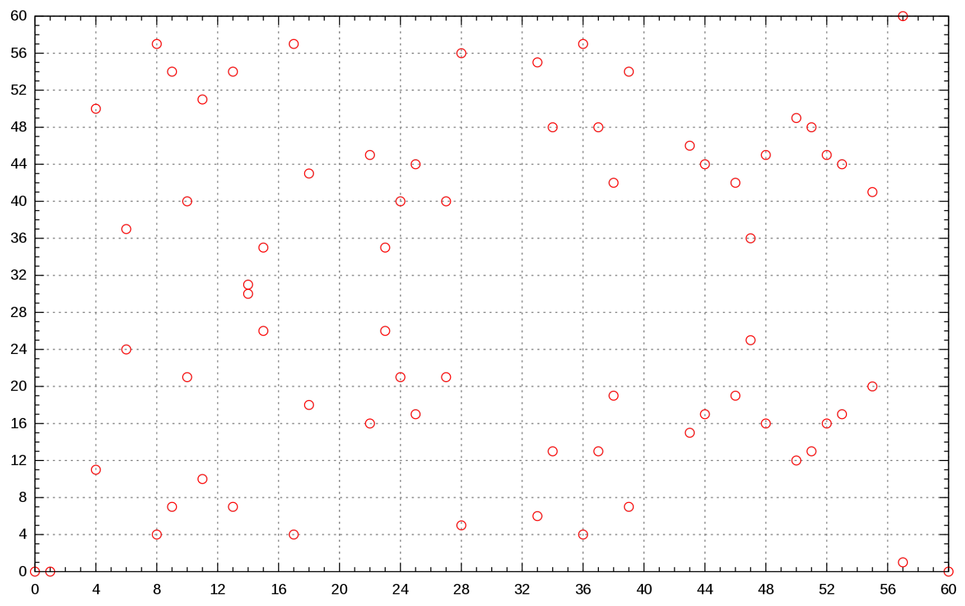


Figure 3.2: $y^2 = x^3 - x$ over $\mathbb{F}_{61}$ source: Wikipedia

Remark 3.14. In the definition above we don't require P to be a primitive root. Thus there can be cases where elliptic discrete logarithm doesn't exists. Still in practical terms we actually we don't need to have a primitive root. In the order of things Alice and Bob would decide over a P in $E(\mathbb{F}_p)$ and then decide their corresponding secrets as $Q = nP$. In this way, by construction, the existence of the elliptic discrete logarithm is guaranteed.

Remark 3.15. As we also commented in 3.3 one solution to the elliptic discrete logarithm is the index s of the Q in the base P over $E(\mathbb{F}_p)$. So it means that actually the elliptic discrete logarithm is not unique as defined. But if we consider the solutions of the problem to be defined in $\mathbb{Z}_s$ then yes.

Proposition 3.16. *$log_p : E(\mathbb{F}_p) \to \mathbb{Z}_s$ as defined above is a group homomorphism*

Proof. the fact that is well defined follows from the above reasoning and from (3.1) it follows easily that it is a homomorphism. □

3.4.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

We will present here the cyptographic foundations for signing transactions. As we have seen with the discrete logarithm problem in 3.3 we can begin the study in a more general way and then move to the Elliptic curves group.

Elgamal scheme

DSA stands for Digital Signature Alforithm and it backs to the 80's. We will start by describing a previous and simpler signature algorithm based on the Elgamal scheme[1].

Suppose Alice wants to sign a certain message M. Then she would choose a large prime p and a primitive root g modulo p. Next Alice would choose a secret signing exponent a and would calculate:

$$A \equiv g^a \bmod p \tag{3.3}$$

Then a together with p and g form Alice public key as we will see in a moment. Alice now wants to sign a digital document say D, where D has to be an integer satisfying $1 < D < p$. If the document is bigger it can the be reduced by appying modulo p over it. Alice would choose a random element $1 < k < p$ that satisfies $gcd(k, p-1) = 1$ and compute the following two quantities:

$$S_1 \equiv g^k (\text{mod p}) \tag{3.4}$$

and

$$S_2 \equiv (D - aS_1)k^{-1} (\text{mod p-1}) \tag{3.5}$$

Then the signature of Alice would be the pair (S_1, S_2). How can Bob check that only Alice, or anyone with the private key, can have signed the document D?

Lemma 3.17. *$A^{S_1} S_1^{S_2} \equiv g^D$ (mod p) Where A is as defined in 3.3 and S_1 and S_2 as defined in 3.4 and 3.5 respectively.*

Proof. By a simple computation we have:

$$A^{S_1} \cdot S_1^{S_2} \equiv g^{aS_1} \cdot g^{kS_2} \equiv g^{aS_1 + kS_2} \equiv g^{aS_1 + (D - aS_1)} \equiv g^D \pmod{p}$$

□

As we have already seen if Bob or anyone can solve the discrete logarithm problem 3.3 then the signature makes no sense, but as we have commented there are not practical ways of attacking this problem. Still, it is not entirely clear if someone could forge a signature of a document without knowing the private key

[1]The Elgamal scheme is a asymmetric key encription algorithm described by Taher elgamal in 1985

a.

Given only the public key A defined in 3.3 the problem can be postulated as finding integers x, y which satisfy:

$$A^x \cdot x^y \equiv g^D \pmod{p} \tag{3.6}$$

taking log_g we obtain:

$$\log_g(A)x + y\log_g(x) \equiv D \pmod{p-1} \tag{3.7}$$

As of today the modular equation 3.7 has not a better solution than actually tacking the discrete logarithm problem 3.3.

As an additional commentary, Elgamal signatures (S_1, S_2) consist of one number modulo p and the other one modulo $p-1$. Usually to be secure against brute force attacks, also known as index calculus attacks the prime p is taken to be between 1000 and 2000 bits.

DSA

The *Digital Signature Algorithm* (DSA) actually works in a subgroup of $\mathbb{F}_p^*$ of prime order q and thus shortens the signature.

We explain in the same situation as before how the DSA works:

Alice would begin taking primes p and q so that:

$$p \equiv 1 \pmod{q}$$

Then she would choose an element $g \in \mathbb{F}_p^*$ of exact order q. Note that this is an easy task given a primitive root of the group we are working with. Then Alice would choose his secret, or private key, namely a and she would compute

$$A \equiv g^a \pmod{p}$$

Now the public key of Alice is the triplet (p, q, g). We are ready now to begin the signing process itself:

Suppose that Alice wants to sign a digital document D, where D is a integer satisfying $1 \leq D < q$. She would choose a random element k such that $1 < k < q$ and she computes:

$$S_1 \equiv (g^k \mod p) \mod q \text{ and } S_2 \equiv (D + aS_1)k^{-1} \pmod q \tag{3.8}$$

As we can see there are lots of similarities between the Elgamal scheme 3.4.2 and the DSA in how the signatures are constructed. The only difference is in the construction of S_1 where first we apply a modulo p and then modulo q. Bob would check the signature by computing:

$$\begin{aligned} V_1 &\equiv DS_2^{-1} \pmod q \\ V_2 &\equiv S_1 S_2^{-1} \pmod q \end{aligned}$$

And he would finally check if:

$$(g^{V_1} A^{V_2} \pmod p) \pmod q \text{ is equal to } S_1$$

Proof.

$$\begin{aligned} g^{V_1} A^{V_2} \pmod p &\equiv g^{DS_2^{-1}} - g^{aS_1 S_2^{-1}} \pmod p \\ &\equiv g^{(D+aS_1)S_2^{-1}} \pmod p \end{aligned} \tag{3.9}$$

Which is equal to $g^k \pmod p$ □

ECDSA

The ECDSA[2], *Elliptic Curve Digital Signature Algorithm* is similar to the DSA signing algorithm. The construction is analogue. Now instead of working over $\mathbb{F}_p^*$ we are working in an elliptic curve $E(\mathbb{F}_p)$ and thus we are in an additive group instead of the multiplicative one of the DSA.

3.4.3 Security of Elliptic curve cryptography

Refering to Discrete logarithm problem:

"All that can be said is that such and such a problem has been extensively studied for N years, and here is the fastest known method for solving it" [11] pag. 323

[2]ECDSA is the signing algorithm that uses bitcoin [4]

As of now we still don't know the real difficulty of the general case of the discrete logarithm problem, currently the problem seems to NP-hard. Together with the fact that elliptic curves have not been found to have any other algebraic structure apart from the structure of a group. Makes elliptic curves perfect to work with discrete logarithm problems. That's the reason they form the basis of cryptography, as seen in ECDSA for example.

The best algorithm known to this day for the ECDLP is the **Pollard ρ-method** [15] in the modified version proposed by Gallant, Lamber and Vanstone, combined with Wiener and Zuccherato [15]. This algorithm takes $O((\sqrt{\pi n})/2)$ where n is the order of the base point. If we denote b the bit representation of n we have $2^b \approx n$. We then have $O((\sqrt{\pi n})/2) = O(2^{2b-1})$ thus exponential, there exist other algorithms that run in similar time such as the baby-step giant-step but Pollard-rho remains the best option as it only requires O(1) storage [5]. Later in this chapter we will expand this algorithm and give an overview.

Attacks

There are several special families of curves that allow us to break the ECDLP in more efficient algorithms that run in polynomial time. For example the Pohlig-Hellman algorithm allows us to reduce the determination of l, a solution of an instance of ECDLP, to the determination of l modulo each of the prime factors prime factors of who?.

