POLITECNICO DI MILANO

Corso di Laurea in Ingegneria Matematica
Scuola di Ingegneria Industriale e dell'Informazione

Schnorr Signature: Additivity and Multisignature

Relatori: **Prof. Daniele Marazzina**
Prof. Ferdinando Ametrano

Tesi di Laurea di:
Chiara Lelli
Matricola 830091

Anno Accademico 2016-2017

A Luca e i miei genitori ...

Contents

List of Figures

List of Tables

Abstract

In 1991 on the *Journal of Cryptology*, Claus Peter Schnorr published a paper titled *"Efficient Signature Generation by Smart Cards"*, where he presented his idea for a new efficient signature scheme.
Even if it has so many interesting features and benefits, it has not been standardised yet. We present our implementation of *Schnorr Signature* applied to Elliptic Curve Cryptography, which is based on the assumption of the Discrete Logarithm Problem.
Throughtout our dissertation, we explain our choices step by step and we illustrate the various properties of this signature, such as shortness, fast verification, but most of all *additivity*. This one is not present in any other signature, and leads to a very important and innovative feature: *multisignature*, a protocol through which a group of signers can generate a single joint signature on a common message.
We illustrate also our implementation of multisignature scheme and the several benefits brougth by this feature.
Finally, we introduce Elliptic Curve Digital Signature Algorithm, currently used in Bitcoin, showing how much improvements Schnorr Signature Algoritmh carries with it.

Chapter 1

Introduction

In this chapter, we illustrate a general overview of cryptography and digital signature and we present the main contribution of this work.

1.1 General overview

In *Encyclopædia Britannica[3]*, Cryptography is defined as "the science of transforming information into a form that is impossible or infeasible to duplicate or undo without knowledge of a secret key". The first known evidence of the use of cryptography dates back to around 1900 BC in Egypt. We have to wait until 100 BC to see the first most famous cipher: Cesar Cipher. He used it to convey secret messages to his army generals. The key was to shift every letter by 3. The most famous example of cryptography is the cipher machine *Enigma*, highly used by German forces during the Second World War.

In 1976 Whitfield Diffie and Martin Hellman published "New Directions in Cryptography", introducing the idea of public key cryptography. In fact, *symmetric cryptography* has some shortcomings. The secret key must be established through a secure channel and has to be known by only two people. Thus, if A wants to communicate through this system with several people, A has to store a large number of secret keys, one for each of the counterparts. There must be mutual trust between the parts and no one should want to cheat the other.

This drawbacks have been overcomed through the introduction of the *public key cryptography*, also known as *asymmetric cryptography*. In this type of systems, the user has a key-pair: a *public key* disclosed, and a *private key*

known only by the user.
The main uses of this system are:

- *public key encryption*: the message is encrypted using the recipient's public key and can be decrypted only by the owner of the corresponding private key.
- *digital signature*: in this case the message is not encrypted, but it is signed using the sender's private key and can be verified by anyone using the corresponding public key.

In this thesis, we focus on a specific digital signature: *Schnorr Signature*. It was proposed in 1989 by Claus Pieter Schnorr, a German mathematician and cryptographer. It was published two years later and suddenly patented. Even if it has always been known to be very simple, while other signatures such as DSA have been standardized, this has not happened to Schnorr signature because of the patent on it. Thus, when in 2005 the patent has expired, people have built on DSA rather than Schnorr signature.

Actually there are some proposals for the implementation of this signature, but still no standard and no code to use it.

1.2 Contributions

This work provides an implementation of *Schnorr Signature Algorithm* applied to Elliptic Curve Cryptography and *Multisignature Protocol*. Since this signature has not been standardized yet, it is possible to find some ideas proposed by some researchers, but not a script on which the schemes are implemented. We chose to follow the guide line designed by Pieter Wuille, a very important Bitcoin Core developer.

We start with an important overview of the mathematics necessary in order to deeply understand Elliptic Curve Cryptography and the assumptions on which it is based. Then we focus on *Schnorr Signature*, starting from the analysis of the idea behind the algorithm, and going on presenting our implementation and explaining it step by step. A key point in our dissertation is the analysis of the benefits of SSA, among which the most important is: *additivity*.

