Mathematics of Bitcoin

Miranda Wood

Brunel University

College of Engineering, Design and Physical Sciences
Department of Mathematics

Mathematics B.Sc.

Year of Submission: 2018
Student Number: 1310205
Supervisor: Dr. Martins Bruveris

Abstract

Bitcoin, now in its tenth year, remains an important and controversial topic in commerce. The first of its kind, it is a completely independent peer-to-peer digital currency. There is no central authority; instead trust is built upon mathematical proof. This report outlines how mathematics is applied to Bitcoin's underlying system. We first describe its application of public key cryptography, which allows transactions to be authenticated with digital signatures. These signatures provide irrefutable proof of authenticity due to the intractability of the elliptic curve discrete logarithm problem. We then show how hash functions are used to provide assurance of data integrity, essential to the open source system's security. Alongside this report is a functional Matlab implementation of the elliptic curve cryptographic algorithms as Bitcoin uses them.

Acknowledgments

I must first extend my thanks to my supervisor, Dr. Martins Bruveris. He has guided me through this project, providing wisdom with the utmost patience, for which I am so grateful.

Over the five tumultuous, joyful, incredible years studying here I have been lucky enough to gain another family who have given me more support than I deserve. Thanks are not limited to Joanne Cassidy, Edward Davies, Coleman Deady Ridge, Katie Dixon-Warren, Rachel Frith, Kate Taylor, and lastly (and more often than not, least) my long-suffering coursemate Sam Ward.

Finally, para mi padre, eres mi mayor fortaleza. Gracias por todo.

Contents

Chapter 1

Introduction

Bitcoin is a decentralised, peer-to-peer currency with no administering financial institution, country, or state. It was introduced in a 2008 whitepaper under the pseudonym Satoshi Nakamoto [25]. Despite extensive efforts to discover Nakamoto's identity, the creator remains anonymous to this day. His goal was to combat flaws intrinsic to modern internet transactions, such as the need to share a large amount of personal information and to earn third party permission from a bank to use your own assets.

By design, Bitcoin is electronic cash that mirrors spending money in person [22]. However, without any mediator, how does Bitcoin do this while ensuring payments are secure as well as safe from double spending?

In lieu of any centralised regulator, trust is built upon cryptographic proof. Transactions are verified by users who are part of a large network. For a new payment to be verified, an instruction is sent to the network to unlink a certain number of bitcoins from one *address* (or *public key*), akin to a bank account, and link them to another. Each Bitcoin address has a secret *private key*, the proof of ownership, that enables the owner to send payments.

The payment instruction has a cryptographic signature created using the Bitcoin sender's private key. This is central to the verification process since *only* the holder of that private key could produce this signature. They are safe from interference due to the application of *hash functions* in the signing process. Crucially, any user on the network can verify that this signature is legitimate without finding the private key. This is due to the intractability of what is known as the *Elliptic Curve Discrete Logarithm Problem* [8, 32, p. 254]. This problem will form the central topic of this project.

In the ten years since its introduction, Bitcoin has considerably grown in popularity and currently boasts 21 million wallet users, many of whom have been drawn in by the promise of a revolutionary way to transfer money [27].

Two distinct groups of users initially drove Bitcoin's popularity. The apparent programming expertise of Nakamoto and the the swift release of an open source prototype, just two months after the publication of Nakamoto's whitepaper, attracted developers and researchers [16, p. 3]. Meanwhile, the lack of regulation attracted criminals and those with controversial businesses, some of whom operated on the notorious 'Silk Road' [35]. This contributed to early scepticism in mainstream media, nevertheless the mystery of Bitcoin's creator and the innovative underlying technology has sustained public interest.

1.1 Bitcoin Transactions

Each bitcoin (or BTC) is associated with with an aforementioned address/public key. This address is visible throughout the network but intentionally contains no personal information, therefore ownership must be proven by cryptographic means. Using the private key provides this proof since it is known only to the owner of the corresponding Bitcoin address [8]. An address's private key is kept secret and is computationally infeasible to discover from public knowledge. Additionally, there is such a huge number of possible private keys it is nearly impossible to choose one that is in use. It allows the owner of an address to authorise payments in such a way that the Bitcoin network is able to confirm ownership without ever finding the private key. We will explore public key cryptography in depth in Section 2.1.

To spend a coin, one *digitally signs* a transaction that transfers ownership of that coin to the new address (*see Figure 1.1*). Much like signing a cheque, it provides assurance that the currency is authorised to change hands. This is an action any peer can verify using public information, but only the sender can complete. In fact, since

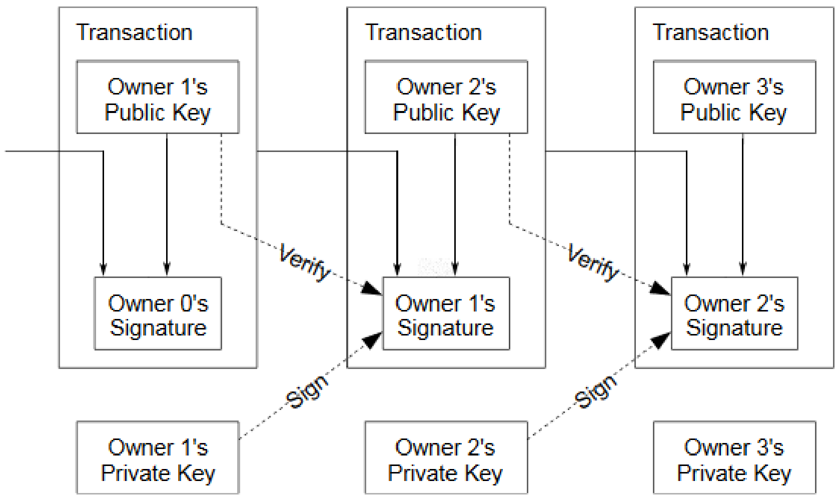


Figure 1.1: Simplified Bitcoin transaction chain [25]

one BTC has no physical form, it is defined as a chain of signatures [25]. Coins are designed in this way so a user can verify its current and all previous owners, analogous to a publicly available ledger. The use of cryptographic signatures and hash functions provides irrefutable proof of this chain of ownership.

In other words, the transaction is a signed instruction to the network. It contains an input amount, from the source(s) of BTC, and an output amount, for the destination(s) of BTC. There can be both multiple sources and destinations in a transaction, though most commonly the source is a single Bitcoin address [4]. The transaction must define a destination for *all* the BTC in the source addresses [1, p. 24]. For example, a user would like to send 0.0139 BTC to their friend, but has just above 0.174 BTC associated with their address. The input is the entire address amount and the outputs are split between two destinations; 0.0139 BTC to their friend and the remaining 'change' is sent back to themselves. This example corresponds to the real Bitcoin transaction[1] below:

Field	Address	Amount
Input 1	`1H6ZZpRmMnrw8ytepV3BYwMjYYnEkWDqVP`	0.17401602 BTC
Output 1	`3NMUhNaFx8azDJtdAZobXVgD7M8etst6hr`	0.0139 BTC
Output 2	`1H6ZZpRmMnrw8ytepV3BYwMjYYnEkWDqVP`	0.15411602 BTC
Fees	-	0.006 BTC

Table 1.1: Transaction `e817c101b4518c0aede8323f84eb3d2b6df433e52ba07fd2da66899ad77607fd`

A new transaction is broadcast to the network where peers check the authenticity of the signature and whether the coin has been spent previously. If it is found to be valid by a user, it is stored in their local memory [17]. It is then grouped with other new transactions and recorded publicly in the blockchain [4, 28].

1.2 The Blockchain

Nakamoto invented what is known as the *blockchain*, simultaneously storing and verifying every single Bitcoin transaction that has happened. It enables data to be recorded securely with no need for a centralised authority, instead relying on the network of users to contribute computing power and rewarding them for doing so. A "truly genuine breakthrough" [16, p. 163], the blockchain has a wide range of applications outside digital currency such as data storage, identity management, and contract administration [33].

The blockchain relies heavily on *hash functions*, which will be explored in detail in Chapter 3. A hash function takes a string of arbitrary length that is *hashed* to output a

[1]Anyone can check this transaction, or any other, via an online block explorer like https://blockchain.info/.

number of fixed length, unique to the input [33]. In short, a hash is a digital fingerprint. The properties that are important for our purposes are:

- It is nearly impossible to find an input that gives a known hash,
- A minor change in the input results in a completely different hash.

When given the original input and its hash, these properties allow anyone to ensure that the input has not been changed.

Newly verified transactions are grouped into a *block* which, when completed, is linked to the previous block, forming the blockchain. Blocks are generated by peers on the network, for which they receive a reward, currently 12.5 BTC per block [4]. This process is known as *mining* and is highly competitive.

Table 1.2: Block Number 510860

Field	Content	Description

Previous Block	`000000000000000000038489d85842113`
	`82cfeb380e2765135eadf0a9a0ab4ef7`
		Block hash of last created block

Transactions	`f2e245eb760a64932793a16eafe2c606`
	`4558a940d234a2670969f0ed5c83299c`
		Hash of all transactions in block

Time	`Feb 25, 2018 2:57:45 PM`	Time of block creation
Version	`02000000`	Blockchain version number
Bits	`175d97dc`	Difficulty of created block
Nonce	`3074094792`	Randomly chosen integer

$\longrightarrow$

Block Hash
`00000000000000000059cc97f89734474ba2116eb61c4caad6d51f80af6ba3e8`

Each block has a *block* or *title* hash that combines various fields as inputs (*see Table 1.2*). To create a block users must discover, or *mine*, the nonce value that is then hashed with the other pre-determined fields. If the resulting title hash starts with a given number of zeros, the block is added to the chain [4]. This new block is sent to all users on the network, who check the hash computation and, if valid, add it to their local version of the chain.

Due to the aforementioned properties of hash functions, there is no efficient way to predict a nonce value that yields a suitable block hash. Hence, users must use computing power to repeatedly try new values to find a hash starting with enough zeros. This ensures no block has been created, and therefore no BTC reward has been gained, without considerable 'work' [28].

Bitcoin mining is based on this *proof of work* system; it is deliberately difficult to create blocks but easy to verify that the work has been done [22]. Before 'accepting' a new block, a user verifies the work by hashing all the fields and checking the result matches the block hash. This is a fast computation and takes a tiny fraction of the time required to mine a block. An attacker attempting to change any part of the blockchain in their favour would need to "rewrite history" [33]. This means the user would need to generate a new block with their version of history to replace the original block, plus any subsequent blocks, competing against all other miners to do so. To reduce the likelihood of an attacker redoing enough proof of work to mine their fraudulent block, transactions in a block are not considered confirmed until 6 blocks have been mined since [4, 17].

As mining hardware becomes more technologically advanced, peers are able to find a suitable nonce value more quickly. To ensure that generating blocks remains a difficult task, Bitcoin recalculates the number of leading zeros for the block hash [28]. This is determined by the *target* value; the lower the target value, the more leading zeros the hash has. Therefore, decreasing the target makes finding a nonce more difficult. The Bitcoin network aims to produce a single block every 10 minutes [4, 28]. Every 2016 blocks the target is recalculated by comparing the average time it took to generate each block to the 10 minute goal, then adjusting it accordingly.

The 10 minute goal has been controversial among those keen to see Bitcoin used in mainstream fast transaction scenarios, such as vending machine payments in which goods are received a matter of seconds after payment [17]. However, the delay is necessary due to the linear structure of the blockchain. After a block is mined, it takes time for it to be broadcast to the network. Until then, other miners are actually competing against the new block instead of adding to it [4]. If a block is mined in that time, based on the previous version of the blockchain, the network can only accept one of the two, wasting considerable work. This waste is reduced by lengthening the time between blocks being produced. Nakamoto chose 10 minutes as a compromise between confirming transactions quickly and minimising wasting work [4, 25].

1.3 Report Overview

This report will describe how mathematics is utilised in Bitcoin to realise Nakamoto's aim of creating peer-to-peer electronic cash.

First, we will discuss public key cryptography in general and explore how key pairs can be used to secure information. Next, we introduce elliptic curves and their properties to describe the particular cryptographic algorithms Bitcoin uses to create keys. With this information, we describe the algorithm to create digital signatures used in Bitcoin transactions. We finally look in detail at hash functions, their mathematical properties, and suitability for use in different aspects of Bitcoin's system.

Accompanying this text is a Matlab implementation of elliptic curve cryptography. This implementation is useful for both comprehensive examples with small integers, which will be used to aid the reader's initial understanding, and to simulate key and signature generation with parameters actually used by Bitcoin, included to illustrate the algorithms as they are described. The code is discussed in full in Chapter 4.

The final chapter summarises the content of this report and important conclusions for Bitcoin, cryptography, and their applications. It also includes recommendations for further work based upon our findings.

Chapter 2

Elliptic Curve Cryptography

2.1 Public Key Cryptography

Cryptography is the study of securing information for transfer; its core objective is to allow private communications between two persons over an insecure channel [32, p. 1]. Classical cryptosystems were based on encrypting messages using a key to an unreadable format. Only those who know that key could decode and read the message [7, p. 73]. Therefore in such systems the persons, traditionally denoted Alice and Bob, need to securely exchange the key prior to communications. These are known as *symmetric key cryptosystems* since the key is used for both encrypting and decrypting. Additionally, if the key is found by an eavesdropper, the system is rendered insecure in its entirety [32, p. 155].