Next if a curve has the property that $\#E(\mathbb{F}_p) = p$ where p is prime we say that the elliptic curve is *prime-field-anomalous*. For every *prime-field-anomalous* it has been shown that it is possible to contruct an isomorphism between $\#E(\mathbb{F}_p)$ and the additive group $\mathbb{Z}_p$. Given that the additive group of $\mathbb{F}_p$ is $\mathbb{Z}_p$ this structure of field allows to have a polynomial time algorithm for solving the ECDLP. This type of attack is called **Semaev–Smart–Satoh–Araki** attack.

Another attack uses the *Weil-pairing* to embeed the group $E(\mathbb{F}_p)$ in the multiplicative group of the field $\mathbb{F}_{q^k}$ for some natural k. In order to be able to produce this embedding is that n divides $q^k - 1$. The minimum natural k that does this is called the **embedding degree of the curve** E. Using this embedding it is possible to solve the ECDLP in subexponential running time.

The discriminant of complex multiplication of Ellitptic curves over finite fields (*CM discriminant of E*) refers to the discriminant of a field generated by adjoining the roots of the characteristic equation of the elliptic curve to $\mathbb{Q}$. These algebraic definitions are out of the scope of this work but a introduction of them can be found here [16]. It has been seen than one can speed up the pollard rho algorithm if the *CM discriminant of E* is small enough [22]. Still this possible attack is highly unfeasible because this discriminant is usually large for large enough elliptic curves [16].

Pollard rho algorithm

Next we give an overview of the Pollard rho algorithm to solve the ECDLP. Given a ECDLP over the elliptic curve group E of cardinal q, we want to find the integer n such that $nP = Q$. Let's asume that $E =< P >$. In the Pollard rho algorithm the goal is to solve the following problem:

$$\text{find integers } \ a, b, A, B \ \text{ such that } \ aP + nQ = AP + BQ \tag{3.10}$$

asuming we have (2.10) we can then obtain the solution to the ECDLP:

$$aP + bQ = AP + BQ$$

$$aP + bxP = AP + BxP \text{ where x is the ECDLP solution}$$

$$(a + bx)P = (A + Bx)P$$

$$(a - A)P = (B - b)xP$$

taking P out we have the following modular equation:

$$a - A \equiv (B - b)x \pmod q$$

$$x = (a - A)(B - b)^{-1} \pmod q$$

which is well defined if an only if $gcd((B - b), q) = 1$. In order to achieve the equation (2.10) the algorithm generates a pseudo-random sequence:

$$X_i = a_i P + b_i Q$$

As the set of possible outcomes is finite we will reach a loop in the sequence sooner or later, the downside with this approach is that we have to store all the sequence elements. We can reduce the algorithm to $O(1)$ space complexity in the following manner:

If we denote i, j $i \leq j$ the two smallest integers such that $X_i = X_j$ rewritting we have that $X_i = X_{i+k}$ for some natural k. We have the following theorem which we present without proof:

Theorem 3.18. *(Knuth (1997)). For a periodic sequence $X_1, X_2, ...$ there exists an integer $l > 0$ such that $X_l = X_{2l}$ and the smallest such l lies in the range $i \leq l \leq i + k$*

Using this theorem we can see that we can just store X_i and X_{2i} for each i until we find a colision. We then require only $O(1)$ storage complexity.

The algorithm has a complexity $O(\sqrt{q})$. We won't provide a demostration of this statement but the proof is very elegant and based on the birthday paradox, the reader can find the proof in the following article [20] by J. M. Pollard the author of the algorithm.

3.5 Hash functions

As we have seen one quality that cryptocurrencies and specifically blockchains rely on are functions which are easy and fast to compute but hard to invert. Such functions are called Hash functions. We make a formalization of such functions in order to give completeness to the work.

Generally a hash function $H : N \rightarrow M$ takes a message of n-Bits (N) and converts them to a message of m-bits (M). Usually we have that $m < n$. Such functions must have also the following properties:

1. $H(message)$ must be easy to compute.
2. Given $h \in M$ it must be difficult (exponential time) to find a n-bit long message n such that $H(n) = h$
3. They have to be collision resistant which means that it has to be two entries to the hash input which result in the same output.

A typical approach to construct such functions is as explained below:

Suppose we want to construct a hash function which takes a message of length m and converts it to a message of length n. These functions make use of mixing algorithms M that take sub messages of length say p and mix them to another message of the same length. Thus these algorithms first append some bits to the original message in order to make it divisible by p. If we denote D the original message with the necessary bits appended we have:

$$D = D_0 \,||\, D_1 \,||\, ... \,||\, D_k$$

where D_i are p-bit messages $\forall i \in 0, ..., k$. Then they start a iterative process where they construct $H_0 = M(D_0)$ and start iterating to construct $H_i = H_{i-1}$ xor D_i for all $i \in 1, ..., k$.

Nowadays the most common used hash functions are called *Secure hash algortithm (SHA)* in its several variations, each one of them differs from the others usually in the output size.

Chapter 4

Bread pudding PoW protocols

In this section we will take a deep look at Proof-of-Work protocols, fomalizing this concept basing us in the paper from M Jakobsson [12] where he first fomalized this concept and its variations.

4.1 PoW and Bread pudding protocols

We begin this chapter by giving a formal definition to what is a PoW protocol. We start by denoting t_s the start time of a PoW protocol and t_c the completion time. Then the aim of a PoW protocol is to enable to a person, say A to demonstrate that she/he has performed a certain amount of computation in his execution interval $[t_s, t_c]$. We assume too that the prover A is allowed to perform computations previously, that is she/he can perform computations in the time interval $[-\infty, t_c]$. We give now two definitions that characterize the lower and upper bounds of a PoW protocol.

Definition 4.1. *We say that a Proof of Work PoW is (w, p)-hard if we have the following: Suppose a prover P with a memory resource bounded by m performs an average, over all coin flips by P and V, of at most w steps of computation in the interval $[t_s, t_c]$. Then the verifier V accepts with probability at most $p + o(\frac{m}{poly(l)})$, where l is a security parameter.*

Note: $poly(l)$ denotes set of polynomials in the variable l.

This first definition gives us an idea of the hardness of the PoW we are dealing with. It tells us actually a bound for a number given of computation steps how much success probability we have. We can follow in this direction with the following definition:

Definition 4.2. *We say a PoW to be (w, p, m)-feasible if there exists a prover P with* *memory bounded by m such that an average of w steps of computation in the interval $[t_s, t_c]$. Then the verifier V accepts with probability at least p.*

Definition 4.3. *We say a PoW is complete, if, for some w, the PoW is $(w, 1, poly(l))$-feasible, where l is a security parameter.*

Note that the last definition means that the Proof of Work can be solved (with a probability of 1) in a l-dependent polynomial amount of memory and for a number w of given computations. It certainy gives a sense that the PoW has completeness.
We now give two definitions that introduce us to the idea of a *Bread pudding* PoW protocol:

Definition 4.4. *Suppose that a PoW which we donete PoW_1 is a (w,p)-hard Proof of Work. Let also P_1 denote the prover and V_1 the verifier of this PoW. Suppose that P_1 is also a verifier in another PoW that we call PoW_2 that is $V_2 = P_1$. We say that PoW_2 is a Bread pudding protocol for PoW_1 if the following holds:*

If P_1 or equivally V_2 verifies the PoW_2, then P_1 can perform $w - \epsilon$ computational steps over the calculation of PoW_1 for $\epsilon \geq 0$ and get a verification probability from V_1 of at least p.

Intuitively what this definition tells us is that the computation that we may have done for PoW_2 is recycled in the PoW_1. Observe that trivially PoW_1 is a Bread pudding protocol for PoW_1 itself. Still this observation is useful to gain insight into what these definitions are telling us.

Although this definitions may seem very abstract in the last decade there have been some implementation of some Bread pudding PoW protocol, the most notable were in the field of protein analysis or random mathematical problems [10], [14].

4.2 An ECDLP Bread pudding PoW model

In this section we will outline a new ECDLP Bread pudding PoW model. This model is entirely based in the article [18] from year 2020.

This model has a rather simple blockchain architecture where there are just two types of blocks. The *Epoch block* and *Standard block*. The standard block contains the usual information in a blockchain such as a header, a list of transactions and the instance of PoW whereas the Epoch block is a standard block plus some other information which allows us to generate a PoW that solves generic instances of

the ECDLP as explained in 3.4.1.
We begin outlining the algorithms that generate the settings needed to have a consistent ECDLP.

At the very first moment of setting an ECDLP we need a finite field to work over, we are gonna work over a field in the form of $\mathbb{F}_p$. We calculate p as follows:

Algorithm 1: prime_Gen

```
Require: $d \geq 1$ , $h$
  while p doesn't satisfy security conditions do
      $h \leftarrow H(h)$
      $p \leftarrow \text{NextPrime}(h\, mod(2, 2d))$
  end while
```

Note: d represents a difficulty parameter, observe that the bigger the d the bigger p and thus the harder the ECDLP becomes. h here represents a binary input usually coming from a hash.