This property is the basis of *multisignature*, the main core of this dissertation. It is a protocol through which a group of signers can generate a single joint signature on a common message. We illustrate our implementation of this feature and we analyse it step by step.

Finally, we present the Digital Signature Algorithm applied to an Elliptic Curve and compare it to ECSSA. We conclude explaining in details why ECSSA should replace ECDSA.

1.3 Structure of the thesis

This work is organized as follows.

Chapter 2 introduces the problem in mathemathical terms and it gives the necessary definitions on which we base our work.

Chapter 3 illustrates Schnorr Signature. Our algorithm is presented and analysed

Chapter 4 presents the most important feature of SSA: Multisignature. In this chapter, we show our implementation explaining it in every detail.

Chapter 5 illustrates the Digital Signature Algorithm and compares it to SSA

Chapter 6 concludes the discussion with a summary of the results and suggests some possible future developments of this work.

Chapter 2

State of the Art

In this chapter, we present some basic notion and definition necessary to deeply understand how elliptic curve cryptography works.
We start with the mathematical foundations, from Modular Arithmetic to Groups and Finite fields. Then we go through an accurate analysis of Elliptic Curve and Hash functions to, finally, comprehend digital signatures.

2.1 Mathematical Fundations

Gauss used to say "*Mathematics is the queen of the sciences, and number theory is the queen of mathematics*" because of its importance in the discipline. Number theory, as it is defined in *Encyclopædia Britannica[3]*, is a branch of pure mathematics concerned with the properties of the positive integers.
It has applications to cryptography and cryptanalysis, particularly with regard to modular arithmetic and diophantine equations (i.e. *elliptic curve*).

2.1.1 Modular Arithmetic

Modular Arithmetic can be informally called the "clock arithmetic". Indeed, the most familiar example of it is in the 12-hour clock, where the day is divided into two 12-hour periods. Here it is visible how numbers "wrap around" upon reaching the *modulus* value, just like the hours of the day do in the clock.
For example, if it is 7:00 AM now, then 8 hours later what time will it be? Typically we would say $7 + 8 = 15$, but in this case we will say 3:00 PM, because clock time "wraps around" every 12 hours. Thus, clock time is an example of *modulus* 12.

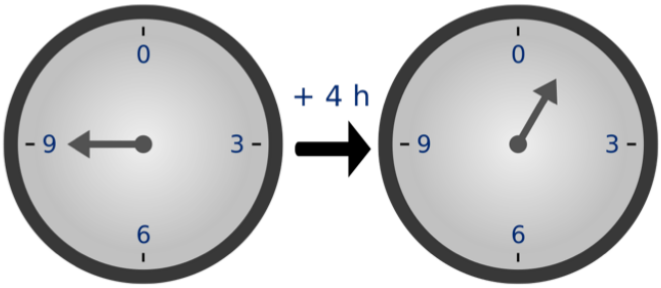

Figure 2.1: Real-life example of modular arithmetic[5].

Definition 2.1. *Given two integers a and b, and a positive one n. a and b are said to be congruent modulo n, if their difference $a - b$ is an integer multiple of n.*
The congruence relation is denoted

$$a \equiv b \bmod n \tag{2.1}$$

The integers modulo m are the possible remainders modulo m. They are denoted by $\mathbb{Z}_m$. The set of integers modulo m is $\mathbb{Z}_m$={0, 1,. . . ,m-1}.

Properties Let a_1, a_2, b_1 and b_2 be integers such that $a_1 \equiv b_1 \bmod n$ and $a_2 \equiv b_2 \bmod n$. Thus:

$$a_1 + a_2 \equiv b_1 + b_2 \bmod n \tag{2.2}$$

$$a_1 - a_2 \equiv b_1 - b_2 \bmod n \tag{2.3}$$

$$a_1 a_2 \equiv b_1 b_2 \bmod n \tag{2.4}$$

these are, respectively, *addition, subtraction* and *multiplication*. Another important operation is the *division*, which should be interpreted as the multiplication by the inverse:

$$ab \bmod n \equiv 1 \bmod n \tag{2.5}$$

Moreover,given a, b:

$$(a \bmod n)(b \bmod n) \equiv (ab) \bmod n \tag{2.6}$$

$$((a \bmod n)(b \bmod n)) \bmod n \equiv (ab) \bmod n \tag{2.7}$$

2.1.2 Groups and Finite Fields

Cryptography has its basis also onto groups, rings and finite fields theory.