A clear issue here is key exchange. If Alice and Bob cannot communicate over a secure channel in the first place, how can they secretly exchange the key? This was a major logistical problem for many years [31, p. 251].

This changed in 1977[1] when a functional *asymmetric key cryptosystem* was introduced. Instead of keeping a single key secret, the idea of asymmetric keys is based on each person having a private decryption key d that cannot easily be calculated from the corresponding encryption key e. Encryption keys can then be shared publicly, hence the term *public key cryptosystem*, allowing anyone to send messages to the owner of the published key e [32, p. 155].

Such a system can be represented by a simplified example: say Alice places her message to Bob in a box and locks it with her padlock, that only she can open. It is sent to Bob, via an interceptable mail service, who clearly cannot open the box and read his message, since that would require her to send him the secret key. Instead, he

[1]It is well publicised that Diffie and Hellman put forward the idea of asymmetric keys in 1976 [10], then Rivest, Shamir, and Adleman invented the RSA system in 1977 [32, p. 155]. However, it was published in 1999 that mathematicians at GCHQ (Ellis and Cocks) who were sworn to secrecy had independently discovered the same concepts in 1970 and 1973 respectively [31, p. 280-285].

then adds his padlock to the locked box and returns it to Alice. She now removes her padlock and sends it back to Bob. Now the box has just one padlock that could not have been left there alone unless Alice removed hers, confirming that the message must be from her.

Only Bob can read the message, since only he has the padlock key, but anyone could have sent their own message to Bob without meeting him beforehand [32, p. 155]. For the system to be secure it must be too impractical for anyone to intercept the locked box, at any stage, and use it to duplicate a padlock key.

In reality, locking the padlock with a key corresponds to encrypting a message. The above example is a close analogy of the *Massey-Omura Cryptosystem*[2], a public key system for message transmission [24, p. 71]. To outline this system, and other cryptosystems in this paper, we introduce some concepts in number theory [18, p. 33].

Definition 2.1. A finite field F is a set containing a finite number of elements that satisfy the properties of associativity, distributivity, commutativity, identity, and inverse.

We will look in detail at some of these axioms above, which are important for elliptic curve cryptography, in Section 2.2.2. For now, we just need to know that $\mathbb{Z}_p^*$ is an appropriate finite field by Proposition 2.4 [32, p. 10, 243].

Definition 2.2. The set $\mathbb{Z}_p = \{0, 1, 2, ..., p-1\}$ is the set of integers modulo a prime number p.

Definition 2.3. The set $\mathbb{Z}_p^* = \{1, 2, 3, ..., p-1\}$ is the *multiplicative* set of integers modulo a prime number p.

Proposition 2.4. *For prime p, the sets $\mathbb{Z}_p$ and $\mathbb{Z}_p^*$ are finite fields in which every non-zero element α has a multiplicative inverse α^{-1}.*

First, Alice and Bob decide publicly on a finite field, in this case $\mathbb{Z}_p^*$ for prime p, to perform calculations in prior to the message exchange above. They then each choose a secret integer e such that $2 \leq e \leq p-1$ and $\gcd(e, p-1) = 1$. These elements somewhat correspond to their padlocks, e_A for Alice and e_B for Bob.

To secretly send a message m to Bob, Alice encrypts, or 'locks', it by computing m^{e_A}. This can be sent publicly, since nobody but Alice knows e_A or the inverse e_A^{-1}, equivalent to her padlock key [18, p. 100]. Bob cannot extract m either, so he encrypts this message with his 'lock', computing $m^{e_A e_B}$ and returns it to Alice. Since exponentiation is commutative, she can remove her secret element by calculating $m^{e_A e_B e_A^{-1}} = m^{e_B}$ and

[2]Proposed in 1982 by James Massey and Jim Omura based upon an earlier idea by Adi Shamir [24, p. 71]

sending it back to Bob. Now only Bob's 'lock' is left that he can remove by computing $m^{e_B e_B^{-1}} = m$.

The example assumes it is impractical to duplicate a key from a padlock; this translates to ensuring that the calculation to discover the private keys from public information is infeasible [7, p. 171].

The reason this cryptosystem is secure is that none of the secret elements m, e_A, or e_B are extractable from the publicly sent terms m^{e_A}, m^{e_B}, or $m^{e_A e_B}$. Anyone who intercepts these cannot decipher the message because they do not know the modular inverses e_A^{-1} or e_B^{-1}. Finding them would require one to solve the discrete logarithm problem [24, p. 71]. We define it as [32, p. 227]:

Definition 2.5. For a multiplicative group G, an element $a \in G$ of order n, and an element $y \in \langle a \rangle$ the Discrete Logarithm Problem (DLP) asks one to find the unique positive integer $x < n - 1$ in:

$$a^x = y. \tag{2.1}$$

The cyclic subgroup generated by a is denoted as $\langle a \rangle$, which we explain in Section 2.2.3. Informally, to solve this problem one must find x in $a^x = y$ where a and y are members of a finite group. In other words it states that, in suitable finite groups, exponentiation is a one way function [32, p. 227].

The relationship between public and private keys is based upon these 'one-way functions' [32, p. 156, 18, p. 85]. This term refers to functions that are easy to compute, but difficult to invert. This means there is an algorithm to calculate a one-way function 'easily' in polynomial time and no such algorithm to calculate the inverse of the function, so we say the inverse is 'difficult'.

For example, the widely used 1977 RSA cryptosystem is based upon the difficulty of factoring large integers. The function here is the multiplication of two large primes, which is easy computationally, and the inverse is finding those prime factors from the result, which is difficult to compute [18, p. 85].

To generate keys, Bitcoin uses a variant of the *Diffie-Hellman Key Exchange Protocol*[3] [8]. Again, Alice and Bob use a publicly known finite field $\mathbb{Z}_p^*$ and choose a fixed element g, which is also shared publicly. This element is usually a generator of $\mathbb{Z}_p^*$, but not necessarily [18, p. 99].

Alice secretly chooses a random integer a such that $2 \leq a \leq p - 1$ and computes g^a. Since she has performed exponentiation in a finite group, she can freely send g^a to Bob without an interceptor finding a. Bob, in the same way, also chooses a secret random number b and sends Alice g^b. They can now apply their respective secret number and use the secret key g^{ab} [24, p. 49]. Anyone attempting to find the secret key from public

[3]Full algorithm, as used by Bitcoin, in Section 2.3.

information faces an infeasible calculation due to the Diffie-Hellman assumption [10, 18, p. 99].

Definition 2.6. The Diffie-Hellman Assumption (DHA) states that, for a finite group G, an element $g \in G$, and integers a and b, it is computationally infeasible to compute g^{ab} knowing only g^a and g^b.

If the discrete logarithm problem can be solved, then the Diffie-Hellman assumption fails. Therefore, the DHA can be considered as strong as the DLP and in some cases they are considered to be equivalent [24, p. 49]. However, it is not known whether the converse holds; that if the DHA is untrue then discrete logarithms can be efficiently computed [18, p. 99].

Elliptic curve cryptography is secure due to the difficulty of the DLP [15]. We shall prove in Section 2.2.2 that an elliptic curve is a multiplicative finite group and an analogous assumption to the DHA can be applied. We will then modify our discrete logarithm definition for elliptic curves in Section 2.2.3.

The above asymmetric key cryptosystems are used to authenticate messages [18, p. 88]. One such method is known as the *Digital Signature Algorithm*, which we can illustrate informally [32, p. 289].

Let Alice have the encryption key e, which is public, and the decryption key d, which is known only to her. Alice wants to broadcast some message M publicly while signing it to ensure everyone knows it was her who wrote it. She hashes M (recall the important properties of hashing discussed in the previous section), then applies her secret *decryption* key d to the hash [33]. This, now encoded, text forms her signature and is published along with M.

Anyone can use Alice's encryption key e to reverse the encoding on her signature, revealing the hashed message [18, p. 89]. By hashing M and checking that it and the decoded signature are the same, one has confirmed two things. First, that the message certainly came from Alice because only she could use the decryption key, and second, that the message could not have been tampered with thanks to the use of hash functions [18, p. 89, 33].

Bitcoin uses this method to sign transactions. The algorithms to generate keys and signatures are variants of the Diffie-Hellman public key protocol and the above digital signature algorithm respectively [10]. What follows is an explanation of the group properties of elliptic curves we require to then describe these algorithms.

2.2 Elliptic Curves

2.2.1 Definition

Bitcoin uses Elliptic Curve Cryptography (ECC) to create keys and digitally sign transactions. It was first proposed[4] as a progression of more traditional cryptography systems, such as RSA which is based on prime factorisation. It has increased in popularity due to the low requirement of storage space and processing power [8]. We will first introduce elliptic curves defined over $\mathbb{R}$ and later explain their suitability for cryptography when defined over a finite field, in particular $\mathbb{Z}_p$.

Definition 2.7. An elliptic curve E is the set of points where $a, b \in \mathbb{R}$ are chosen constants that satisfy $4a^3 + 27b^2 \neq 0$:

$$E = \{(x, y) \in \mathbb{R}^2 \mid y^2 = x^3 + ax + b\} \cup \{O\}. \tag{2.2}$$

O is the point at infinity which will play the role of the identity group element in Section 2.2.2. The equation $y^2 = x^3 + ax + b$ is known as the Weierstrass Normal Form of an elliptic curve [34]. The assumption that $4a^3 + 27b^2 \neq 0$, (the discriminant is non-zero), is necessary and sufficient to exclude singular curves [32, p. 248]. If the discriminant is negative, the graph of the curve will have one smooth component while if it is positive, it will have two components. Below (*Figure 2.1*) are three curves with negative discriminant, positive discriminant, and zero discriminant respectively. The rightmost curve is therefore singular and has a 'crossover' point, which is unsuitable for cryptographic use.

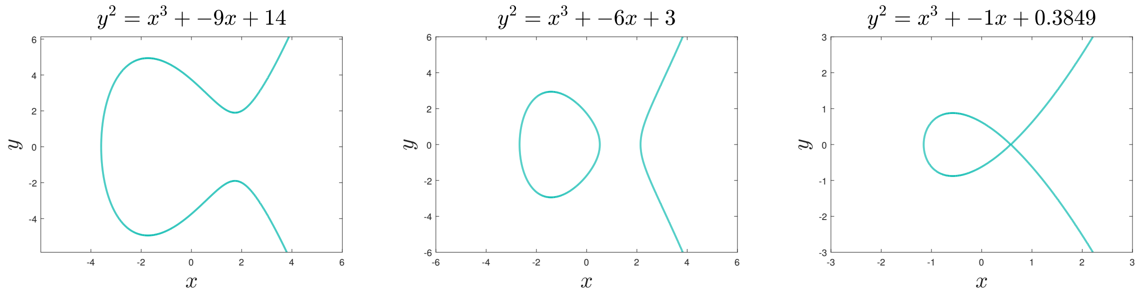


Figure 2.1: Examples of elliptic curves over $\mathbb{R}$

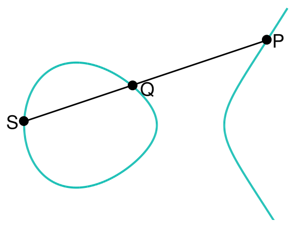
To show that E is a group, and apply the Diffie-Hellman Assumption (*Definition 2.6*), we must define a group operation. First, we look at a a straight line L intersecting the curve E at two points. It will also intersect E at a third point that we can find algebraically if we know the initial two points. This may not be the case with singular

[4]Independently introduced by Victor Miller and Neal Koblitz in 1985 [15].

curves or if the line is vertical [12]. We exclude singular curves and consider a vertical line of the form $x = v, v \in \mathbb{R}$ as a special case below.

Suppose our two initial points on L and in E have the real coordinates (x_1, y_1) and (x_2, y_2) respectively. We consider the following cases to find the third point with coordinates (x_3, y_3) on L intersecting our curve [32, p. 248]:

- $x_1 \neq x_2$:
  Algebraically, we have a line L with the equation $y = \lambda x + v$ which intersects the curve at three distinct points. We can define the slope λ and constant v in terms of the two points that we know:



$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \quad v = y_1 - \lambda x_1 = y_2 - \lambda x_2. \tag{2.3}$$

We can now express y in terms of our known coordinates by substituting into (2.2):

$$\begin{aligned} y^2 &= (\lambda x + v)^2 = x^3 + ax + b, \\ 0 &= x^3 - \lambda^2 x^2 + x(a - 2\lambda v) + b - v^2. \end{aligned} \tag{2.4}$$

The equation (2.4) is cubic and, since we already know that roots x_1 and x_2 are real and distinct, has three real roots. These roots are the points at which L intersects our elliptic curve. We find x_3 using the quadratic term in (2.4):

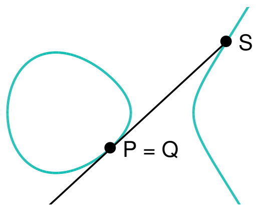
$$-(-\lambda^2) = x_1 + x_2 + x_3 \quad \Longrightarrow \quad x_3 = \lambda^2 - x_1 - x_2. \tag{2.5}$$

Now we have x_3, finding y_3 is trivial using the definition of the slope:

$$\lambda = \frac{y_3 - y_1}{x_3 - x_1} \Longrightarrow y_3 = \lambda(x_3 - x_1) + y_1. \tag{2.6}$$

Due to the symmetry of an elliptic curve in the x-axis (*see Figure 2.1*), when $x_1 = x_2$, either $y_1 = y_2$ or $y_1 = -y_2$.