With the following security conditions:

Prime security conditions

1. p is not a Crandall prime that means is not of the form $2^k - c$ for a relatively small and positive c.
2. p is not a Generalized Mersenne prime nor a more generalized Mersenne prime.
3. p is not Motgomery-friendly. That is it is not obtained in the form $2^\alpha(2^\beta - \lambda) - 1$ for smalls integers α, β, γ

Next we would determine the elliptic curve E: $y^2 = x^3 + Ax + B$ over $\mathbb{F}_p$:

With the following security conditions:

Algorithm 2: E_Gen

```
Require: $p, \ h$
  $i \leftarrow 0$
  while E doesn't satisfy Security conditions do
      $i \leftarrow i + 1$
      $A \leftarrow H(h + 1)$
      $B \leftarrow H(A)$
  end while
```

Elliptic curve security conditions

1. $\#E = q$ where q is prime and $p \neq q$.
2. The embedding degree of the ellitpic curve is greater than 20, that is $\#E \nmid p^B - 1$ for each $1 \leq B \leq 20$
3. Let D be the *field discriminant* we require $D \geq 2^{40}$

It is important to note that we impose the group of the elliptic curve to be prime, which consequently means that its a cyclic group. Finally the node would have to compute the Base Point P over $E(\mathbb{F}_p)$. Thus the ECDLP to solve is **well defined**.

Algorithm 3: P_Gen

```
Require: $h \quad E$
  $i \leftarrow 0$
  $N \leftarrow 0$
  while $N = 0$ do
      $i \leftarrow i + 1$
      $N \leftarrow$ Number of points in E with x-cord equal to h + i
      $B \leftarrow H(A)$
  end while
  $y \leftarrow$ y such that $(h + 1, y) \in E$ and $0 \leq y \leq p/2$
  $P \leftarrow (h + i, x)$
```

Blockchain structure

As we briefly commented this ECDLP PoW model requires two types of block in the blockchain, the standard block which we will denote $[SB]$ and the epoch block denoted $[EB]$. Next we outline the structure of this two types of blocks

1. **Standard Block**: They don't have any other information apart from the usual: Header, transaction list and PoW instance.
2. **Epoch Block**: Added to the standard block structure they contain the prime p, the elliptic curve E over this prime field and two points $N_1, N_2 \in E(\mathbb{F}_p)$.

The idea behind this model with two kinds of blocks is to be able to change from time to from one elliptic curve to another. The original authors sets one [EB] every 2015 [SB] but this choice seems to be arbitrary.

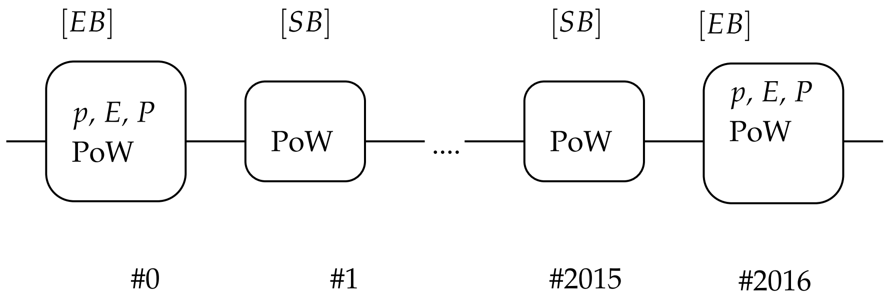


Figure 4.1: Blockchain structure of the proposed model

In order to generate the parameters for a given Epoch block B the calls would be the following:

$$p = \text{p\_Gen}(d, h_{prev})$$
$$E \equiv (A_E, B_E) = \text{E\_Gen}(p, h_{prev})$$
$$P = \text{P\_Gen}(p \,||\, A_E \,||\, B_E, E)$$

Where h_{prev} is the hash of the previous Block header and d is a difficulty parameter settled based on the wanted block confirmation time in the same spirit as bitcoin's mining dificulty parameter. With this all the special *Epoch Block* parameters are defined.

Now that we have exposed the blockchain model, the elliptic curve and base point generating functions we just have left to define the actual PoW to be solved:

Definition 4.5. *Given a Block B_i and a current Epoch Block parameters E, p, P and given σ_k a deterministic digital signature with a given key k, we define the Proof of Work (PoW) to the process of finding the two integers N_1, N_2 such that:*

$$\begin{aligned} P\_Gen(H(\sigma_k(h_{prev})), E) &= N_1 P \\ P\_Gen(H(M), E) &= N_2 P \end{aligned} \tag{4.1}$$

We should observe that by finding N_1 we already force the miner to solve at least a generic instance of ECDLP. Since E and P are determined by the epoch and h_{prev} as is based on the previous block the miner has no control over them. As σ_k is deterministic each miner has to solve at least one instance.

One could argue that there might be collisions of the type: $H(\sigma_k(h_1)) = H(\sigma_k(h_2))$ this can be easily avoided choosing a long enough hash function and also avoided given the change of elliptic curve every 2016 blocks which ensures that the finite possibilities of ECDLP for a given curve are not exhausted. Still we add extra security by solving another ECDLP with the calculation of N_2. As the original author of this model comments this adds extra security but still with just one ECDLP it would be enough.

The Epoch Block and Standard block will then have this structure:

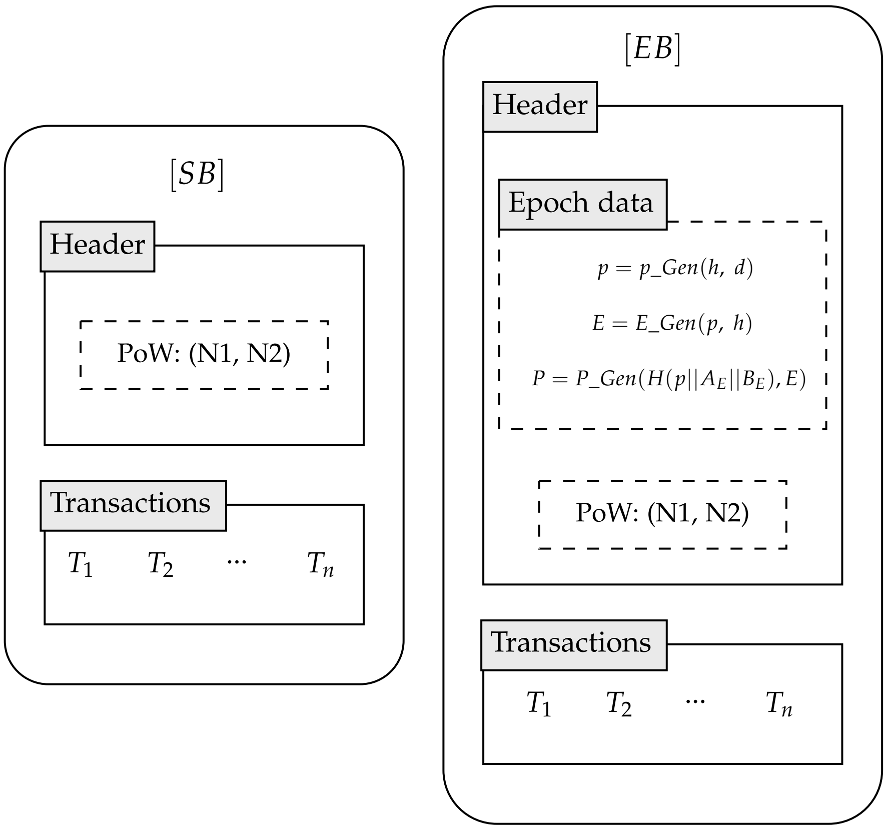


Figure 4.2: $[EB]$ and $[SB]$ structure of the proposed model

4.3 An implementation

In this last chapter we explain an implementation of the above exposed Blockchain mining model. The goal was to check if the above model is realistic in a real programming language, not using specialized mathematical software and thus in the future seeing it in a real life scenario.

The programming language of choice was C mainly due to optimization reasons as we wanted the algorithm to be as fast and efficient as possible, we predicted we could have some efficiency issues (as we had) so the C as our programming language choice is justified. Also we wanted an easily cross-compiled language so it could be easy to make portable an implementation of a blockchain with this PoW.

The following libraries are used in the implementation:

1. OpenSSL: An open source cryptography library. [1]

2. Flint: A C library for doing fast number theory. [2]
3. GMP: A free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. [3]
4. SEA: An implementation of the Schoofs-Elkies-Atkin.[1] [21]

Our initial approach was to just use the OpenSSL library because we needed an interface for working with the arithmetic elliptic curves and applying hash functions mainly.

Later, as we will comment, we also found us in the need of working with polynomials over finite fields and thus the Flint library was used which does use underground the GMP library. Later we added also to our project an implementation of the SEA algorithm.

4.3.1 Details of implementation and problems

Firstly the code makes extensive use of hash functions we decided the algorithm to be using the *SHA256* hash function. Next we outline the core functions of the algorithm and the problems we had with their implementation, please refer to the chapter 4.2 in order to see the pseudocode and to `https://github.com/mariomoliner/TFG` for the actual source code of the implementation.

prime_gen (algorithm 1)

```
/*
*  INPUT: d,  const unsigned char *
*  OUTPUT: BIGNUM *
*  given the difficulty parameter and a hash generates a prime number
*/
BIGNUM * prime_Gen(int d, const unsigned char * hash);
```

For the prime generation, we didn't implement any of the *Prime security checks* due to firstly time constraint and second because we considered this checks to not be of critical importance for the algorithm. *Prime security checks* restrain us some types of prime numbers from our choice. These kind of primes make multiplication over their respective prime field faster but they don't provide any vulnerability whatsoever when working in Elliptic curves over their prime field [22] pag. 114. Thus from security standpoint it was not critical to implement those. The only possible

[1]The Schoofs-Elkies-Atkin (SEA) is an efficient algorithm to count points on elliptic curves over finite fields.

outcome when this is not implemented is that there might be some ECDLP that are faster to compute, but for all participants in the network. Thus reducing the block generating time, at this point is where the Epoch block frequency might be useful as way to provide faster or slower frequency depending on how fast the ECDLP are getting solved.