Definition 2.2. ***Group (G,*)***
A set G together with a binary operation $*$, that combines two elements a and b to form another element $a * b$, is a group if it satisfies four requirements, known as the group axioms:

- ***Closure**: $\forall a, b \in G,\ a * b \in G$.*
- ***Associativity**: $\forall a, b, c \in G,\ (a * b) * c = a * (b * c)$.*
- ***Identity**: there exists an unique element $e \in G$ such that, $\forall a \in G$, $a * e = e * a = a$.*
- ***Invertibility**: $\forall a \in G$ there exists an element $b \in G$ such that, $a * b = b * a = e$.*

If $\forall a, b \in G\ a*b = b*a$, $(G, *)$ is a *commutative group*, also known as *Abelian Group.*

Example 2.1. *Integers under addition $(\mathbb{Z}, +)$ is an Abelian Group; while integers under multiplication $(\mathbb{Z}, \times)$ is not even a group.*

Example 2.2. *For any modulus p ([0,p-1],+) is a commutative group:*

- *0 is the identity element*
- *$\forall$ a, p-a is the inverse*

Example 2.3. *For any prime number p ([1,p-1], $\times$) is a commutative group:*

- *1 is the identity number*
- *$\forall a$ there exists its inverse, such that $ab \equiv 1 \bmod p$*

A group is called finite if it has a finite number of elements. The number of elements is called the *order* of the group.

A group is *cyclic* if its elements are *generated* from a particoular one, indicated as g.
In this case, the element g is the *generator* of the group.

Definition 2.3. ***Ring (G,+,$\times$)***
A set G together with two binary operations (+, $\times$), is a ring if it satisfies the following requirements, the ring axioms:

- *(G,+) is an abelian group.*

- *(G,×) is a semigroup, which means that it satisfies the associativity property.*
- *× is distributive with respect to +, it means that* $\forall a, b, c \in G$*:*
  1. $a \times (b + c) = (a \times b) + (a \times c)$.
  2. $(a + b) \times c = (a + c) \times (b + c)$.

Definition 2.4. ***Field*** $\mathbb{F}$
A field $\mathbb{F}$ *is a ring* $(\mathbb{F},+,\times)$*, such that* $(\mathbb{F}, \times)$ *is a group, which satisfies all the properties for all the elements but the identity element of the first operation.*

A field which contains a finite number of elements is a *finite field.*

2.2 Elliptic Curve

Now that we have given the preliminary definitions, we can introduce the theory of the Elliptic Curve, which is the basis of this thesis.

Definition 2.5. ***(Formal definition)*** *In mathematics, an elliptic curve is a plane algebraic curve defined in terms of the Weierstrass equation:*

$$y^2 = x^3 + ax + b \tag{2.8}$$

that is non-singular: that is, it has no cusps or self-intersections; so a and b must satisfy $4a^3 + 27b^2 \neq 0$

Depending on the value of a and b, elliptic curves may assume different shapes on the plane.

Figure 2.2: Examples of elliptic curves[7].

The points of an elliptic curve together with the *addition operation, +,* form an *Abelian Group*, where:

- **identity point**: the *point at infinity* $\mathcal{O}$.
- **inverse** of the point P: its symmetric about the x-axis.
- **addition rule**: give three aligned, non-zero points over the EC P, Q and R, $P + Q + R = 0$.

As this is an abelian group, the addition becomes: $P + Q = -R$. So, in order to compute $P + Q$ you have to draw the line passing through P and Q; R is the third intersected point. The result of the addition operation is the inverse of R, $-R$.

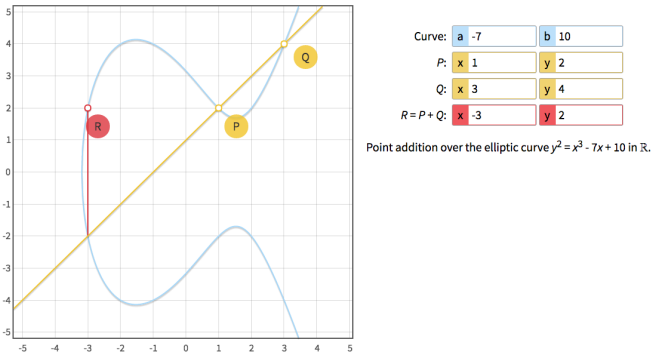

Figure 2.3: Addition[7].