- $x_1 = x_2$, $y_1 = y_2$:
  Here we consider a case where the two 'known points' on our line are the same point. Geometrically, we can think of this as when some point Q approaches point P, the line connecting them becomes the tangent to the curve [8].

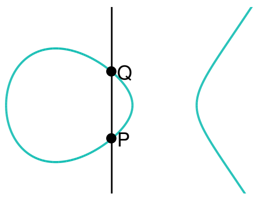


  This is how we consider the line L. Therefore, we must modify our slope λ, assuming $y_1 \neq 0$ (since this could be considered as $y_1 = -y_2$, the next case). We differentiate (2.2) implicitly:

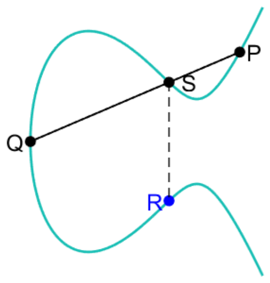
$$2y\frac{\partial y}{\partial x} = 3x^2 + a \qquad \implies \qquad \lambda = \frac{3x^2 + a}{2y}. \tag{2.7}$$

Since we have only one initial point, we can substitute (x_1, y_1) into the above equation for λ. As previously, we still have L defined as $y = \lambda x + v$, so our formulae for x_3 and y_3 are identical.

- $x_1 = x_2$, $y_1 = -y_2$:
  Our final case considers two points that reflect in the x-axis. Since L, passing through these two points, is simply in the form $x = v$, we do not have a real third point. We can, however, say that third point is actually O, the point at infinity, as explained further below.



2.2.2 Abelian Groups



We can define a binary operation such that E becomes an abelian group. Given points P and Q in E, let L be the line connecting them. We know from the previous section that if L intersects the curve E in two points, it will also intersect in a third point, which we will denote by S.

Point addition is defined as $P + Q + S = O$. Therefore, the sum of P and Q is equal to $-S$, which is the reflection of S about the x-axis. If S has coordinates (c, d), then $R = -S$ has coordinates $(c, -d)$, so we have $P + Q = R$.

Definition 2.8. A set $(G, +)$ is an abelian group if its elements satisfy the properties of closure, associativity, identity, inverse, and commutativity under the operation $+$.

Proposition 2.9. *An elliptic curve under point addition as defined above is an abelian group.*

We summarise our equations in the previous section below. For $y_1 \neq -y_2$, where $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, we find λ thus:

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{for} \quad x_1 \neq x_2, \\ \frac{3x^2 + a}{2y}, & \text{for} \quad x_1 = x_2, \quad y_1 = y_2 \neq 0. \end{cases} \tag{2.8}$$

An explicit formula for $S = (x_3, y_3)$ is given by:

$$x_3 = \lambda^2 - x_1 - x_2, \tag{2.9}$$

$$y_3 = \lambda(x_3 - x_1) + y_1. \tag{2.10}$$

Note that here we have found $S = (x_3, y_3)$ where $P + Q = R = (x_3, -y_3)$.

We have considered all cases apart from $y_1 = -y_2$. Here, we again consider a vertical line L in the form $x = v$ that does not intersect a third point we can see. In this case, we define $P + Q = P + (-P) = O$.

We can prove Proposition 2.9 for E with points P, Q and S defined above:

1. **Closure:** From (2.9) and (2.10) we know our third point, S, on L is in E. By (2.2), it is clear from the squared term that for each x we will have two y coordinates, $-y$ and y. Since S exists in E, so does $-S = R$. Therefore, we have $P + Q = R \in E,\ \forall\, P, Q \in E$ satisfying the closure property.

2. **Associativity:** In terms of our algebraic equations, proving associativity is a rather long and difficult process that will not be considered here[5] [32, p. 250]. However, for all cases the associativity property is satisfied: $(P + Q) + S = P + (Q + S)$ [8].

3. **Identity:** We define $P + O = O + P = P$, where O is the point at infinity, $\forall P \in E$. This satisfies the identity element property.

4. **Inverse:** Addition was defined such that $\forall P \in E$, we have $P + (-P) = O$. If $P = (x_1, y_1)$ is on the curve, so is $-P = (x_1, -y_1)$ due to the square in (2.2). Therefore, the inverse of P always exists.

5. **Commutativity:** We have defined our three aligned points irrespective of order. Our point addition formulae are still valid since for $x_1 \neq x_2$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{y_1 - y_2}{x_1 - x_2}, \quad v = y_1 - \lambda x_1 = y_2 - \lambda x_2. \tag{2.11}$$

   For $x_1 = x_2$, $y_1 = y_2$, we clearly have only one point, so order does not matter (2.7). We know that for $x_1 = x_2$, $y_1 = -y_2$: $P = -Q$. In terms of our group operation, P and Q are inverses, so no matter what order they are added in we will always have $P + Q = P + (-P) = O$. Therefore, we now have $P + Q = Q + P = R$ for all cases, proving that our group operation is commutative.

[5]A proof by Stefan Friedl that uses elementary algebra is cited in the bibliography [12].

2.2.3 Elliptic Curves over $\mathbb{Z}_p$

We have just seen how elliptic curves defined over $\mathbb{R}$ form an abelian group with the binary operation point addition. These curves defined over $\mathbb{Z}_p$ are central to Bitcoin's application of elliptic curve cryptography.

Let $p > 3$ be a prime, then $\mathbb{Z}_p$ is a finite field (*Proposition 2.4*) with p elements: $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ [32, p. 243]. We now modify our definition of an elliptic curve (2.2).

$$a, b \in \mathbb{Z}_p,\ 4a^3 + 27b^2 \neq 0: \quad y^2 \equiv x^3 + ax + b \mod p. \tag{2.12}$$

We define E as the set of all solutions to (2.12) along with the point O. In $\mathbb{R}$, calculating point addition is graphically clear. However, in $\mathbb{Z}_p$ E is a set of isolated points (*see Figure 2.2*).

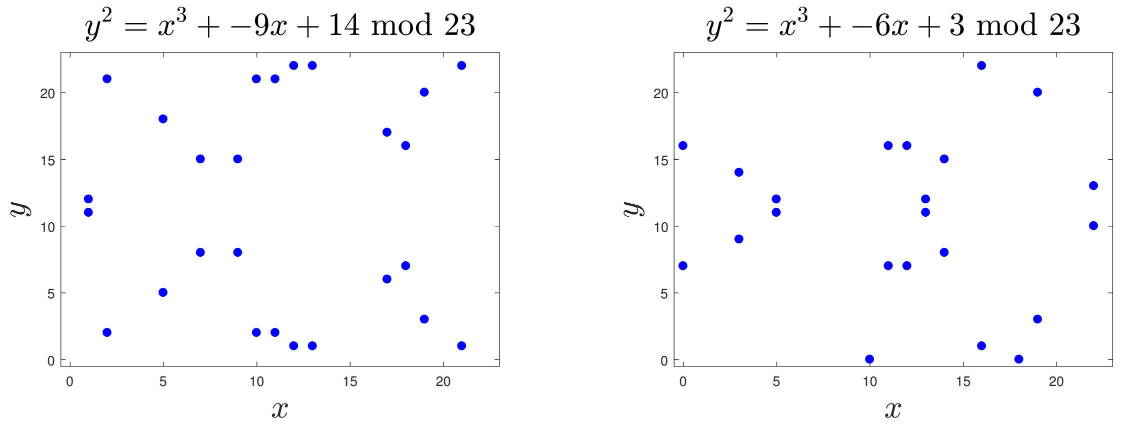


Figure 2.2: Examples of elliptic curves over $\mathbb{Z}_{23}$

We can define the line L as the set of points that satisfy $y - \lambda x - v \equiv 0 \mod p$ (note that this is simply the standard line equation we used previously). Then, for our defined points P, Q and $S = -R$ on L, the point addition formulae (2.8), (2.9) and (2.10) are still valid when calculated modulo p. We note that $P + Q = R = -S = (x_3, -y_3) \mod p$. As before, when $y_1 = -y_2 \mod p$, P is the inverse of Q in E, so $P + Q = P + (-P) = O$. Otherwise, we have:

$$\lambda = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} \mod p, & \text{for} \quad x_1 \neq x_2 \mod p, \\ (3x^2 + a)(2y)^{-1} \mod p, & \text{for} \quad x_1 = x_2, \quad y_1 = y_2 \neq 0 \mod p. \end{cases} \tag{2.13}$$

An explicit formula for $S = (x_3, y_3)$ is given by:

$$x_3 = \lambda^2 - x_1 - x_2 \mod p, \tag{2.14}$$

$$y_3 = \lambda(x_3 - x_1) + y_1 \mod p. \tag{2.15}$$

Proposition 2.10. *An elliptic curve* (2.12) *over* $\mathbb{Z}_p$ *with the operation point addition defined above is an abelian group.*

Without such a comprehensive graphical representation, it may seem difficult to compute point addition in this field. However, rather notably, the equations (2.8), (2.9), and (2.10) remain valid in *every field* when $p \neq 2, 3$ [12].

Example 2.11. For example, take the elliptic curve $y^2 \equiv x^3 + 9 \mod 7$ (*see Figure 2.3*). Here we have $a = 0$, $b = 9$ and $p = 7$ from the form defined in (2.12). Let $P = (3, 6)$ and $Q = (5, 1)$. To compute $P + Q = R$ we simply use the formulae above. First we find λ by substituting our coordinates into (2.13):

$$\lambda = (y_2 - y_1)(x_2 - x_1)^{-1} = (1 - 6)(5 - 3)^{-1} = (2)(2)^{-1} \mod 7 \equiv 1.$$

Now we can find $S = (x_3, y_3)$ using (2.14) and (2.15):

$$x_3 = 1^2 - 3 - 5 = -7 \equiv 0 \mod 7, \qquad y_3 = 1(0 - 3) + 6 \equiv 3 \mod 7.$$

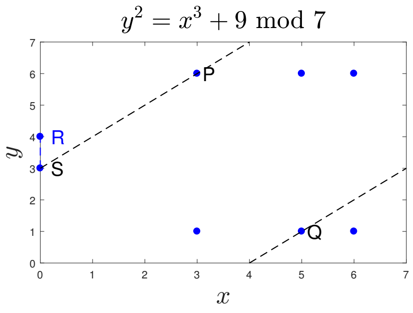


Figure 2.3: Point addition over $\mathbb{Z}_7$

Finally, $R = -S = (0, -3) = (0, 4) \mod 7$. This point addition is represented in Figure 2.3; the dashed line is L that connects points P, Q and S.

From this point we will only be considering $P, Q \in \mathbb{Z}_p^2 \cup \{O\}$, therefore all equations will be over mod p, which is omitted for simplicity.

As we are working with the finite field $\mathbb{Z}_p$, multiplying a point P by a scalar k is defined as the repeated point addition of P to itself. Computing $kP = Q$ can be completed efficiently by adding multiples of P instead of adding P k times. For example, to find $5P$ we add P to itself twice, so we have the points P, $2P$ and $3P$. We simply compute $3P + 2P = 5P$, finding point $5P$ in just three additions.

This method is known as the *Double and Add Algorithm* and has complexity $\mathcal{O}(\log k)$ [8]. For a point P on an elliptic curve and an integer k in binary form of l digits, we can compute kP thus [32, p. 258]:

Algorithm 1: Double and Add

```
$Q \leftarrow O$
for $i \leftarrow l$ to $0$ do
    double $Q$
    if $k_i = 1$ then
        $Q \leftarrow Q + P$
    end
end
return $Q$
```

In binary form, 5 is written as 101, so for the first iteration we double $Q = O$ (essentially doing nothing) and assign $Q = O + P = P$. The next iteration corresponds to a 0 digit, so we only double Q and have $Q = 2P$. Finally, we have a 1 digit so we again double Q, now $Q = 4P$, and add P, finding $5P$ in three iterations.

In general $uP + vP = (u + v)P$ for integers u and v [8]. Though this may seem trivial, it shows that multiples of P are closed under addition, forming a subgroup of E.

Definition 2.12. For $P \in E$, the set:

$$\langle P \rangle = \{kP : k \in \mathbb{Z}_p\}, \tag{2.16}$$

is called the cyclic subgroup generated by P.

Our $\langle P \rangle$ is of size n, some positive integer, where the nth multiple of P is O, completing the cyclic subgroup. In other words, $(n + 1)P = P$.

Point multiplication is crucial for ECC because finding k in $kP = Q$ is an intractable, or difficult problem, for large k. However, calculating Q from k and P is easy using repeated addition. This is the aforementioned Elliptic Curve Discrete Logarithm Problem [8, 32, p. 254]. Interestingly, it is simply a common belief that the problem is difficult and there is no mathematical proof [8]. We can now modify our Definition 2.5:

Definition 2.13. For an elliptic curve E, an point $P \in E$ of order n, and a point $Q \in \langle P \rangle$ the Elliptic Curve Discrete Logarithm Problem asks one to find the unique positive integer $k < n - 1$ in:

$$kP = Q. \tag{2.17}$$

We can also relate elliptic curves to the Diffie-Hellman assumption (*Definition 2.6*): given three points P, uP and vP, it is computationally infeasible to compute uvP [8].

2.2.4 Calculating Group and Subgroup Order

To ensure that finding k from $kP = Q$ is sufficiently difficult, we need a large n, that is to say, a subgroup of E generated by P with many points. To find such an n, we first need the order of E. Graphically, it is clear that an increase in the base p yields an approximately proportional increase of points in E.