We want to highlight that we also had to implement, as OpenSSL doesn't include it, the function for computing the next prime following a given natural number.

```
/*
*  INPUT: BIGNUM *
*  OUTPUT: BIGNUM *
*  Given a natural number computes the bigger closest prime number
*  and returns it
*/
BIGNUM * Next_prime(BIGNUM * num);
```

E_gen (algorithm 2)

```
/*
*  INPUT: BIGNUM *,  const unsigned char *
*  OUTPUT: EC_GROUP *
*  given a prime p and a hash calculates an elliptic curve and
*  returns it
*/
EC_GROUP * E_Gen(BIGNUM * p, const unsigned char * hash);
```

For the Elliptic curve generation we had to implement *three security checks*, we didn't implement checking if the *CM discriminant* was big enough due to complexity, as getting the square free part of a random integer was not available to us in the flint library and in fact there is not known any algorithm to compute the square free part of an integer in polynomial time [17]. As we also saw not checking this property could only result in some cases getting the resolution of ECDLP with pollard rho faster than the general case. Eventually a new Epoch block will be generated and so we consider not to be a very important case.
Calculating the embedding degree of the curve was done without major complications. But computing the cardinal of the elliptic curve group brought us many problems. We first approached this problem in a naive way, iterating trough all the possible $x \in \mathbb{F}_p$ for the x-coordinate and adding the existing points. This

approach was really impractical. As the difficulty parameter d was increased the prime field got bigger and so elliptic curve, this approach made the computation of the cardinal of the elliptic curve group exponential. For $d > 10$ the function took too long.
We then implemented a better algorithm using the *Hasse bound*, a profound result of elliptic curves which gives a bound of the cardinal. This theorem tells us that $\#E(\mathbb{F}_q)$ lies in the interval $(q+1-2\sqrt{q}, q+1+2\sqrt{q})$ by making use of this theorem and Lagrange theorem for groups one can iterate over the range of possible cardinalities and if necessary by several points. With this approach one gets an improvement in the efficiency although the algorithm still is exponential. We could up the difficulty d, but for $d > 14$ we faced the same problem as before. It was insufficient to produce in a reasonable timeframe elliptic curves for which the ECDLP was difficult enough.
The faster algorithms to compute de cardinal of the group of elliptic curves are the Schoofs algorithm and the improved version Schoof–Elkies–Atkin algorithm (SEA) algorithm, both with a polynomial complexity. We implemented the Schoofs algorithm which can be found in the file `Utility_Flint.c` but at the end we incorporated the SEA algorithm by Benjamin Wesolowski which also uses the Flint library [21].

P_gen (algorithm 3)

```
/*
*  INPUT: hash,  size, EC_GORUP * E
*  OUTPUT: EC_POINT *
*  given the hash, its size and a pointer to a elliptic curve
*  calculates the base point and returns it
*/
EC_POINT * P_Gen(const unsigned char * hash, int size , EC_GROUP * E);
```

The implementation of P_gen presented no problem. The only challenge is to generate random numbers and check if an x-coordinate exists as a point in the elliptic curve.

API interface

This implementations incorporates a series of abstractions and interfaces so that the algorithm can be easily incorporated into any existing blockchain. We give a brief overview of this API:

```
EPOCH_PoW_INSTANCE EpochPoWInstance_new(const char * hash, int d);
```

This method generates for a given hash and difficulty parameter d the Epoch block parameters, that is (p, E, P)

```
ECDLP_PoW_PROBLEM ECDLPPoWProblem_new(EPOCH_PoW_INSTANCE * instance,
const char * hash_prev, const char * M);
```

Generates the actual PoW to solve given the $[EB]$ the previous hash and the transactions bit data M.

```
ECDLP_PoW_SOLUTION ECDLPPoWSolution_new(ECDLP_PoW_PROBLEM * problem);
```

Does the PoW. (Computes Pollard-rho internally over the ECDLP's of the PoW).

```
bool ECDLPPoWCheckSolution(ECDLP_PoW_SOLUTION * solution);
```

Checks if a PoW is valid, that is if the ECDLP solutions are correct.

The source code of this methods and the type definitions can be found in the file `ECDLP_PoW.c`

In the actual library files the main entry of the algorithm (file `Main.c`) consists just for a given difficulty parameter a complete execution of the algorithm: $[EB]$ generation, PoW instantiation and PoW solution. For a detailed manual on how to install the algorithm please refere to the appendix.

4.3.2 Efficiency

In this section we estimate the efficiency of our implmentation, we shall study the efficiency in the two usual metrics separately: Time complexity and space complexity. We study them using the conventional big O notation, for a good reference in this notation we recommend the following reference [8].

Time Complexity

We give a concise analysis of the time complexity our implementation. For clarity we divide the algorithm in two parts:

1. $[EB]$ generation and PoW parameters generation
2. Actual PoW time

For the first part we have to analyze in the $[EB]$ generation the prime generation first. We have that given our implementation of *NextPrime(n)* we iterate over the odd integers with some properties such as not being congruent 4 modulo 6, etc. By the prime number theorem one can argue that for a sufficiently big integer the next prime is not more separated than a function in $ln(n)$, we denote it f. As we then check the primality of each prime candidate with the *Miller-Rabin* test we obtain in a worst case scenario of time execution in the notation of the O that we have $O(k \cdot f \cdot ln^3(m))$ because *Miller-Rabin* has $O(k \cdot ln^3(m))$ complexity for k checks. As $m = n + ln(n) + c$ for a small constant c and n large. We have that the prime generation has polynomial complexity.

For the Elliptic curve we do per each trial of Elliptic curve a computation of its group cardinal with the SEA algorithm which has complexity $O(log^5(q)$ where q is the prime of the prime field. As $E(\mathbb{F}_q) \approx q$ for large q by the *Hasse bound* if one wants and by the fact that the probability that the elliptic curve group cardinal being prime approaches $\frac{1}{logp}$ as p increases [6]. We have to iterate a function of $logp$ times in order to reach an elliptic curve with prime order and different from the prime p of the prime field, we note such function as g. For the embedding degree we have a result from R. Balasubramanian and Neal Koblitz stating that the probability of finding elliptic curves of low embedding degree is vanishingly low [6]. Summarizing we have that for the Elliptic curve computation we have a time complexity $O(g \cdot log^5(n))$ which is polynomial.

Finally for the base point generation as we just try to get a random point in the curve first getting a random possible x-coordinate and then checking if it belongs to the curve. With the *Hasse bound* taking the lower bound of the cardinal $(q+1) - 2\sqrt{q}$ we have that the probability of a random $x \in \mathbb{F}_q$ as being the x-coordinate of a point in the curve is approximately $\frac{q+1-2\sqrt{q}}{2q} = \frac{1}{2} - \frac{1-2\sqrt{q}}{2q}$ which is approximately for q large $\frac{1}{2} - \frac{1}{\sqrt{q}} \approx \frac{1}{2}$. Thus we would have $O(2)$ time complexity.

As the three parameters have polynomial time complexity the generation of the epoch block $[EB]$ is polynomial. Finally for the generation of the actual PoW recall from formula 4.1 that we require two calls to the function P_gen over some hashes, as argumented before we have polynomial complexity.

For the second part of the algorithm, the actual PoW we implemented Pollard rho and so we have a complexity of $O(2\sqrt{n})$ because we have to solve two generic instances of the ECDLP where n is the cardinal of the elliptic curve.

All in all we proved that **the actual PoW is exponential** and **the Epoch block settings and PoW initialization runs in polynomial time**. This is important because in a real live enviroment it encourages nodes to just worry about the actual

PoW and not the Epoch Block, avoiding lazy miners in those blocks and making the algorithm feasible.

Space complexity

The algorithm works with the *BIGNUM* type of the OpenSSL and the *fmpz* type from the GMP library for long integer arithmetic. We consider these types as the unit storage units. As before we study the storage complexity of the algorithm in two parts:

1. $[EB]$ generation and PoW parameters generation

2. Actual PoW time

for the $[EB]$ generation in the prime generation we have simply $O(1)$ as we iterate the function just keeping in memory two BIGNUMs. For the elliptic curve generation, again as before keeping a constant number of variables, such as the A B. We have to run the SEA algorithm for each iteration which in our implementation has a space complexity $O(ln^2(q))$ where q is the prime of the prime field. [7]

for the Base point and PoW instantiation again we have just $O(1)$ complexity. All in all we have $O(ln^2(q))$ complexity, thus polynomial.

For the second part of the algorithm, the actual PoW solution as we just run two instances of Pollard rho we have $O(1)$ space complexity as seen in the security chapter 3.4.3.

Summarizing we have that **the $[EB]$ generation and PoW parameters generation has polynomial space complexity** whereas **the actual PoW has polynomial space complexity** also.

Chapter 5

Results

In this section we present the results obtained from the implementation of the algorithm. For a manual on how to install the software please refer to the appendix where the reader can find step by step the installation process.