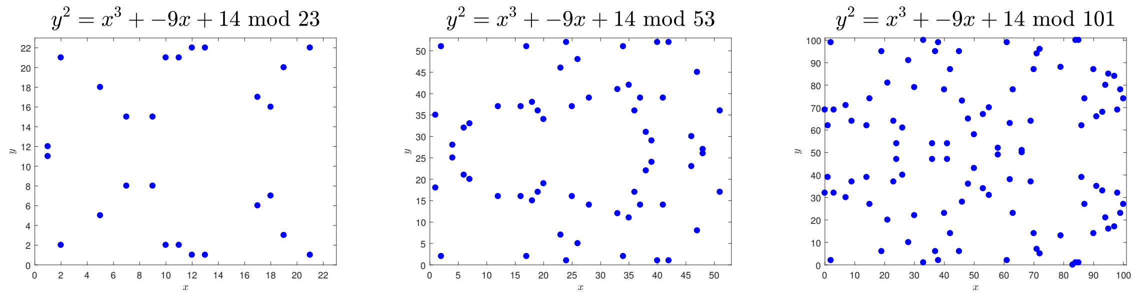


Figure 2.4: Elliptic curves over $\mathbb{Z}_{23}$, $\mathbb{Z}_{53}$, and $\mathbb{Z}_{101}$

This is confirmed by Hasse's theorem [32, p. 254].

Theorem 2.14 (Hasse)**.** *For an elliptic curve E defined on the finite field $\mathbb{Z}_p$,*

$$p + 1 - 2\sqrt{p} \leq |E| \leq p + 1 + 2\sqrt{p}. \tag{2.18}$$

Hasse shows that an elliptic curve over $\mathbb{Z}_p$ has roughly p points. For instance, a curve over $\mathbb{Z}_{53}$ has between 40 and 68 points. There exists an efficient, though complex, algorithm to compute the order of E known as Schoof's Algorithm[6] [8]. Its complexity in terms of operations in $\mathbb{Z}_p$ is $\mathcal{O}((\log p)^6)$ [32, p. 254].

With the order of E, which we will denote as N, we can find n, the order of $\langle P \rangle$. We know that n divides N, due to Lagrange's Theorem [7, p. 44, 32, p. 164].

Theorem 2.15 (Lagrange)**.** *For a finite group G of order N and a generator $g \in G$, the order of g divides N.*

For each divisor of N, an integer l, we compute lP starting from the smallest l since the smallest l such that $lP = O$ is our subgroup order n [8].

A special case to consider is when N is prime, so clearly our divisors are only 1 and N. When $n = 1$, the subgroup must contain only O whereas when $n = N$, the subgroup must contain all points and therefore is identical to E.

As mentioned, we require a large n to ensure finding k from $Q = kP$ is difficult. Instead of choosing P, we choose a large divisor n from N and find an appropriate P from there. Since $n|N$, we know $N/n = h$, the cofactor of the subgroup, is a positive integer. Therefore:

$$n(hP) = NP = O, \tag{2.19}$$

[6]René Schoof discovered the first polynomial time algorithm for $|E|$ in 1985 [18, p. 189, 8].

proving that some point hP generates a subgroup of order n when $hP \neq O$. Let the point $hP = G$ that we can find thus:

- Compute $|E| = N$ using Schoof's Algorithm
- Find a prime divisor, n, of N
- Compute the cofactor $h = N/n$
- Choose a random point, $P \in E$, and compute $G = hP$
- Repeat the last step until a non-O G is found

This G is a generator of a subgroup of E with order n and cofactor h. Note we find a prime n to ensure that the true order of G is n, not a divisor of n.

Example 2.16. We can use the curve $y^2 = x^3 - 9x + 14 \mod 101$ (rightmost graph of Figure 2.4) and find a generator G. The function `Gnh` (discussed in Chapter 4) computes this from the curve parameters, $a = -9$, $b = 14$, and $p = 101$.

This curve has order $N = 104$ which has prime factors 2 and 13. The function randomly chooses $n = 13$, so our cofactor is given by $h = N/n = 8$. The point $(70, 87)$ is also chosen randomly as P. Using the double and add algorithm, we find $hP = G = (61, 99)$. We know that this point is a generator of a cyclic subgroup of order 13. In other words, multiplying our G by any integer will result in one of 13 points in the subgroup.

2.3 Key Exchange

We now have the parameters required to describe ECC as it is used by Bitcoin. We use the sextuple (p, a, b, G, n, h) described thus:

- **p**, the prime size of the finite field $\mathbb{Z}_p$,
- **a** and **b**, constants defined by the Weierstrass form of E over $\mathbb{Z}_p$,
- **G**, the generator of our subgroup such that $G = hP$ as defined above,
- **n**, the order of our subgroup $\langle G \rangle$,
- **h**, the cofactor of our subgroup $\langle G \rangle$.

Here we use a variant of the Diffie-Hellman Key Exchange Protocol to produce keys [26]. This algorithm produces an individual private key and public key and a shared secret key for both the sender and receiver of traffic, as outlined in Section 2.1. Recall that this protocol relies upon difficult problems. The believed difficulty of the elliptic curve discrete logarithm problem (*Definition 2.13*) means that the calculation to find the private key from the public key is infeasible. It is important to note that this algorithm only produces keys that two people can later encrypt traffic with; it does not secure any such traffic [8].

Our private key d is a random integer such that $1 \leq d \leq n-1$ and our public key is $H = dG$. The process for Alice and Bob to agree on keys in terms of (p, a, b, G, n, h) is described thus:

- Alice chooses her private key, d_A, and public key, $H_A = d_A G$,
- Bob chooses his: d_B and $H_B = d_B G$,
- Alice and Bob exchange their public keys, H_A and H_B, over an insecure channel,
- Alice calculates $S = d_A H_B$,
- Bob calculates $S = d_B H_A$.

Now S is a secret point in $\langle G \rangle$ only Alice and Bob know. Both of them receive the same S since the multiplication of an integer, d_A, and a point, $d_B G$, is commutative by our group laws (it is simply repeated addition d_A times). This can be represented:

$$S = d_A H_B = d_A(d_B G) = d_B(d_A G) = d_B H_A. \tag{2.20}$$

An eavesdropper has the sextuple, (p, a, b, G, n, h), and the public keys, H_A and H_B, but cannot find S. The secret point S is safe due to the Diffie-Hellman assumption (*Definition 2.5*) and the keys d_A, d_B are safe due to the elliptic curve DLP (*Definition 2.13*) [8]. Of course, if the eavesdropper manages to solve the DLP and retrieve the private keys, they can also find S using point multiplication and therefore break the DHA [7, p. 189]. In other words, if the DHA is untrue then S can easily be found. However, if the DLP can be solved efficiently then S *and* the keys d_A, d_B can easily be found.

Example 2.17. Using the curve in Example 2.16, we have the parameters $(p, a, b, G, n, h) = (101, -9, 14, (61, 99), 13, 8)$. The Matlab function `btckeyprodall` carries out the above algorithm to generate key pairs. The private key d_A is randomly chosen as 6 and $H_A = d_A G = (62, 63)$ is calculated using the double and add algorithm as before. Similarly, we find $d_B = 9$ and $H_B = d_B G = (7, 71)$. Now we can check for the secret point S; the point multiplication function underlying the Matlab algorithms, `PMulD`, gives $d_A H_B = (100, 27)$ and $d_B H_A = (100, 27)$ as expected.

2.4 Digital Signatures

Bitcoin uses elliptic curve cryptography to digitally authenticate transactions [8, 25]. We use the same parameters (p, a, b, G, n, h) however here Alice wants to sign a non-secret message with her private key d_A and Bob can validate it with her public key H_A.

The use of these parameters ensures that only Alice will be able to produce valid signatures, since only she knows d_A, while anyone can validate it with the shared H_A without finding the private key. We introduce a new parameter, z, which is the hash of the message Alice would like sign. We outline this hash in detail in Chapter 3; for now it is only important to know that z is a positive integer. Signature creation can be described thus:

- Alice chooses a secret random positive integer $k < n$ and computes $kG = P$,
- Alice then calculates $r = x_p \mod n$, where x_p is the x coordinate of P,
- If $r = 0$, she must choose another k and repeat the process,
- Alice calculates $s = k^{-1}(z + rd_A) \mod n$,
- If $s = 0$, Alice must choose another k and repeat the process.

Alice now has the double (r, s), which is the signature for z [8]. Anyone can now verify this signature with G, H_A, the hashed message z, and (r, s):

- Compute $u_1 = s^{-1}z \mod n$,
- Compute $u_2 = s^{-1}r \mod n$,
- Compute the point $P = u_1G + u_2H_A$,
- Finally, verify that $r = x_p \mod n$.

Without the secret k or d_A, any observer can discover x_p and compute r to confirm the signature is valid. We can see how this works by defining P in terms of H_A and the integers u_1 and u_2 [8]:

$$\begin{aligned} P &= u_1G + u_2H_A \\ &= u_1G + u_2d_AG \\ &= (u_1 + u_2d_A)G \implies = (s^{-1}z + s^{-1}rd_A)G \\ &= s^{-1}(z + rd_A)G = kG. \end{aligned} \tag{2.21}$$

Note that the integer k is, like d_A, not easily found by an eavesdropper since it is used as a multiplier to produce a new point, P. In fact, it is crucial that k remains secret and changes with each new signature. If Alice created two signatures using the same value of k, then d_A could be extracted with some simple algebra [8]. Say we have two signatures, (r_1, s_1) for z_1 and (r_2, s_2) for z_2, produced by Alice with a static k. We have the same value of $P = kG$, therefore $r_1 = r_2 = r$ and we can derive:

$$\begin{aligned} s_1 &= k^{-1}(z_1 + rd_A) \mod n, & s_1 - s_2 &= k^{-1}(z_1 - z_2) \mod n, \\ s_2 &= k^{-1}(z_2 + rd_A) \mod n, & k &= (z_1 - z_2)(s_1 - s_2)^{-1} \mod n. \end{aligned}$$

With k, we can extract Alice's private key[7] in a computationally easy way. We simply rearrange our s_1 definition to $d_A = r^{-1}(s_1 k - z_1) \mod n$.

Example 2.18. We can continue with our example curve $y^2 = x^3 - 9x + 14 \mod 101$ and public parameters $(p, a, b, G, n, h) = (101, -9, 14, (61, 99), 13, 8)$. In Example 2.17, we found the key pair, $d_A = 6$ and $H_A = (62, 63)$, which can be used to create a digital signature.

The Matlab function `sigc` creates a signature from the perspective of Alice using the public sextuple, d_A, and the hash z. Say Alice wants to sign $z = 9999$. Following the signature algorithm above, the function chooses $k = 11$ randomly and finds:

$$\begin{aligned} P = kG &= 11 \cdot (61, 99) = (100, 74), \\ r = x_p \mod n = 100 &\mod 13 \equiv 9, \\ k^{-1} = 11^{-1} &\mod 13 \equiv 6, \\ s = k^{-1}(z + rd_A) \mod n = 6(9999 + 9 \cdot 6) &\mod 13 \equiv 11. \end{aligned}$$

We have the signature $(r, s) = (9, 11)$ for the message $z = 9999$. Now anyone can verify this signature was from Alice by using the public key H_A. The function `sigv` uses the public curve parameters, the signature (r, s), and the message z to check the validity. From the method above, it finds:

$$\begin{aligned} s^{-1} = 11^{-1} &\mod 13 \equiv 6, \\ u_1 = s^{-1}z = 11 \cdot 9999 &\mod 13 \equiv 12, \\ u_2 = s^{-1}r = 11 \cdot 9 &\mod 13 \equiv 2, \\ P = u_1 G + u_2 H_A &= 12 \cdot (61, 99) + 2 \cdot (62, 63) = (100, 74), \\ 100 &\mod 13 \equiv 9 = r. \end{aligned}$$

In this case we are using small numbers for ease of presentation, so it is coincidental that $s^{-1} = k^{-1} = r = 6$. The above calculations demonstrate that one does not require any private information to prove that a digital signature is valid.

[7]Sony made the mistake of using a static k in their signature scheme [8]. In 2011, their private key to authenticate Playstation 3 games was extracted in a well-publicised hack using the method above.

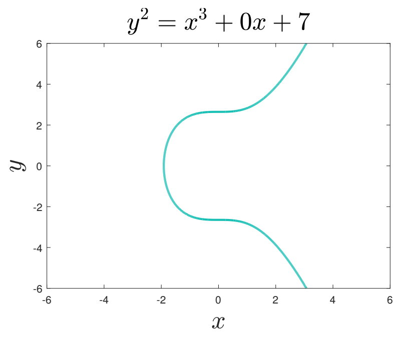


Figure 2.5: `secp256k1` over $\mathbb{R}$

Bitcoin uses the curve known as `secp256k1`[8] [4, 6]. This corresponds to $y^2 = x^3 + 7 \mod p$, so $a = 0$ and $b = 7$. This graph is plotted over real numbers in Figure 2.5. Of course, in reality the curve over a very large finite field and is of no use to plot accurately. Due to these large integers, we use hexadecimal format for some parameters:

$$
\begin{aligned}
p &= \texttt{fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f},\\
a &= 0,\\
b &= 7,\\
G &= (\texttt{79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798},\\
&\qquad \texttt{483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8}),\\
n &= \texttt{fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141},\\
h &= 1.
\end{aligned}
$$

Example 2.19. With the Matlab function `btckeyprodallVPI` we can simulate key creation with parameters used by Bitcoin above. Following the algorithm in Section 2.3, we find two key pairs:

$$
\begin{aligned}
d_A &= \texttt{662c280e5084611d2d487aff14ccd9d379e9f955f0e8690d2f2902d6e12b94c6},\\
H_A &= (\texttt{0a019541f7807004d734108a70e66544bc9c2cc70f806438582922cf24a15def},\\
&\qquad \texttt{9423b0b9531b6e0bb4809823ff06cb905e4cce49b61fa0f23669ea253157db0c}),\\
d_B &= \texttt{ce9ac57ad041afb8d4b472fd1febb15097d596a312dcb719a5f4126d796d7217},\\
H_B &= (\texttt{c353e7c6c830d39a2e591082c8e1fd8a61f1e1698a5e1e2ab02786485968d2a1},\\
&\qquad \texttt{bf41153b8dc67f0136096617d61fb41d9b10d1afd3ee915dea7699f89ddde7a0}).
\end{aligned}
$$

These key pairs can be used as real Bitcoin private keys and addresses. Now, with the function `sigcVPI`, we can create a signature as Alice. As an example, we take the hash[9] of the sentence "A Bitcoin Transaction" to be our z. With this, the public sextuple,

[8]This curve was specifically chosen by Nakamoto for its useful properties. The `256` in `secp256k1` refers to the bit size and the `k` stands for Koblitz curve, a curve with parameters found to allow efficient calculations without compromising security [6].

[9]The SHA-265 hash is used here, discussed in detail in Chapter 3.

and the private key d_A we create the digital signature:

$r =$ `d65c818c5d9975e1953f31b75f6b58351b2da09ecaf58af866209e82a2a8d0d6`,

$s =$ `b7fc7bfacce2b3aa3a91a1bdefa927fdbc09407be5f1620a0249dcf32a25c8b6`, for

$z =$ `b6c1299d842891bf294a3e5aed88147e76f5372aecdc23b69e227daaad6d193a`.

Anyone can check the validity of this signature using public information. With the function `sigvVPI` we find it to be valid for the hash z, showing mathematically that the owner of d_A must have signed z. Additionally, by hashing "A Bitcoin Transaction" we find z, confirming that the original message is intact.

Chapter 3

Hash Functions

3.1 Definition

A hash function is a type of one-way computation that converts an input of arbitrary length to a string of fixed length. Most commonly, the input string is very large and the hash function is used to shorten it considerably [33]. We have already explored the important 'fingerprint' property; a hashed piece of data can be considered a unique assurance of its identity [32, p. 117]. If the data is altered only slightly, then its hash will be completely different. For this reason, hash functions are extremely useful for ensuring publicly stored information is not tampered with by allowing the owner to occasionally recompute the hash and compare it to the previous hash.

For example, using the SHA-256 hash function[1] we input the sentence "Mathematics of Bitcoin". Its output is:

```
dde21b0e792f0bbf7742398c154c679709b764789caadd9203e314678ef24bb
```

Now we change the input to "Mathematics of Bitcoin.", adding only a full stop:

```
30468de5f7ca2a804525848dcd4852e0520ab85542f6fa9a1384c6e2e84c7b3
```

Clearly, the hashes are completely different and it is easy to check that the input data has been altered [33].

To explain the particular hash functions used by Bitcoin and their purposes, we first use the following definition [18, p. 89, 32, p. 119]:

Definition 3.1. A hash function is a map $f : x \to h$ that is easy to apply and infeasible to reverse, in that the only way to produce a valid pair (x, h) is to choose x and compute $f(x) = h$.

[1]Example hashes in this paper are generated using the tools at https://www.conversion-tool.com/.

When used in conjunction with a cryptographic signature, one must ensure the hash function does not weaken the security of the signature scheme. This is because the hashed message is signed, not the original message itself (h, not x).

3.2 Security

For a hash function to be considered secure, the following problems should be computationally infeasible to solve [32, p. 119]. We have a hash function f, the input data x, and the output hash h:

- **Preimage:** finding x from h. Solving Preimage for a given h allows one to produce a valid pair (x, h) without first choosing x. Since this is essentially reversing f, a hash function for which Preimage cannot be efficiently solved is known as *one-way* or *preimage resistant*.
- **Second Preimage:** finding x' from x and h such that $x' \neq x$ and $f(x) = f(x')$. Given some input data x it should be difficult to find another input x' that results in the same hash. If this can be done, then the pair $(x', f(x))$ is valid. If it cannot be solved efficiently, then f is said to be *second preimage resistant*.
- **Collision:** finding x' and x from f such that $x' \neq x$ and $f(x) = f(x')$. Note that we are not given any inputs, only the hash function itself (unlike Second Preimage). It should be difficult to determine two distinct inputs that yield the same hash. Though solving Collision does not give one a valid pair, one can simply compute $f(x)$ and find the pair $(x', f(x))$. A hash function for which Collision cannot be solved efficiently is known to be *collision resistant*.

A more comprehensive way to think of Collision is to imagine the hash function $f(x)$ as the birthday of person x, where x is any living person. Finding two people with the same birthday[2] is equivalent to finding a collision for this f [32, p. 123].

The above problems describe different methods to break the statement in Definition 3.1. Finding a valid pair (x, h) without first choosing x and computing $f(x) = h$ can allow one to tamper with the original data undetected or forge a digital signature [32, p. 279]. For our purposes, we will look at the latter situation, letting x be the original message, $f(x) = h$ be the hashed message, and $sig(h) = y$ be the signed hashed message for some digital signature algorithm sig. Now the pair (x, y) is a valid signature.

A simple attack is the *Known Message* attack. Given the pair (x, y), an attacker hashes the original message to find $f(x)$. If they are able to find a different message x'

[2]The mathematics behind the famous Birthday Paradox (the total number of people required to have two of them probably share a birthday is much lower than expected) can be exploited in a *Birthday Attack* that solves Collision [24, p. 186].

that gives $f(x') = f(x)$ then Second Preimage has been solved. Now, (x', y) appears to be a valid signed message though y is actually forged for the message x' [32, p. 280]. The user has managed to create a valid signature without using sig.

Another attack known as *Chosen Message* involves finding two distinct messages $x' \neq x$ and $f(x) = f(x')$. If the attacker can convince the original sender to sign the message x, obtaining y, then they can also use the forged signed message (x', y) [32, p. 280]. In this case the attacker has solved the Collision problem.

A *Key-only* attack relies upon certain vulnerabilities of the signature scheme sig. Now, say an attacker manages to compute a signature y on a random hashed message h and finds another message x', where $h = f(x')$. The attacker has the signed message (x', y) which appears to be valid but the signature is forged for the message x'. The attacker found x' from h, therefore solving the Preimage problem [32, p. 280]. Here, we must assume that the signature scheme is susceptible to to the attacker creating a signature for at least one message.

Though no signature scheme is completely secure, since an attacker can test all possible signatures for a certain message x, using hash functions that are preimage, second preimage and collision resistant allows it to be protected from the above attacks [32, p. 278]. Typically, digital signature schemes use hash functions that are collision resistant since collision resistance implies second preimage resistance [29].

3.3 Hashes and Bitcoin

SHA-256 is a hash function belonging to the SHA-2 family, designed by the NSA as a successor to the SHA-1 family with numerous security improvements [4]. RIPEMD-160 is another hash function that was, conversely, designed openly by academics. Both are based upon the Merkle–Damgård construction[3]. They are used by Bitcoin to create addresses, complete transactions, and mine blocks in the proof of work process [26]. Crucially, they are collision resistant and therefore suitable for use in a secure signature scheme.

From the previous section we know Bitcoin's addresses are initially generated using ECC. The Bitcoin address is the public key generated from the algorithm in Section 2.3 (a point with x and y coordinates) that is then hashed in the following way [4]. We use the public key H_A from Example 2.19 to illustrate the algorithm:

1. Combine the x and y coordinates and append `04` at the front:

```
040a019541f7807004d734108a70e66544bc9c2cc70f806438582922cf24a15def
9423b0b9531b6e0bb4809823ff06cb905e4cce49b61fa0f23669ea253157db0c
```

[3]The Merkle–Damgård construction (1979) is a method of creating a collision resistant hash function from a compression function with the same property [32, p. 128].

2. Compute the SHA-256 hash of the above key:

   ```
   34062fc313f4967d277d99229f59a99a34957232e2739cb2fb9389a9a559eef6
   ```

3. Compute the RIPEMD-160 hash of the result:

   ```
   61ad25a5ff151c45a20a7f95fec28c09dacd3715
   ```

4. Append with the version number of the network:

   ```
   0061ad25a5ff151c45a20a7f95fec28c09dacd3715
   ```

5. Compute the SHA-256 hash *twice* (known as the double SHA-256 hash):

   ```
   660d774a92f413d843d4455af6077ab9ec0d05ba0c7d412c65d74196e2f49195
   ```

6. Append the end of the result of step 4 with the first 4 bytes of step 5:

   ```
   0061ad25a5ff151c45a20a7f95fec28c09dacd3715660d774a
   ```

7. Convert the result into a string[4]:

   ```
   19uTy5SGUFJW1U3foQ6pTh2im8qLhy1DFo
   ```

This complex system reduces the likelihood of a collision which could result in two users being able to send BTC from a colliding address. Bitcoin designed the algorithm to ensure that it is more profitable to gain BTC legitimately than attempt to create a collision [4]. In fact, it is "more likely that the Earth is destroyed in the next 5 seconds, than that a collision occur in the next millennium" [4].

Once transactions are complete, they are also hashed for convenience of tracking and use in the blockchain. Simply put, transactions are structured messages that are encoded as the following fields in a single string [4, 28]:

Field	Size	Description
Version	32 bit	Data format version
In Count	16-64 bit	Number of transaction inputs
Inputs	-	List of sources of BTC
Out Count	16-64 bit	Number of transaction outputs
Outputs	-	List of destinations for BTC
Lock Time	32 bit	Earliest time the transaction can join the blockchain

Table 3.1: Bitcoin transaction fields

[4]Bitcoin addresses are converted into strings using Base58Check encoding [4]. Here, I used the tool at http://lenschulwitz.com/base58.

The final encoded string is clearly of variable size and can be very large, so it is convenient to hash it to a number of fixed length. Bitcoin computes the double SHA-256 hash of the transaction [28]. For example, the whole message of the first Bitcoin transaction, which has only one input and output, is:

```
01000000010000000000000000000000000000000000000000000000000000000000000000fff
ffff4d04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e636
56c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f72206261
6e6b73ffffffff0100f2052a01000000434104678afdb0fe5548271967f1a67130b7105cd6a82
8e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c
702b6bf11d5fac00000000
```

Which is then hashed twice to give the 256 bit number used for reference:

```
4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
```

The implementation of hashing in the blockchain has been introduced in Section 1.2. We can now explain the process in detail corresponding to the fields in Table 1.2. Each block hash is generated by hashing the concatenation of all block fields twice with SHA-256 in the following order [4]:

- **Version:** A 32 bit number to track upgrades and define block structure.
- **Previous Block:** The 256 bit previous block hash, generated using this algorithm.
- **Transactions:** The 256 bit hash of the root of the Merkle Tree (*discussed below*), a fingerprint of all the transactions in the block.
- **Time:** A 32 bit time stamp of the approximate block creation time in seconds from the Unix Epoch[5].
- **Bits:** The 32 bit representation of the difficulty target in coefficient/exponent format for convenience [1, p. 192].
- **Nonce:** A 32 bit 'counter' required for proof of work, as described in Section 1.2.

For example, the title hash of block 510860:

```
000000000000000000059cc97f89734474ba2116eb61c4caad6d51f80af6ba3e8
```

is the double SHA-256 hash of all the fields in Table 1.2 encoded and ordered in a 640 bit string as described above. For blocks to have a variable number of transactions, as is often the case, and be mined at a constant rate Bitcoin summarises them using a

[5]The Unix Epoch refers to the number of seconds from 1.1.1970 to a specified date and is widely used as an integer representation of a time stamp in computer science [1, p. 114].

Merkle Tree [28, 1, p. 170]. Merkle Trees are formed by hashing pairs of data points in a hierarchical structure until only one remains, the Merkle Root. Clearly, each row must have an even number of elements until the root has been reached, so the final hash of a row is duplicated if this is not the case [4]. Bitcoin uses the double SHA-256 function as the hash here also. As an example, say we have a block with three transactions in their 256 bit hashed form: Tx_1,Tx_2 and Tx_3. Our Merkle Tree is created thus:

$$\begin{tikzcd}
 & & & \text{Root Hash} \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & H_1 \arrow[dl, no head] \arrow[dr, no head] & & & & H_2 \arrow[dl, no head] \arrow[dr, no head] & \\
Tx_1 & & Tx_2 & & Tx_3 & & Tx_3
\end{tikzcd}$$

Since we are starting with an odd number of data points, we must duplicate the final element, Tx_3. Then, H_1 is computed as the hash of the concatenation of Tx_1 and Tx_2, similarly H_2 is the hash of Tx_3 appended with itself. The Merkle Root is the hash of the concatenation of H_1 and H_2.

This method can be efficiently used for very large numbers of transactions, both in calculation and verification. To check whether a single transaction is in a block one needs only to produce some hashes on a *Merkle Path* from that transaction to the root [1, p. 175]. For a tree starting from a large number of data elements, say N, only $\log_2(N)$ hashes must be computed to prove the existence of a certain element. For instance, Block 510860 summarised in Table 1.2 has 2145 transactions represented by the single Merkle Root.