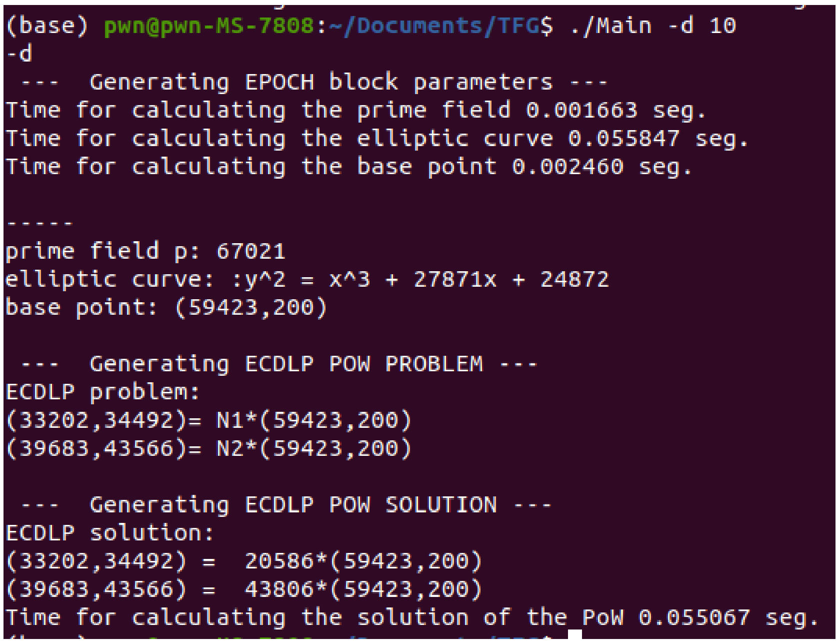
5.1 Results

All of the executions that are showed in this chapter have been produced over the following hardware and software:

- RAM: DIMM DDR3 Synchronous 1333 MHz 8Gb Kingston
- Processor: Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz
- OS: Ubuntu 20.04 LTS

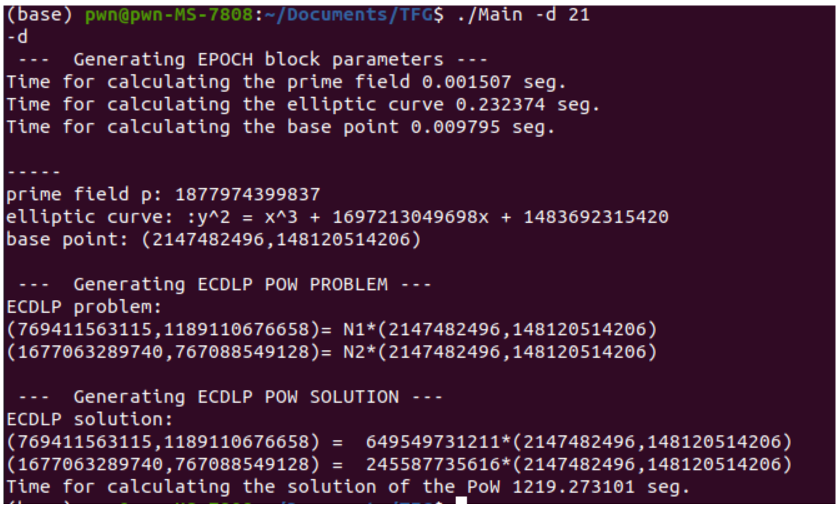
The results of the algorithm were sucessfull starting from $d \geq 5$. For $d < 5$ we found out that the elliptic curve generation 2 failed due not finding one satisfying the security conditions 4.2. This is in part normal because the space of possible elliptic curves is much smaller and actually the ECDLP makes no sense in terms of computational complexity. Thus we just care about executions from $d = 5$.

We give the output of two results of execution one with a difficulty parameter $d = 10$ and another with $d = 21$. As the entry input to the algorithm requires also the *hash_prev* and the transactions chunk data M they are both generated randomly at each execution.



```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main -d 10
-d
 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001663 seg.
Time for calculating the elliptic curve 0.055847 seg.
Time for calculating the base point 0.002460 seg.

-----
prime field p: 67021
elliptic curve: :y^2 = x^3 + 27871x + 24872
base point: (59423,200)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(33202,34492)= N1*(59423,200)
(39683,43566)= N2*(59423,200)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(33202,34492) =  20586*(59423,200)
(39683,43566) =  43806*(59423,200)
Time for calculating the solution of the PoW 0.055067 seg.
```

Figure 5.1: Execution with d = 10



```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main -d 21
-d
 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001507 seg.
Time for calculating the elliptic curve 0.232374 seg.
Time for calculating the base point 0.009795 seg.

-----
prime field p: 1877974399837
elliptic curve: :y^2 = x^3 + 1697213049698x + 1483692315420
base point: (2147482496,148120514206)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(769411563115,1189110676658)= N1*(2147482496,148120514206)
(1677063289740,767088549128)= N2*(2147482496,148120514206)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(769411563115,1189110676658) =  649549731211*(2147482496,148120514206)
(1677063289740,767088549128) =  245587735616*(2147482496,148120514206)
Time for calculating the solution of the PoW 1219.273101 seg.
```

Figure 5.2: Execution with d = 21

Logically the prime p gets bigger as the difficulty parameter increases. Please check out the table 1 of the appendix to see how the execution times evolve and a more detailed execution results for each d.

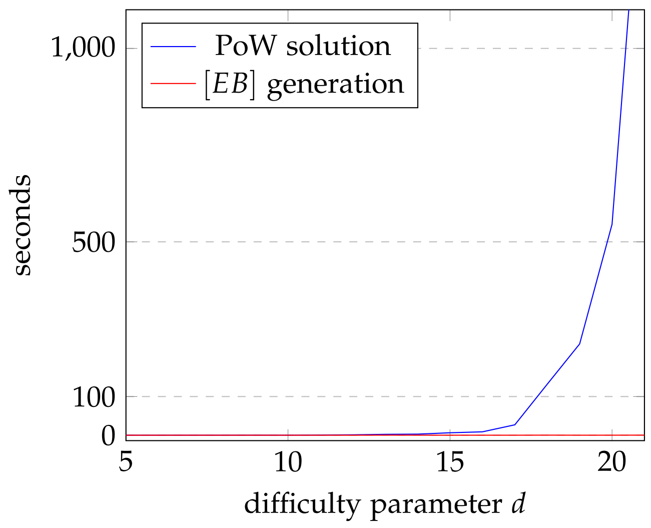


Figure 5.3: Execution times

In order to check the consistency of the results we also did some extensive testing for all difficulties ranging from $5 \leq d \leq 20$, and then checking if the ECDLP was correct. These tests configuration can be found in `Main_test.c`. please refer to the appendix in order to see how to execute it.

5.2 Benchmarks

In this section we shall give a comparison with the PoW algorithm of bitcoin. For this we developed a very simple PoW that mimicks the one used by bitcoin, it is available in the source code of the implementation in the file `Bitcoin_POW/Pow.c`. Summarized this algorithm is just applying a double *SHA256* until some criteria is met. Please check chapter 2.4 for a more in-depth explanation of what this PoW does.

We executed this PoW and found out the following metrics using *PowerTOP* [1]:

Hash/s	PoWer consumption
1.16 GHash/s	ca. 1 Watt

Table 5.1: mean Bitcoin PoW metrics (n=5)

[1]PowerTOP is a software utility that serves to measure, and monitor electrical PoWer consumption of computer programs.

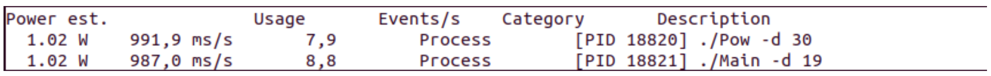
We repeat the same metrics but for our Elliptic curve based algorithm, note that in this implementation the mining basic unit is not Hash/s as we are doing Pollard-rho internally in the mining process not hash functions. We propose to call the basic mining unit in our algorithm as *Elliptic curve walks* or abbreviated ECW, this unit should refer to the number semi-random walks we do in the curve as the defined in the Pollard rho algorithm in chapter 3.4.3. The metrics in our implementation have been then:

ECW/s	Watts
10.4 kECW/s	ca. 1 Watt

Table 5.2: mean ECDLP based PoW metrics (n=5)

We observe how the number of Hash/s and ECW/s differ significantly but this doesn't tells us anything in terms of energy consumption. Still, this metric may serve as a benchmark in the future if a more spezialized hardware/software would be developed for Elliptic curve arithmetic or if a better solution to ECDLP were to be found.

The Watt consumption of the algorithm does tells us than on quite generic hardware settings **the energy consumption of both algorithms is similar**. This is important because proves that our algorithm may be a viable alternative to the one that Bitcoin uses.



```
Power est.              Usage       Events/s    Category        Description
  1.02 W    991,9 ms/s       7,9        Process        [PID 18820] ./Pow -d 30
  1.02 W    987,0 ms/s       8,8        Process        [PID 18821] ./Main -d 19
```

Figure 5.4: Watts consumption of both PoWs

5.3 Discussion

As we expected from the complexity analysis in chapter 4.3.2 the $[EB]$ time generation grows slow while the actual PoW grows exponentially (figure 5.3). This as already commented is important because nodes in a real live enviroment only care about the mining process, because they usually are incentivized with coin returns. As the $[EB]$ block generation time is negligible in comparison with the actual PoW the lazy miners are avoided. This confirms what was stated in the original model of this algorithm but now in a more low-level and applicable language such as C.