Chapter 4

Matlab Implementation

Alongside this report, we have seen worked examples of calculations and algorithms. These are taken from Matlab scripts created to simulate ECC.

We start with a function that carries out point addition on modular elliptic curves. This function is called and repeated to compute point multiplication, using the efficient double and add algorithm, for use in ECC algorithms. In Section 4.1 we look at these functions and how we find the suitable parameters (p, a, b, G, n, h) required for cryptographic operations.

Section 4.2 explores the functions that generate private and public key pairs, create digital signatures, and verify those digital signatures. They mirror the algorithms described in Sections 2.3 and 2.4.

To simulate Bitcoin's use of ECC as closely as possible, we must implement large integers. Unfortunately, Matlab's standard packages and storage limitations are unsuitable for such numbers. The freely available toolbox *Variable Precision Integer Arithmetic*[1] (VPI) allows us to input parameters that are actually used in modern cryptography. Section 4.3 outlines these modified VPI functions.

4.1 Group Operations

In Sections 2.2.2 and 2.2.3 we have defined the binary operation point addition for elliptic curves. The function `PAdd` computes point addition for modular curves in the form stated in (2.12). The input arguments a, b and p define the curve and P and Q represent the points in vector form. It implements (2.13) using nested for-loops to determine cases for $\lambda = $ `sl`. It then calculates $S = (x_3, y_3)$ from (2.14) and (2.15) for the solution $P + Q = R = (x_3, -y_3 \mod p)$. The function is as below:

[1]A Matlab toolbox by John D'Errico available at https://uk.mathworks.com/matlabcentral/fileexchange/22725.

```
function R=PAdd(P,Q,a,b,p)
%Point addition for all cases
if isscalar(P)
    if isscalar(Q) %case P=Q=0
        R=inf;
    else %case P=0
        R=Q;
    end
elseif isscalar(Q) %case Q=0
    R=P;
else %case P,Q not 0
    x1 = P(1); y1 = P(2); x2 = Q(1); y2 = Q(2); %assign coordinates
    if mod( x1^3 + a*x1 + b , p) ~= mod( y1 ^ 2 , p) %checks P in E
        error('P not in E')
    elseif mod( x2^3 + a*x2 + b , p) ~= mod( y2 ^ 2, p) %checks Q "
        error('Q not in E')
    end
    if and(x1 == x2, y1 == mod( -y2 , p )) % case P=-Q
         R=inf;
         return
    end
    if x1 == x2 %case x1=x2 => y1=y2
            [~,U,~] = gcd( (2*y1) , p);
            Y = mod( U , p ); %inverse of 2y1 mod p
            sl = mod(  ( 3 * x1 ^ 2 + a) * Y  , p);
    else %x1 ~= x2
        [~,U,~] = gcd( ( x2 - x1 ) , p);
        X = mod(U, p); %inverse of x2-x1 mod p
        sl = mod( ( y2 - y1 ) * X  , p);
    end
        %finds P + Q = R
        x3 = mod(  ( sl ^ 2 - x1 - x2 )  , p);
        y3 = mod(  sl * ( x3 - x1 ) + y1  , p);
        R = [x3 , mod( -y3, p)];
end
```

Multiplying points by a scalar has also been described in Section 2.2.3. The efficient double and add algorithm (*Algorithm 1*) for point multiplication is implemented in `PMulD`. This function uses the same input arguments as `PAdd` but without Q and with the integer scalar coefficient m. As described in Algorithm 1, the scalar is converted to binary before calculation. The script converts m to base 2 binary as k then computes $k \cdot P$:

```
function kP=PMulD(P,m,a,b,p)
%Point multiplication - Double and Add Algorithm
kP = inf; %Starts at 0
```

```
Q = P; %Starts at 1P
k= de2bi ( m , [] , 2) ; %converts m to binary
for ii= 1:length(k)
    if k(ii)==1
        kP=PAdd(kP,Q,a,b,p); %for each 1 in k, add Q to kP
    end
     Q=PAdd(Q,Q,a,b,p); %for each digit in k, double Q
end
```

In all the Matlab scripts created, `inf` is used as the identity element O since it can easily be checked against a two component point with the `isscalar` command. We initialise the solution, kP, as O and the point Q as $1P$. The scalar m is converted to binary as k and the algorithm is computed for each digit of k. Simply put, the point Q is doubled at each iteration and added to kP if the digit of k is 1 for the current run. In the above Algorithm 1, we only use one iterative point Q. Here two are implemented to aid in troubleshooting.

We describe an algorithm to find a suitable generator G in Section 2.2.4. This is implemented in `Gnh` with the inputs a, b and p to define the curve E. The function outputs the full sextuple of parameters (p, a, b, G, n, h) we require to simulate ECC:

```
function [G, n, h, a, b, p]=Gnh(a, b, p)
E=EllC(a, b, p); %generates the set E
N=length(E)+1; %number of points in E (including O=inf)
L=factor(N); %prime factors of N
l=randi(length(L),1); %random number to choose a prime factor
n=L(l);
h=N/n; %cofactor
for ii=1:N
    m=randi(N-1); %random number to choose a point, P (not O)
    P=E(m,:);
    G=PMulD(P, h, a, b, p); %G=hP
    if G ~= inf %G cannot be O
        return
    end
end
```

The function `EllC` creates a matrix of points in E by simply checking for solutions to the Weierstrass normal form (2.12) for each integer[2] between 0 and $p-1$. We then assign the size of this matrix as N, adding one to the `length` to include the identity element, and find its prime factors that we store in the vector L. The integer n is used as the order of the subgroup G and is found as a random element of L. We use n to calculate the cofactor h, then we repeatedly find random points P until a suitable (non-O) generator G is discovered.

[2]Clearly, this method is inefficient for large values used in real applications, so we alternatively use VPI as discussed in Section 4.3.

4.2 ECC Simulation

We are now able to find all the required parameters to create cryptographic key pairs. The function `keyprodall` outputs d_A, H_A, d_B, H_B, and the public sextuple (p, a, b, G, n, h) from the curve parameters. In reality, (p, a, b, G, n, h) contains the input parameters. For simplicity, this code combines finding, from any curve, a suitable G, n, and h with finding keys. There is an additional commented line to check the secret S, as described in Section 2.3. This functions calls those introduced above:

```
function [da,Ha,db,Hb,G,n,h,a,b,p] = keyprodall(a, b, p)
%Outputs private and public keys from E, outputs all parameters
[G, n, h, a, b, p]=Gnh(a, b, p);
da=randi(n,1); %choose random private key (A)
db=randi(n,1) ;%choose random private key (B)
Ha=PMulD(G,da,a,b,p); %computes public key (A)
Hb=PMulD(G,db,a,b,p); %computes public key (B)
%S=PMulD(Ha,db,a,b,p); %'secret' S - for checking only
```

When we have the full set of public parameters (p, a, b, G, n, h), for instance when simulating real parameters, the function `btckeyprodall` can be used to generate keys:

```
function [da,Ha,db,Hb,G,n,h,a,b,p] = btckeyprodall(G,n,h,a,b,p)
%Finds private and public keys from sextuple, outputs all parameters
da=randi(n,1); %choose random private key (A)
db=randi(n,1); %choose random private key (B)
Ha=PMulD(G,da,a,b,p); %computes public key (A)
Hb=PMulD(G,db,a,b,p); %computes public key (B)
%S=PMulD(Ha,db,a,b,p); %'secret' S - for checking only
```

Ensuring that key generation is completed so that all the outputs are saved to the workspace, we can now create and verify a digital signature. We complete this with the function `sigc` from the perspective of person A, using their private key d_A, the public curve parameters, and their message z. As discussed in Section 3.3, z is actually the hashed payment instruction. In this version, the limitations of Matlab mean that we cannot enter a realistically large integer, so we use any appropriate natural number. The function, mirroring the algorithm in Section 2.4, outputs a 'signed message' (the signature (r, s) and the message z). It is given by:

```
function [r,s,z]=sigc(da,G,n,a,b,p,z)
%Creates signature (r,s) - use keyprodall.m to create parameters
r=0;
s=0;
while r==0
    while s==0
        k=randi(n-1); %secret k
        P=PMulD(G,k,a,b,p); %P=kG
        r=mod(P(1), n); %computes r (cannot be 0)
        [~,U,~] = gcd(k, n); %computes the inverse of kmodn
        kinv = mod(U,n);%"
        s=mod(kinv*(z+r*da),n); %computes s (cannot be 0)
    end
end
```

The signature (r, s) can be seen and verified by anyone with public information. The function `sigv` confirms the validity of the signature from the public curve parameters, person A's public key H_A, and the signed message itself. Clearly, all parameters here must be the same as were used to create the signature. The verification is as follows:

```
function sigv(r,s,z,Ha,G,n,a,b,p)
%Verifies signature (r,s)- use keyprodall.m and sigc.m
[~,U,~] = gcd(s, n); %computes the inverse of smodn
sinv = mod(U,n); %"
u1=mod(sinv*z,n);
u2=mod(sinv*r,n);
P1=PMulD(G,u1,a,b,p); %u1G
P2=PMulD(Ha,u2,a,b,p); %u2Ha
P=PAdd(P1,P2,a,b,p); %P = u1G + u2Ha
if r == mod(P(1), n)
    disp('Signature is valid') %verifies r=xmodn
else
    disp('Signature not valid')
end
```

Following the algorithm at the end of Section 2.4, `sigv` finds the x coordinate of the point P (identical to that of `sigc`) and checks it is equal to r. Note that this function takes public information as input arguments and has no outputs; only a confirmation message.

In a few cases above, we have seen examples of elliptic curves over small finite fields for ease of understanding. Group operations and cryptographic algorithms are comprehensive in this way, but can be problematic to carry out. One problem I faced was finding a generator G for certain small curves; in some cases the function `Gnh` could not find a non-O G. This turned out to be because all the subgroups of some curves have order n that is a divisor of the cofactor, h. Therefore, $G = hP$ is always O.

To check if curves have this property, the function `subgsize` outputs a matrix with each point on the curve and the subgroup order n corresponding to that point. The first column is the x coordinate, the second is the y coordinate, and the third is the subgroup order. The function is:

```
function ES=subgsize(a,b,p)
%outputs all points in E and the size of the subgroup generated by
    each point
```

```
E=EllC(a,b,p);
N=length(E)+1
ES=[E(:,1),E(:,2),zeros(N-1,1)];
for ii=1:N-1
    for jj=1:N
        Q=PMulD(E(ii,:),jj,a,b,p);
        if isscalar(Q)
            ES(ii,3)=jj;
           break
        end
    end
end
```

For example, using the curve $y^2 \equiv x^3 - 14x + 9 \mod 31$, we have the following output:

```
N =

    27

ans =

     0     3     9
     0    28     9
     2    12     9
     2    19     9
     3     5     9
     3    26     9
     5     8     9
     5    23     9
    10     1     9
    10    30     9
    11    15     3
    11    16     3
    12     9     3
    12    22     3
    13     3     3
    13    28     3
    18     3     9
    18    28     9
    20    14     9
    20    17     9
    25     1     9
    25    30     9
    26     4     3
    26    27     3
    27     1     9
    27    30     9
```

For instance, the point $(11, 15)$ is a generator of a subgroup of order 3. Since $N = 27$, the only prime factor is 3. For all iterations of `Gnh`, we have $n = 3$ and $h = 9$, therefore $G = O$ and is unsuitable. Running `subgsize` for a small curve first allows us to check that there exist appropriate subgroups.

4.3 VPI Implementation

To accurately simulate ECC as it is used by Bitcoin, we require a way to store and calculate very large integers. Matlab, by default, stores numbers in `double` format, which has a size limit below Bitcoin's actual parameters. Additionally, the built in functions that we use for modular arithmetic either have an inaccurate output or no output for such integers.

Instead, we implement the toolbox VPI, as mentioned above, that not only includes a new data type for large integers (`vpi`) but the functions that we require. Our group operation and cryptographic functions described above, with some modifications, can be used for real Bitcoin curve parameters. We have seen examples above of keys and signatures in hexadecimal format from the adaptations.

Our function `PAdd` only needs a few changes to take `vpi` values. The toolbox contains a modular inverse function `minv` that we use instead of the original `gcd` method. As long as all inputs, even if a or b are small, are stored as `vpi` objects the below function `PAddVPI` computes point addition:

```
function R=PAddVPI(P,Q,a,b,p)
%Point addition for all cases
if isscalar(P)
    if isscalar(Q) %case P=Q=O
        R=inf;
    else %case P=O
        R=Q;
    end
elseif isscalar(Q) %case Q=O
    R=P;
else
    x1 = P(1); y1 = P(2); x2 = Q(1); y2 = Q(2);%assigns coordinates
    if mod( x1^3 + a*x1 + b , p) ~= mod( y1 ^ 2, p) %checks P in E
        error('P not in E')
    elseif mod( x2^3 + a*x2 + b , p) ~= mod( y2 ^ 2, p) %checks Q "
        error('Q not in E')
    end
    if and(x1 == x2, y1==mod(-y2,p)) %P=-Q
         R=inf;
         return
    end
    if x1 == x2 %case x1=x2 => y1=y2
            Y = minv( 2 * y1  , p ); %inverse of 2y1 mod p
            sl = mod( ( 3 * x1 ^ 2 + a)*Y , p);
    else %x1 ~= x2
        X = minv( x2 - x1  , p); %inverse of x2-x1 mod p
        sl = mod( ( y2-y1)*X   , p);
    end
        %finds P + Q = R
        x3 = mod(  ( sl ^ 2 - x1 - x2 )  , p);
        y3 = mod(  sl * ( x3 - x1 ) + y1   , p);
        R = [x3 , mod( -y3, p)];
end
```

Similarly for `PMulD`, we only need some minor changes. To convert the multiplier m to binary we use the `vpi2bin` function from the VPI toolbox and reverse the digits with `fliplr`, since the double and add algorithm (*Algorithm 1*) must be computed in decreasing powers of two [32, p. 258]. The digits of a long `vpi` number are also considered `vpi` numbers, so we must check each digit of k against `vpi(1)`. The modified function `PMulDVPI` is as below:

```
function kP=PMulDVPI(P,m,a,b,p)
%Point multiplication - Double and Add Algorithm
kP = inf; %Starts at 0
Q = P; %Starts at 1P
k=fliplr(vpi2bin(m)); %converts m to binary (in 'reverse' order)
for ii= 1:length(k)
    if k(ii)== vpi(1)
        kP=PAddVPI(kP,Q,a,b,p); %for each 1 in k, add Q to kP
    end
     Q=PAddVPI(Q,Q,a,b,p); %for each digit in k, double Q
end
```

More modification must be done to convert the function `Gnh`. We need to factor the group order N without simply counting points. While Schoof's algorithm, mentioned in Section 2.2.4, is efficient, it is also time consuming and complex to implement. We instead include N as an input and use elliptic curves, actually used in cryptography, that have public parameters [6]. Though this now means it is unnecessary to use `Gnh`, because we are using the public (p, a, b, G, n, h), for completeness we have `GnhVPI`:

```
function [G, n, h, a, b, p]=GnhVPI(a, b, p, N)
L=factor(N); %prime factors of N
l=randi(length(L), 1); %random number to choose a prime factor
n=L(l);
h=N/n; %cofactor
y=[]; %assigns empty y
G= inf; %assigns G=0
m=randint(p); %random number for x coordinate
while G == inf
    while isempty(y)==1
        m=m+1; %changes x for each iteration we have no modular root
            /suitable G
        y= modroot(m^3+a*m+b,p); %calculates y coordinate
    end
    P=[m,y]; %random point P
    G=PMulDVPI(P, h, a, b, p); %G=hP
end
```

To choose a random point P, we first choose a random x coordinate. With the VPI `modroot` function we are able to check if this x is on our elliptic curve. If it is, we have a y coordinate and therefore a random P. If either x is not on the curve or we find $hP = G = O$, then we add 1 to the x coordinate and repeat the calculations.

To create keys, we simply call the point multiplication and subgroup functions. The function `keyprodallVPI` is nearly identical to the key generation function above, apart from the recommended use of `randint` to choose d:

```
function [da,Ha,db,Hb,G,n,h,a,b,p] = keyprodallVPI(a, b, p, N)
%Outputs private and public keys from E, outputs all parameters
[G, n, h, a, b, p]=GnhVPI(a, b, p, N);
da=randint(n)+1; %choose random private key (A) (add 1 so d =/= 0)
db=randint(n)+1; %choose random private key (B)
Ha=PMulDVPI(G,da,a,b,p); %computes public key (A)
Hb=PMulDVPI(G,db,a,b,p); %computes public key (B)
```

For examples in which we have a full set of public parameters, such as the Bitcoin curve, we use `btckeyprodallVPI`. This function does not include `Gnh` to find G:

```
function [da,Ha,db,Hb,G,n,h,a,b,p] = btckeyprodallVPI(G,n,h,a,b,p)
%Outputs private and public keys from sextuple, outputs all
    parameters
da=randint(n)+1; %choose random private key (A) (add 1 so d =/= 0)
db=randint(n)+1; %choose random private key (B)
Ha=PMulDVPI(G,da,a,b,p); %computes public key (A)
Hb=PMulDVPI(G,db,a,b,p); %computes public key (B)
```

The VPI functions to create and verify signatures are nearly identical to that of the previous section. Creating a signature with `sigcVPI` differs in the use of the `randint` and `minv` functions. Verifying that signature with `sigvVPI` also uses `minv` instead of `gcd`. Now we are able to input a realistically large value of z:

```
function [r,s,z]=sigcVPI(da,G,n,z,a,b,p)
%Creates signature (r,s) - use keyprodallVPI.m to create parameters
r=vpi(0);
s=vpi(0);
while r==vpi(0)
     while s==vpi(0)
        k=randint(n-1)+1; %secret k (+ 1 to negate 0)
        P=PMulDVPI(G,k,a,b,p); %P=kG
        r=mod(P(1), n); %r=xmodn (cannot be 0)
        kinv = minv(k,n);%computes the inverse of kmodn
        s=mod(kinv*(z+r*da), n); %computes s (cannot be 0)
    end
end
```

```
function sigvVPI(r,s,z,Ha,G,n,a,b,p)
%Verifies signature (r,s)- use keyprodallVPI.m and sigcVPI.m
sinv = minv(s,n); %computes the inverse of smodn
u1=mod(sinv*z,n);
u2=mod(sinv*r,n);
P1=PMulDVPI(G,u1,a,b,p); %u1G (a point)
```

```
P2=PMulDVPI(Ha,u2,a,b,p); %u2Ha (")
P=PAddVPI(P1,P2,a,b,p); %P = u1G + u2Ha
if r == mod(P(1), n)
    disp('Signature is valid') %verifies r=xmodn
else
    disp('Signature not valid')
end
```

The curve parameters we use with the VPI functions are found in *Standards for Efficient Cryptography 2 (SEC 2)* [6], including Bitcoin's chosen curve, `secp256k1`. This document outlines recommended parameters for secure curves to be used in cryptography, categorised by bit size. We convert the integers found in this document from hexadecimal to `vpi` with the function `hex2vpi`[3] and predetermine MAT-files for simulation.

4.4 Results

In running the initial functions created for small examples, we find a solution almost immediately. However, when using real world parameters in the VPI modified functions, it takes a few minutes at each step. To study this delay, I have taken the following run times in Matlab to find keys, create a signature from those keys, and verify that signature from `secp256k1` parameters.

Function	Run Times							Mean
`keyprodallVPI`	06:25	06:31	07:06	06:34	06:55	06:20	06:34	06:38
`sigcVPI`	02:57	03:12	03:10	03:04	03:04	02:58	03:06	03:04
`sigvVPI`	06:06	06:18	06:17	06:25	06:06	06:14	06:13	06:14
`btckeyprodallVPI`	06:12	06:13	06:15	06:08	06:00	06:04	05:59	06:07
`sigcVPI`	03:09	03:07	03:03	03:03	03:02	03:00	03:03	03:04
`sigvVPI`	06:16	06:11	06:02	06:14	06:10	06:09	06:00	06:09

Table 4.1: Function run times for `secp256k1` parameters (minutes)

As expected, `btckeyprodallVPI` is faster than `keyprodallVPI` because it does not have to find G, n, and h before establishing keys. We can ascertain from the average times that each point multiplication calculation accounts for approximately three minutes of run time; both the key generation and signature verification functions require two multiplications while `sigcVPI` requires only one.

[3]A function by Paulo Fonte to complement the VPI toolbox, also found at https://uk.mathworks.com/matlabcentral/fileexchange/22725.

Chapter 5

Conclusions and Recommendations

5.1 Conclusions

We have explored in depth different areas of mathematics that underly Bitcoin's system. Relatively recent advancements in number theory have been utilised by Nakamoto, rather ingeniously, to create a completely new type of currency.

Perhaps the most crucial use of mathematics here is public key cryptography. Bitcoin keeps no secrets regarding its structure, every single coin and line of code can be checked by anyone, and yet it relies on mathematics formulated for the utmost secrecy. Initially, asymmetric key cryptosystems were designed for secure message exchange, but have been developed for use in signature schemes. These schemes solved one of the problems Nakamoto faced in creating his cash-like currency; proof of ownership without identity. Though conventional online banking is cryptographically secure, one requires proof of identity and permission from the bank to access one's funds. Bitcoin uses one free, pseudonymous, universally available key pair to complete all the necessary security requirements.

Elliptic curve cryptography was not in widespread use at the time of Bitcoin's introduction (the National Institute of Standards and Technology (NIST) only approved use in 2000 [32, p. 291]). However, it allows for smaller key size and faster implementation at the same level of security as RSA [15]. This successful implementation of ECC has arguably popularised the system; for instance the curve `secp256k1` was "barely used" before Bitcoin despite it being very efficient computationally [4].

We have described many interesting properties of elliptic curves. By defining our point addition operation, we have seen how such curves form an abelian group, over both $\mathbb{R}$ and $\mathbb{Z}_p$, and how to identify useful cyclic subgroups. The intractability of k from $Q = kP$ is central to our key exchange and digital signature algorithms. These algorithms seem complex, since they required considerable introduction, but have been implemented in comprehensive Matlab functions.

Interestingly, operations on elliptic curves had previously only been researched by pure mathematicians for academic interest. It was not thought that there would be any practical application. Even when ECC was introduced very few people thought it could be actually used since point arithmetic was then inefficient [15]. Koblitz and Miller's novel application[1] of elliptic curves attracted interest and, subsequently, efficient algorithms were discovered.

Hash functions form the other central mathematical topic used by Bitcoin. We have described how they are used widely; in compressing addresses and transactions, signing algorithms, and constructing the blockchain. The important 'digital fingerprint' property is easy to comprehend but has been applied in complex, new ways. Nakamoto carefully recognised possible issues and used hash functions to design a system that has collision resistant addresses, to avoid fraud, and transactions of a consistent bit size, to maintain efficiency. The most crucial application is arguably the proof of work system that allows transactions, or any data, to be recorded publicly with assurance of integrity.

Nakamoto managed to create the blockchain, a truly unique and groundbreaking system, on the basis of hash functions. It is generally considered that the lasting impact of Bitcoin will not be the decentralised currency itself, but the blockchain. Many influential bodies have published reports lauding the technology, such as the World Economic Forum and UK Government [33]. In fact, the UK Government Office for Science published an 88 page report reviewing how Bitcoin's "distributed ledger technology can revolutionise services, both in government and the private sector" [13]. Since the blockchain can verify and store anything that can be expressed in code, it has applications far beyond cryptocurrency. In particular, businesses that share data would find value in replacing trust in institutions or people with mathematical proof.

5.2 Recommendations

This report has only touched on various topics in number theory, algebra, cryptography, and computing that underly Bitcoin's system. There is much more depth to explore in these areas.

In Section 4.4, we briefly outline the results of the ECC Matlab simulation. Despite Matlab's limitations, we have functions that compute all the algorithms required for digital signatures, including the possibility of using real parameters. However, they are unrefined and in some cases slow to run. We have yet to explore the efficiency of the individual functions. Matlab provides a wealth of methods for algebraic computation.

[1]Around the time efficient point arithmetic was gaining interest (1995), it was proven that all rational elliptic curves are also modular (*The Taniyama-Shimura conjecture*). By proving this, Andrew Wiles also proved Fermat's Last Theorem [34].

It is possible that our functions could be optimised by changing such methods, for example investigating our use of `gcd` to find modular inverses. Additionally, in `PMulD` we implement the double and add algorithm (*Algorithm 1*) using two placeholders. This was to ensure correctness; in particular check that the number of 1s in binary `k` matched with the number of changes to `kP`. Most likely, this is an inefficient way (especially considering the large integer adaptation) to code the algorithm.

Table 4.1 demonstrates that it takes around three minutes of run time per point multiplication for realistically large integers. Considering that wallet software can produce Bitcoin key pairs in a matter of seconds, we can safely assume our VPI functions would benefit from optimisation [4]. The VPI toolbox is an unofficial open source package, so there is no available analysis on how suitable certain adapted functions actually are. For example, we choose a random private key with `randint`. This is not recommended by Matlab but is essential for `vpi` objects. Whether `randint` is fast enough for our large parameters, or even random enough for cryptographic use, would be a useful investigation.

The biggest drawback in the Matlab implementation is the lack of an efficient point counting algorithm. To find the parameters G, n, and h, we require calculations with the number of points on the curve; for small numbers, we use `EllC` and count each solution to (2.12) (known as the *naive algorithm* [15]). For large numbers, this is an infeasible operation.

As mentioned above, the introduction of theoretical ECC in 1985 led to advancements in point arithmetic that allowed it to become practical. Also in that year, the first efficient point counting algorithm was formulated by René Schoof [8]. Prior to this the most useful bound for the order of E was Hasse's Theorem (*Theorem 2.14*), found much earlier in 1934.

The difficultly in describing and implementing Schoof's algorithm lies in its complexity and length. His idea was to find the order of E when taken modulo all primes under a certain bound, then recover the true value using the Chinese Remainder Theorem (CRT). Though the CRT is not outside the scope of this report, Schoof finds the different orders modulo a prime with some complex methods [15]. Primarily, the *Frobenius endomorphism* which maps the points (x, y) on an elliptic curve E over $\mathbb{Z}_p$ to the set (x^p, y^p). With some calculations, we can derive:

$$(x^{p^2}, y^{p^2}) + p(x, y) = t(x^p, y^p), \tag{5.1}$$

where p and t are coefficients of point multiplication. We reduce our curve E to only include points P that satisfy $lP = O$ for some prime l. In other words, we only include generators of order l.