Taking for instance the average bitcoin block generation time which is around

9 minutes [13]. We find from our results that a difficulty parameter d around 20 to be the best in order obtain this block mining time. This difficulty gives primes with around 40 bit size. We want to remark that this is node-implementation dependent and if a more eficient Pollard-rho implementation were to be found, this difficulty parameter would have to go higher. This presents no problem as the $[EB]$ generation time worked correctly for any d, even ones higher to 40. see Table 2.

As we compared in the chapter of benchmarks 5.2 this novel PoW implementation seems to be somewhat similar to the PoW of bitcoin. At least from a energy consumption standpoint, our implementation has the major difference that it carries with it some intrinsic value. Supposing a mathematical research around the ECDLP then **this PoW solution in a public blockchain could serve as a public database of known ECDLP's** as commented in chapter 2.1. This would serve the researchers to have more information from the start and thus reducing some possible work. Indeed this PoW algorithm is a Bread pudding PoW as defined in 4.4.

Chapter 6

Conclusions and future work

We started this work with the motivation of studying cryptocurrencies in order to gain insight at the points where they might be improved in terms of ecosostenibility. As we made progress in our study, we identified the Proof of Work protocols as the main culprit of their big energy consumption. We then checked the alternatives and found out that possible or at least more sustainable alternatives exist, such as Bread pudding PoW protocols. From this point we had in our mind the possible implementation of a better alternative in line with a Bread pudding protocol.

Later, having found the proposal of a Proof of Work based in the ECDLP [18], We started studying the Elliptic curves in chapter 3. There we uncovered all the complexity and thus security that these mathematical objects have to offer in terms of cryptographical applications. As we found out the DLP and also the ECDLP are used in a lot of signing algorithms and thus are a big area of mathematical investigation. This made the aforementioned Proof of Work (4.2) even more interesting being based on this critical problem. We found out that this protocol was still not implemented in a low level applicable language so we decided to try to implement it. As we progressed we found some problems expecially with the elliptic curve generation (4.3.1). This required us to implement the best up to date algorithm for tasks such as counting the cardinal of the elliptic curve group. All in all we ended having a fully functional library of this PoW protocol, providing an interface so that it can be easily incorporated in future blockchain solutions.

We have high hopes with the implementation given here, we consider the results good enough to answer the main question we had in the beginning of this work. We consider as proven the algorithm to be a feasible and efficient enough and to be a real alternative to the traditional PoW mining models.

Still, there are some questions that remain unanswered such as what would be the best Epoch block frequency strategy? Or if the prime security conditions, which weren't implemented in this work, may become a challenge in terms of efficiency of the algorithm. Also the third security condition of the elliptic curve 4.2 stating the the CM discriminant to be high couldn't be implemented. As argued in 5.3 in the discussion, this shouldn't suppose a big issue.

Another big unknown that we in part answered is in the metrics department. As we found in 5.2 the energy consumption was found to be similar to the one of bitcoin, this was done in a generic hardware but there is still the interrogation if maybe in a large scale deployment of this algorithm keeps being similar. Nevertheless we consider the preliminary results promising.

Concluding thoughts, we view this work as a necessary starting point for a real life implementation of this PoW model, which we consider comparable to the one that uses bitcoin. And given its research applicability, a better and more sustainable choice.

Bibliography

[1] Cryptography and ssl/tls toolkit. URL: `https://www.openssl.org/`.

[2] Flint: Fast library for number theory. URL: `https://www.flintlib.org/`.

[3] The gnu multiple precision arithmetic library. URL: `https://gmplib.org/`.

[4] Andreas M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'reilly, 2017.

[5] Shi Bai1 and Richard P. Brent. On the efficiency of pollards rho method for discrete logarithms.

[6] R. Balasubramanian and Neal Koblitz. The improbability that an elliptic curve has subexponential discrete log problem under the menezes-âokamoto-âvanstone algorithm. *Journal of Cryptology*, 2015.

[7] F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Advances in Elliptic Curve Cryptography*. Cambridge university press, 2005.

[8] Thomas H. Cormen. *introduction ot algorithms*. MIT Press, 2012.

[9] STEFAN FRIEDL. An elementary proof of the group law for elliptic curves. URL: `https://www.uni-regensburg.de/Fakultaeten/nat_Fak_I/friedl/papers/elliptic_2017.pdf`.

[10] Marcella Hastings, Nadia Neninger, and Eric Wustrow. Proofs of work for solving discrete logarithms. URL: `https://eprint.iacr.org/2018/939.pdf`.

[11] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Sringer, 1986.

[12] Juels A. Jakobsson M. Proofs of work and bread pudding protocols. *Springer, Boston, MA*, 1999.

[13] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. URL: `https://github.com/Calodeon/sea`.

[14] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. URL: `https://primecoin.io/bin/primecoin-paper.pdf`.

[15] Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Designs, Codes and Cryptography*, 2000.

[16] Florian Luca and Igor E. Shparlinsk. Discriminants of complex multiplication fields of elliptic curves over finite fields. *Canadian mathematical bulletin*, 2015.

[17] Leonard M.Adleman and Kevin S.McCurley. Open problems in number theoretic complexity. 2014.

[18] Alessio Meneghetti, Massimiliano Sala, and Daniele Taufer. A new ecdlp-based pow model. *Mathematics 2020*. URL: `https://arxiv.org/pdf/1911.11287.pdf`.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: `https://bitcoin.org/bitcoin.pdf`.

[20] J. M. Pollard. Monte carlo methods for index computation. 1978.

[21] Benjamin Wesolowski. Schoofs-elkies-atkin implementation using flint. URL: `https://github.com/Calodeon/sea`.

[22] Michael J. Wiener and Robert J. Zuccherato. *Faster Attacks on Elliptic curve Cryptosystems*. Sringer, 2002.

Appendix

Results

We present detailed tables and screenshots from the execution results of the implementation provided.

We begin with a table describing the computation times of the different parts of the algorithm for different values of d:

d	prime_gen	E_gen	P_gen	$[EB]$ generation	PoW solution
5	0.001780	0.023609	0.001365	0.026754	0.007589
6	0.001648	0.018227	0.001380	0.021255	0.005994
7	0.001513	0.032428	0.032428	0.066369	0.021581
8	0.001885	0.014570	0.001423	0.017878	0.026716
9	0.002001	0.048787	0.001459	0.052247	0.143009
10	0.001840	0.016735	0.001455	0.02003	0.021660
11	0.002248	0.023011	0.001577	0.026836	0.349596
12	0.002321	0.024308	0.001670	0.028299	1.025723
13	0.001978	0.126582	0.126582	0.255142	2.229151
14	0.002445	0.143358	0.003201	0.149004	2.828037
15	0.002057	0.019242	0.002681	0.02398	6.348178
16	0.002314	0.057190	0.001900	0.061404	8.778656
17	0.001875	0.030198	0.004450	0.036523	26.921133
18	0.002715	0.220471	0.004818	0.228004	131.960876
19	0.002337	0.042459	0.005091	0.049887	236.030474
20	0.001395	0.068363	0.004278	0.074036	545.357499
21	0.001507	0.232374	0.009795	0.243676	1619.273101

Table 1: mean execution times for different diffciulties in seconds (n=5)

Next we provide a table describing the increasing number of bits of the prime generated number, p for $\mathbb{F}_p$, as d increases:

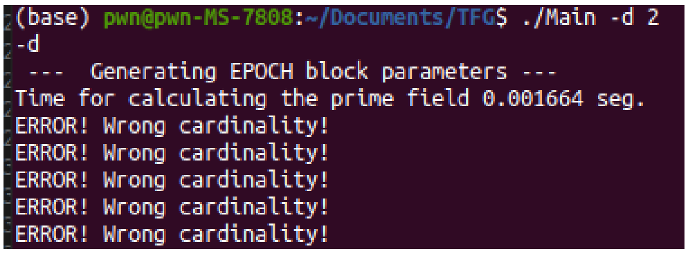
d	prime num_bits
5	10
7	14
9	18
11	21
13	24
15	29
17	34
19	38
21	42
23	46
25	50
27	53
29	57
31	61
33	64
35	70
37	74
39	77

Table 2: mean prime p number of bits for different d (n=5)

Some execution examples

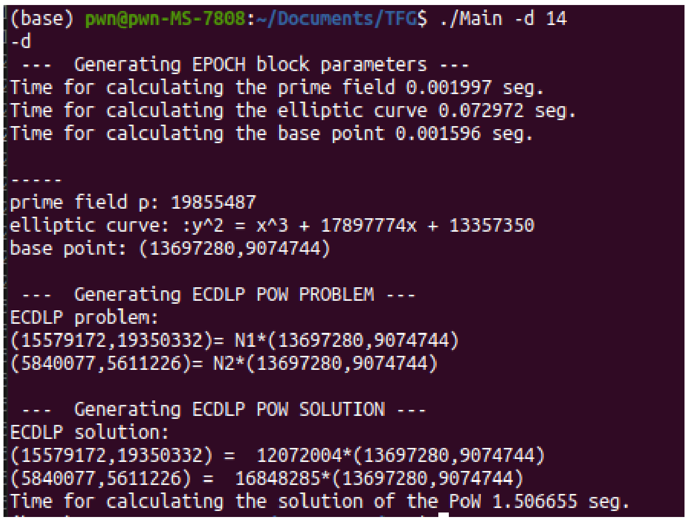
Below the reader can find some execution examples:

1. `Main` compiled from `https://github.com/mariomoliner/TFG/blob/main/Main.c`. Executions of the main entry point of the algorithm, for a given difficulty [-d], executes a normal instance of the PoW, from beginning to end.