For this new set, $\bar{E}$, if $t(x^p, y^p) = \bar{t}(x^p, y^p)$ where $\bar{t} = t \mod l$ and $\bar{p} = p \mod l$, we have (5.1) reduced to [15]:

$$(x^{p^2}, y^{p^2}) + \bar{p}(x, y) = \bar{t}(x^p, y^p). \tag{5.2}$$

To eventually find the order of E, we must find $\bar{t}$ for different values of l. To do this, we also require *division polynomials*, that are defined by Schoof in his 1985 paper [30].

Definition 5.1. Division polynomials $\Psi(x, y)_n \in \mathbb{Z}[x, y, a, b]$ are given by:

$$\begin{aligned}
&\Psi_{-1} = -1,\\
&\Psi_0 = 0,\\
&\Psi_1 = 1,\\
&\Psi_2 = 2y,\\
&\Psi_3 = 3x^4 + 6ax^2 + 12bx - a^2,\\
&\Psi_4 = 4y(x^6 + 5xa^4 + 20bx^3 - 5a^2x^2 - 4abx - 8b^2 - a^3).
\end{aligned}$$

We recursively find $\Psi_{n\geq 5}$ thus:

$$\begin{aligned}
&\Psi_{2n} = \Psi_n(\Psi_{n+2}\Psi_{n-1}^2 - \Psi_{n-2}\Psi_{n+1}^2)/2y,\ n \geq 3,\ n \in \mathbb{Z},\\
&\Psi_{2n+1} = \Psi_{n+2}\Psi_n^3 - \Psi_{n+1}^3\Psi_{n-1},\ n \geq 2,\ n \in \mathbb{Z}.
\end{aligned}$$

Division polynomials at points $(x, y) \in E$ have useful properties that relate scalar point multiples $k(x, y)$ to $\Psi(x, y)_k$. Schoof uses these to find values of $\bar{t}$, then recovers t using the CRT, and finally calculates the order of E.

Deriving properties of division polynomials and the Frobenius endomorphism requires complex algebra. This is challenging to explain and even harder to implement in Matlab. Even after outlining the above background in detail, a more recent paper (Kamarulhaili and Jie, 2012 [15]) requires 20 steps to find $|E|$. In Matlab, these steps would be implemented in many if-statements and for-loops which are prone to error and slow computation (particularly as we only need this algorithm for very large integers). Since division polynomials Ψ_n grow exponentially[2] with n, it is possible that even VPI would be unable to cope. For further reading, the aforementioned 2012 paper [15] and Schoof's original paper [30] are available in the footnotes. A well-known C++ adaptation[3] of Schoof's original algorithm is also below.

[2]Due to this, Schoof's original algorithm was found to be unsuitable for cryptographic use. Many fast adaptations have been found since (for example, SEA due to Elkies and Atkin) but require yet more complex methods [15]. A polynomial time algorithm (Adleman and Huang) was even described by ECC co-founder Koblitz as "completely impractical, as well as very complicated" [18, p. 190].

[3]Mike Scott (1999), found at https://github.com/miracl/MIRACL/blob/master/source/curve/schoof.cpp.

We have described Bitcoin's choice of elliptic curve, `secp256k1`, and its basic useful properties. Much of the work regarding this curve has been implementing it in Matlab to provide an accurate simulation. Though Nakamoto never stated why this 'unpopular' curve was chosen before vanishing, further study of `secp256k1` has great value.

Elliptic curves used in cryptography are generally chosen from one of two methods [6]. The parameters (p, a, b, G, n, h) are either of a *verifiably random* curve or associated with what is known as a Koblitz curve. In both cases, the prime base p is of a special form that allows faster modular arithmetic than primes of the same size [23]. Verifiably random curves use the hash function SHA-1 (a precursor to SHA-256) to create a *seed* from which a and b of the Weierstrass normal form are generated. This means that the curve does not have cryptographic vulnerabilities known only to the author[4] [8]. Koblitz curves are designed predictably to have efficient computations. In general they have $a = 0$ and b small positive which, along with the choice of p, allows for them to have the following properties [19]:

- They are nonsupersingular,
- The order of the curve has a large prime factor,
- Point doubling is very efficient,
- They are easy to discover.

Nonsupersingular curves are safe from a certain attack (The *Menezes- Okamoto- Vanstone reduction*, 1993) that reduces the discrete logarithm problem. Additionally, the existence of a large prime factor of $|E|$ protects Koblitz curves from *baby-step/giant-step* and *Pollard-Rho* algorithms to compute the discrete log [8, 19].

Bitcoin's curve `secp256k1` is a Koblitz curve with prime order. So, using notation from Section 2.2.4, our subgroup order n equals the curve order N. Demonstrating the efficiency and security of this curve would be a lengthy but useful extension of this report. We could also compare such Koblitz curves to verifiably random curves. Further reading of these topics can be found in the SEC 2 document [6], a paper on Koblitz curves and Bitcoin (Bjoernsen, 2015 [5]), and Koblitz's original 1991 paper on secure curves [19]. An interesting project that outlines protocol for choosing secure curves can be found at [2].

Theoretically, any 'curve' that is an abelian group under a point operation can be used for cryptography. In this report, we have only seen elliptic curves and, briefly, some other cryptosystems. Further work to supplement this could regard one of the

[4]At the time of Nakamoto's paper, popular curves in cryptography were created by the NIST [4]. They have since been deemed untrustworthy due to 'backdoors' purposefully implemented for the NSA, information that was famously leaked by Edward Snowden [5].

many alternative ‘curves’ in cryptography; hyperelliptic, Edwards, Montgomery, and Hessian are some examples [23]. We introduce the first two, starting with hyperelliptic curves. We define C thus [21]:

Definition 5.2. A hyperelliptic curve C of genus g over finite field $\mathbb{Z}_p$ is the set of solutions to:

$$y^2 + h(x)y = f(x) \in \mathbb{Z}_p[x, y], \tag{5.3}$$

where f is monic and has degree $2g + 1$, h has degree no greater than g. Both f and h have coefficients in $\mathbb{Z}_p$.

As before, the curve C cannot be singular for cryptographic purposes. To find an abelian group, we use what is known as the *jacobian* of the curve, not the curve itself [21]. We cannot define a geometric point operation like that of elliptic curves (which are actually hyperelliptic curves of genus 1). Deriving the jacobian is a rather complex process that requires group theory beyond what has been laid out in this report. It is defined formally by Koblitz in his paper introducing the idea of hyperelliptic curve cryptography [20] and, perhaps more comprehensively, in a 2004 article by Jacobson, Menezes, and Stein [14]. Though it clearly harder to understand and implement, hyperelliptic cryptography has many advantages over ECC, including fast arithmetic [9].

Like Montgomery and Hessian, an Edwards curve is a transformation of an elliptic curve. All elliptic curves of characteristic $p > 2$ can be transformed to an Edwards curve, which is particularly useful in cryptography due to the speed at which point operations are performed [3].

Definition 5.3. An Edwards curve is an elliptic curve in the form:

$$x^2 + y^2 = c^2(1 + x^2y^2). \tag{5.4}$$

The curve in this form remains an abelian group. Point calculations on an Edwards curve, as opposed to the Weierstrass elliptic curve (2.2), are simple and symmetric [3]:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{c(1 + x_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - x_1x_2y_1y_2)} \right). \tag{5.5}$$

If we compare the above to our point arithmetic equations ((2.8), (2.9), and (2.10)) it is clear that the Edwards form requires far fewer operations. Furthermore, our identity element O cannot be expressed numerically but for an Edwards curve it is simply $(0, c)$ [3]. If one replaces x_2 by 0 and y_2 by c in (5.5), the derivation becomes clear.

A relatively recent development, Edwards curves have great cryptographic potential and are widely accessible. For further reading, Bernstein and Lange have completed extensive research on Edwards curves [3] from his original 2007 paper [11].

Bibliography

[1] A. M. Antonopoulos. *Mastering Bitcoin.* O'Reilly Media, 2015.

[2] D. J. Bernstein and T. Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography.* URL: https://safecurves.cr.yp.to (visited on Mar. 15, 2018).

[3] T. Bernstein Daniel J.and Lange. "Faster Addition and Doubling on Elliptic Curves". In: *Advances in Cryptology – ASIACRYPT 2007.* Ed. by K. Kurosawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[4] *Bitcoin Wiki.* 2017. URL: https://en.bitcoin.it/wiki/ (visited on Nov. 2, 2017).

[5] K. Bjoernsen. *Koblitz Curves and its practical uses in Bitcoin security.* 2015. URL: https://www.semanticscholar.org/paper/Koblitz-Curves-and-its-practical-uses-in-Bitcoin-Bjoernsen/ (visited on Mar. 15, 2018).

[6] D. R. L. Brown. *Standards for Efficient Cryptography 2 (SEC 2).* 2010. URL: www.secg.org (visited on Nov. 15, 2017).

[7] J. B. Buchmann. *Introduction to Cryptography.* New York: Springer-Verlag, 2004.

[8] A. Corbellini. *Elliptic Curve Cryptography: a Gentle Introduction.* 2015. URL: http://andrea.corbellini.name/ (visited on Oct. 20, 2017).

[9] C. Costello. "The state-of-the-art in hyperelliptic curve cryptography". In: *Workshop on Curves and Applications.* Calgary, 2013.

[10] W. Diffie and M. E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22 (6 1976).

[11] H. Edwards. "A normal form for elliptic curves". In: *Bulletin of the American Mathematical Society* 44.3 (2007).

[12] S. Friedl. "An Elementary Proof of the Group Law for Elliptic Curves". In: *Groups Complexity Cryptology* 9 (2 2004).

[13] M. Hancock and E. Vaizey. *Distributed ledger technology: beyond block chain.* Tech. rep. UK Government Office for Science, Jan. 2016. URL: https://www.gov.uk/government/news/distributed-ledger-technology-beyond-block-chain (visited on Feb. 15, 2018).

[14] M. J. Jacobson Jr, A. J. Menezes, and A. Stein. “Hyperelliptic curves and cryptography”. In: *High primes and misdemeanours: lectures in honour of the 60th birthday of Hugh Cowie Williams* 41 (2004).

[15] H. Kamarulhaili and L. K. Jie. “Cryptography and Security in Computing”. In: InTech, 2012. Chap. Elliptic Curve Cryptography and Point Counting Algorithms.

[16] G. O. Karame and E. Androulaki. *Bitcoin and Blockchain Security.* Artech House, 2016.

[17] G. O. Karame, E. Androulaki, and S. Capkun. “Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin”. In: *Conference on Computer and Communication Security.* 2012.

[18] N. Koblitz. *A Course in Number Theory and Cryptography, 2nd Ed.* New York: Springer-Verlag, 1994.

[19] N. Koblitz. “CM-curves with good cryptographic properties”. In: *Annual International Cryptology Conference*. Springer. 1991.

[20] N. Koblitz. “Hyperelliptic cryptosystems”. In: *Journal of cryptology* 1.3 (1989).

[21] T. Lange. “Koblitz curve cryptosystems”. In: *Finite Fields and Their Applications* 11.2 (2005).

[22] A. Lewis. *A Gentle Introduction to Bitcoin.* 2015. URL: https://bitsonblocks.net/ (visited on Oct. 20, 2017).

[23] A. Miele and A. K. Lenstra. “Efficient Ephemeral Elliptic Curve Cryptographic Keys”. In: *Information Security Conference (ISC 2015).* Vol. 9290. Springer-Verlag Berlin. 2015.

[24] R. A. Mollin. *RSA and Public-Key Cryptography.* Chapman and Hall/CRC, 2003.

[25] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on Nov. 1, 2017).

[26] N.Mistry. *An Introduction to Bitcoin, Elliptic Curves and the Mathematics of ECDSA*. 2015. URL: https://www.slideshare.net/NikeshMistry1/introduction-to-bitcoin-and-ecdsa (visited on Nov. 1, 2017).

[27] *Number of Blockchain wallet users globally 2015-2017.* 2017. URL: http://www.statista.com (visited on Jan. 12, 2018).

[28] P. Pares. *An Introduction to the Bitcoin System.* 2017. URL: https://pascalpares.gitbooks.io/implementation-of-the-bitcoin-system/content/ (visited on Jan. 25, 2018).

[29] P. Rogaway and T. Shrimpton. “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In: *Fast Software Encryption*. Vol. 3017. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004.

[30] R. Schoof. “Elliptic curves over finite fields and the computation of square roots mod p”. In: *Mathematics of computation* 44.170 (1985), pp. 483–494.

[31] S. Singh. *The Code Book*. Fourth Estate, 1999.

[32] D. R. Stinson. *Cryptography: Theory and Practice, 2nd Ed.* Chapman and Hall/CRC, 2002.

[33] M. Thomas. “Will Bitcoin and the Blockchain Change the Way We Live and Work?” Transcript of Gresham College lecture available at https://www.gresham.ac.uk/lectures-and-events/. Jan. 2018.

[34] E. Weisstein. *Elliptic Curves*. URL: http://mathworld.wolfram.com/EllipticCurve.html (visited on Oct. 20, 2017).

[35] M. G. Wilson and A. Yelowitz. *Characteristics of Bitcoin Users: An Analysis of Google Search Data*. 2014. URL: https://ssrn.com/abstract=2518603 (visited on Jan. 12, 2018).