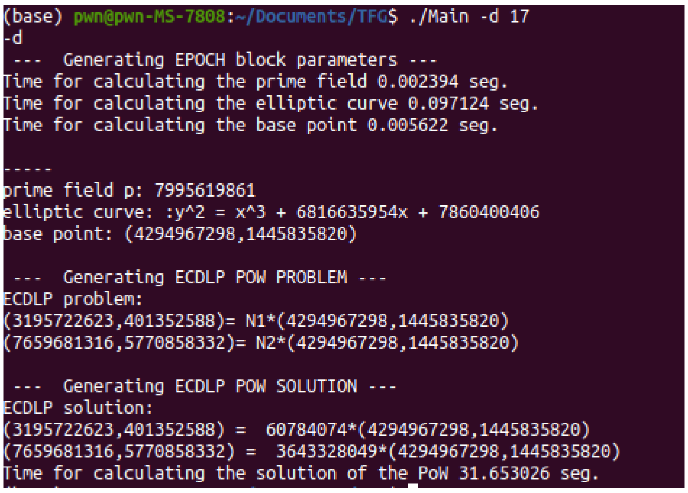
```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main -d 2
-d
 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001664 seg.
ERROR! Wrong cardinality!
ERROR! Wrong cardinality!
ERROR! Wrong cardinality!
ERROR! Wrong cardinality!
ERROR! Wrong cardinality!
```

Figure 1: Execution with d = 2 (failed)



```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main -d 14
-d
 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001997 seg.
Time for calculating the elliptic curve 0.072972 seg.
Time for calculating the base point 0.001596 seg.

-----
prime field p: 19855487
elliptic curve: :y^2 = x^3 + 17897774x + 13357350
base point: (13697280,9074744)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(15579172,19350332)= N1*(13697280,9074744)
(5840077,5611226)= N2*(13697280,9074744)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(15579172,19350332) =  12072004*(13697280,9074744)
(5840077,5611226) =  16848285*(13697280,9074744)
Time for calculating the solution of the PoW 1.506655 seg.
```

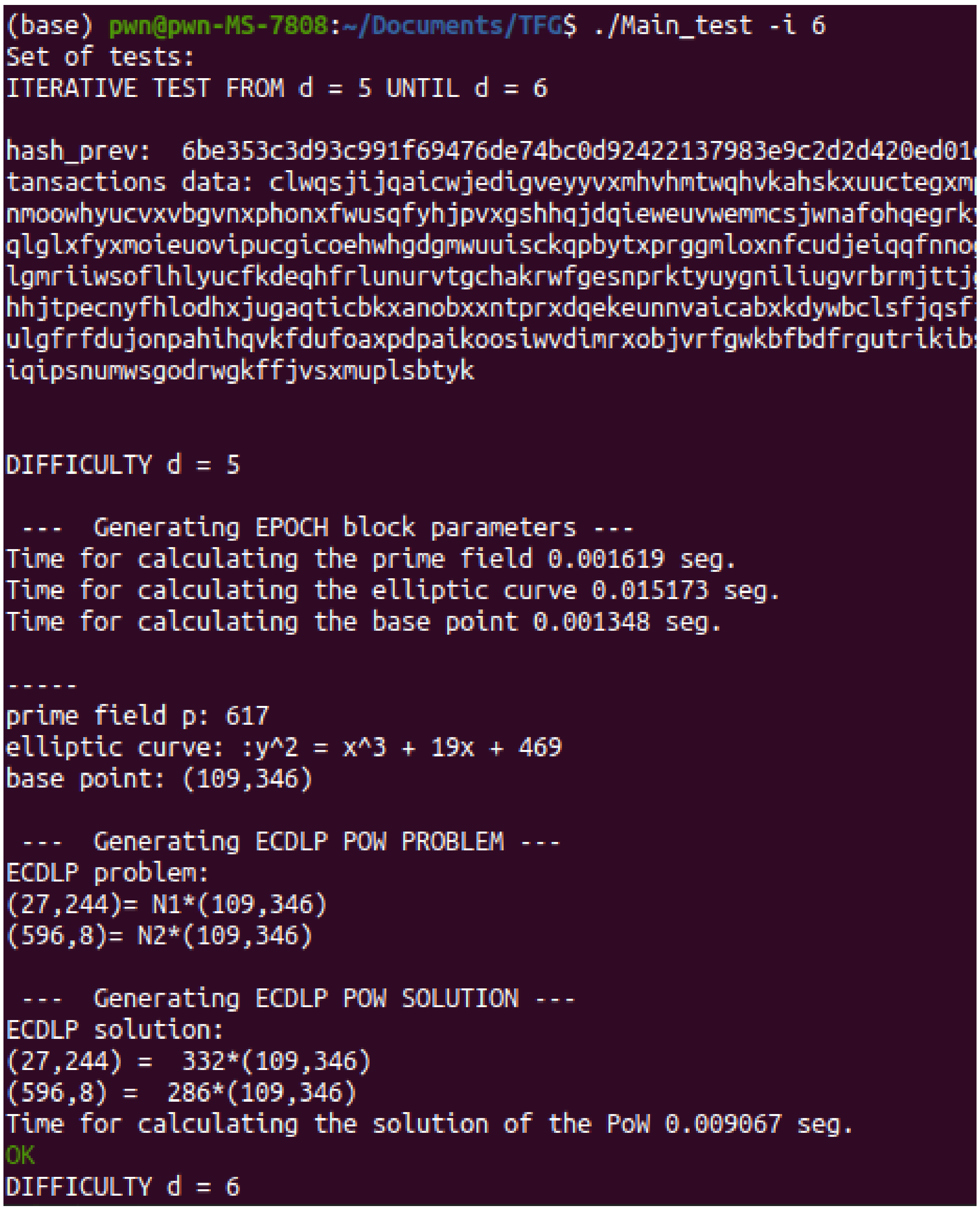
Figure 2: Execution with d = 14



```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main -d 17
-d
 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.002394 seg.
Time for calculating the elliptic curve 0.097124 seg.
Time for calculating the base point 0.005622 seg.

-----
prime field p: 7995619861
elliptic curve: :y^2 = x^3 + 6816635954x + 7860400406
base point: (4294967298,1445835820)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(3195722623,401352588)= N1*(4294967298,1445835820)
(7659681316,5770858332)= N2*(4294967298,1445835820)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(3195722623,401352588) =  60784074*(4294967298,1445835820)
(7659681316,5770858332) =  3643328049*(4294967298,1445835820)
Time for calculating the solution of the PoW 31.653026 seg.
```

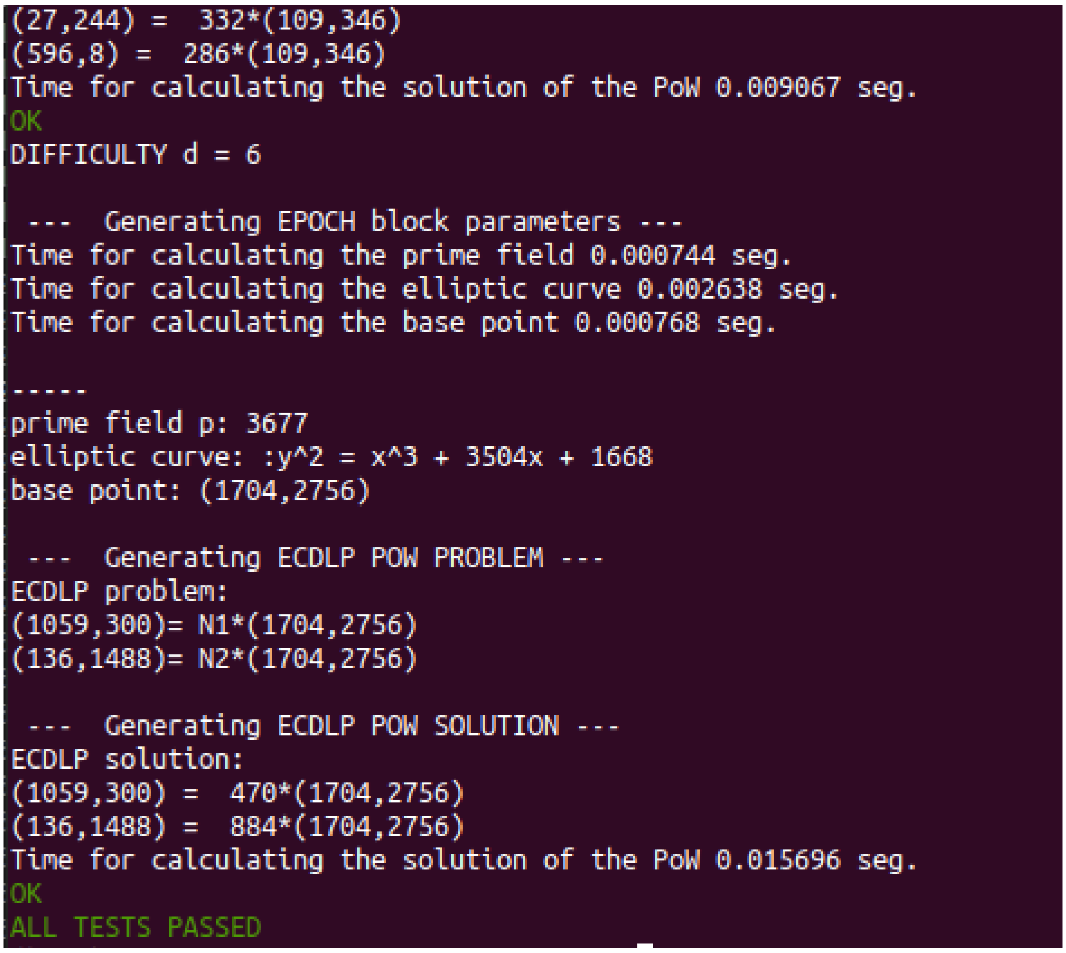
Figure 3: Execution with d = 17

2. Main https://github.com/mariomoliner/TFG/blob/main/Main_test.c. Executions of the tests of the algorithm, check the documentation provided in https://github.com/mariomoliner/TFG for a better understanding of the execution modes



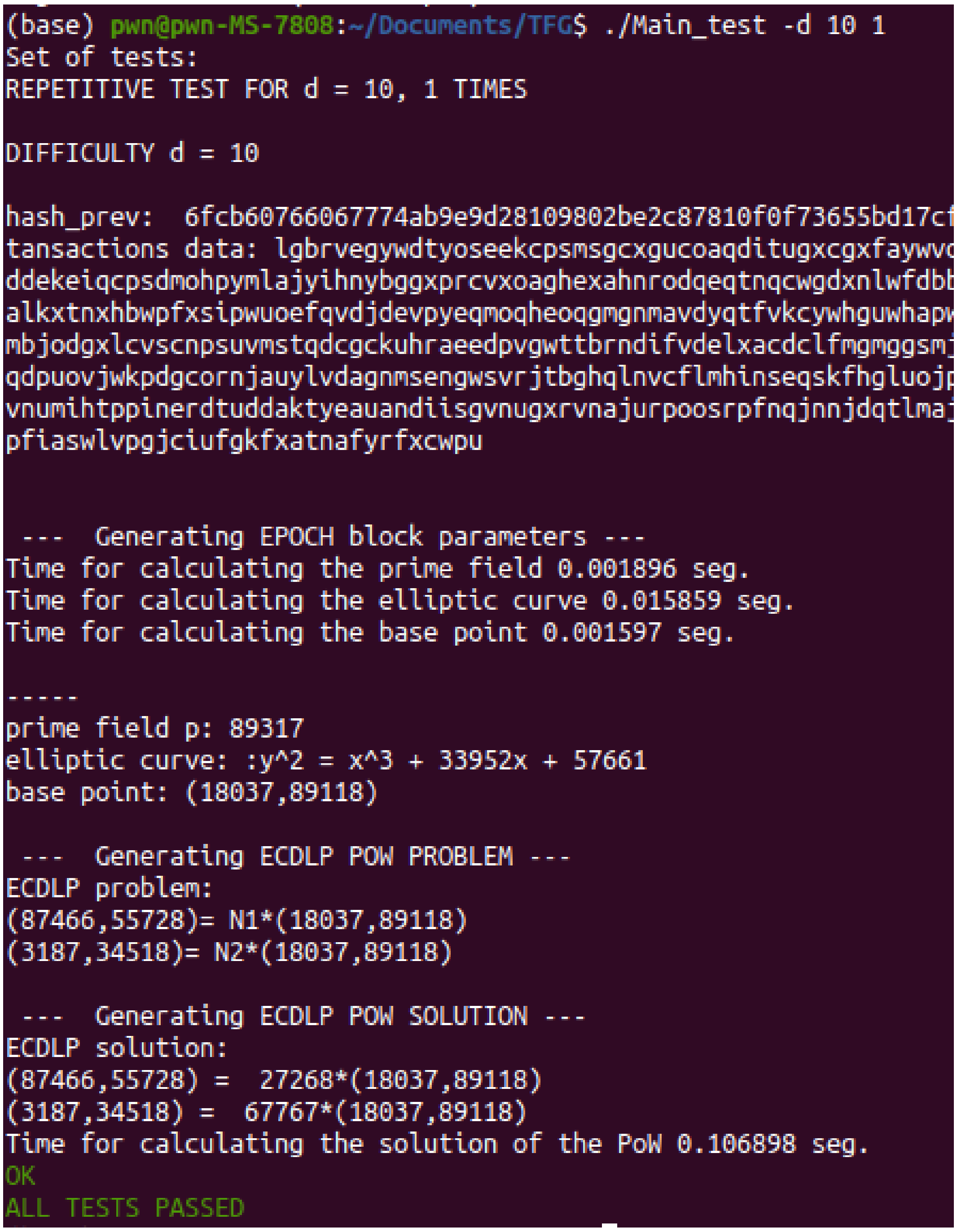
```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main_test -i 6
Set of tests:
ITERATIVE TEST FROM d = 5 UNTIL d = 6

hash_prev:  6be353c3d93c991f69476de74bc0d92422137983e9c2d2d420ed01
tansactions data: clwqsjijqaicwjedigveyyvxmhvhmtwqhvkahskxuuctegxm
nmoowhyucvxvbgvnxphonxfwusqfyhjpvxgshhqjdqieweuvwemmcsjwnafohqegrk
qlglxfyxmoieuovipucgicoehwhgdgmwuuisckqpbytxprggmloxnfcudjeiqqfnno
lgmriiwsoflhlyucfkdeqhfrlunurvtgchakrwfgesnprktyuygniliugvrbrmjttj
hhjtpecnyfhlodhxjugaqticbkxanobxxntprxdqekeunnvaicabxkdywbclsfjqsf
ulgfrfdujonpahihqvkfdufoaxpdpaikoosiwvdimrxobjvrfgwkbfbdfrgutrikib
iqipsnumwsgodrwgkffjvsxmuplsbtyk


DIFFICULTY d = 5

 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001619 seg.
Time for calculating the elliptic curve 0.015173 seg.
Time for calculating the base point 0.001348 seg.

-----
prime field p: 617
elliptic curve: :y^2 = x^3 + 19x + 469
base point: (109,346)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(27,244)= N1*(109,346)
(596,8)= N2*(109,346)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(27,244) =  332*(109,346)
(596,8) =  286*(109,346)
Time for calculating the solution of the PoW 0.009067 seg.
OK
DIFFICULTY d = 6
```

Figure 4: Test iterative -i 6 (pt1)



```
(27,244) =  332*(109,346)
(596,8) =  286*(109,346)
Time for calculating the solution of the PoW 0.009067 seg.
OK
DIFFICULTY d = 6

 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.000744 seg.
Time for calculating the elliptic curve 0.002638 seg.
Time for calculating the base point 0.000768 seg.

-----
prime field p: 3677
elliptic curve: :y^2 = x^3 + 3504x + 1668
base point: (1704,2756)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(1059,300)= N1*(1704,2756)
(136,1488)= N2*(1704,2756)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(1059,300) =  470*(1704,2756)
(136,1488) =  884*(1704,2756)
Time for calculating the solution of the PoW 0.015696 seg.
OK
ALL TESTS PASSED
```

Figure 5: Test iterative -i 6 (pt2)



```
(base) pwn@pwn-MS-7808:~/Documents/TFG$ ./Main_test -d 10 1
Set of tests:
REPETITIVE TEST FOR d = 10, 1 TIMES

DIFFICULTY d = 10

hash_prev:  6fcb60766067774ab9e9d28109802be2c87810f0f73655bd17cf
tansactions data: lgbrvegywdtyoseekcpsmsgcxgucoaqditugxcgxfaywvc
ddekeiqcpsdmohpymlajyihnybggxprcvxoaghexahnrodqeqtnqcwgdxnlwfdbb
alkxtnxhbwpfxsipwuoefqvdjdevpyeqmoqheoqgmgnmavdyqtfvkcywhguwhapw
mbjodgxlcvscnpsuvmstqdcgckuhraeedpvgwttbrndifvdelxacdclfmgmggsmj
qdpuovjwkpdgcornjauylvdagnmsengwsvrjtbghqlnvcflmhinseqskfhgluojp
vnumihtppinerdtuddaktyeauandiisgvnugxrvnajurpoosrpfnqjnnjdqtlmaj
pfiaswlvpgjciufgkfxatnafyrfxcwpu


 ---  Generating EPOCH block parameters ---
Time for calculating the prime field 0.001896 seg.
Time for calculating the elliptic curve 0.015859 seg.
Time for calculating the base point 0.001597 seg.

-----
prime field p: 89317
elliptic curve: :y^2 = x^3 + 33952x + 57661
base point: (18037,89118)

 ---  Generating ECDLP POW PROBLEM ---
ECDLP problem:
(87466,55728)= N1*(18037,89118)
(3187,34518)= N2*(18037,89118)

 ---  Generating ECDLP POW SOLUTION ---
ECDLP solution:
(87466,55728) =  27268*(18037,89118)
(3187,34518) =  67767*(18037,89118)
Time for calculating the solution of the PoW 0.106898 seg.
OK
ALL TESTS PASSED
```

Figure 6: Test repetitive -d 10 1

Installation and execution

We refer the reader to the main repository: `https://github.com/mariomoliner/TFG` where the source code is hosted and detailed instructions on how to install the dependencies and execute the software can be found.