Formally, we should compute:

$$m = \frac{y_P - y_Q}{x_P - x_Q} \tag{2.9}$$

$$x_R = m^2 - x_P - x_Q \tag{2.10}$$

$$y_R = y_P + m(x_R - x_P) \tag{2.11}$$

If $P = Q$, you have to draw the tangent to the curve in Q.

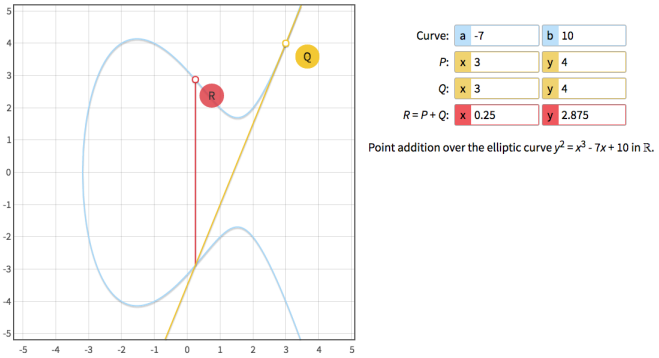

Figure 2.4: Point Doubling[7].

If $P \neq Q$ but the line intersects only two points, then the line should be tanget to the curve. R is the inverse of the point of tangency.

2.2.1 Over a finite field

Let E be an elliptic curve over a finite field $\mathbb{F}_p$, where p is a very high prime number, $p \neq 2, 3$. Then, E is described by the Equation (2.8), where $a, b \in \mathbb{F}_p$, such that $4a^3 + 27b^2 \neq 0$.
The last requirement ensures that E is non-singular, this means in particular that it is possible to compute the tangent in every point of the curve.
The set of the *rational points* in E over $\mathbb{F}_p$, denoted by $E(\mathbb{F}_p)$, is:

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 : \; y^2 \equiv x^3 + ax + b \pmod{p},\; 4a^3 + 27b^2 \not\equiv 0 \pmod{p}\} \cup \{\mathcal{O}\} \tag{2.12}$$

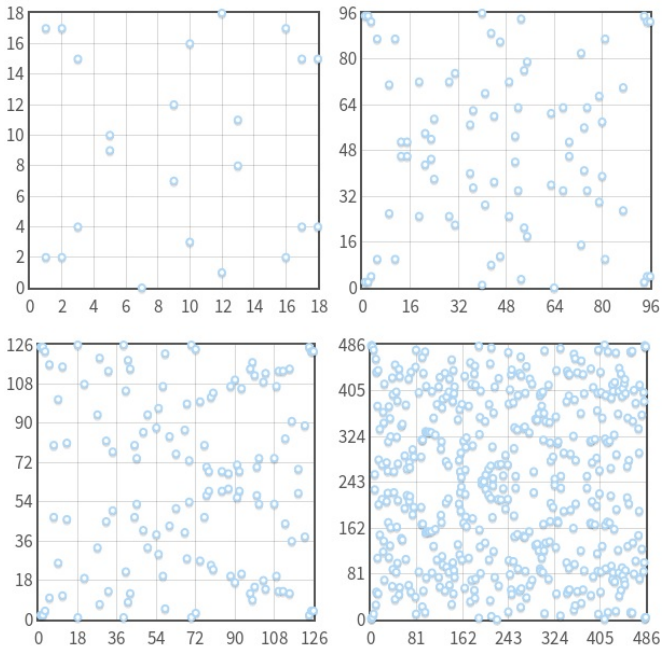

Figure 2.5: Examples of Elliptic Curve over Finite Fields[7].

How does the addition operation changes in a finite field?
Informally, we can say that a line in $\mathbb{F}_p$ is the set of points (x, y) which satisfy the equation $ax + by + c \equiv 0 \pmod{p}$.

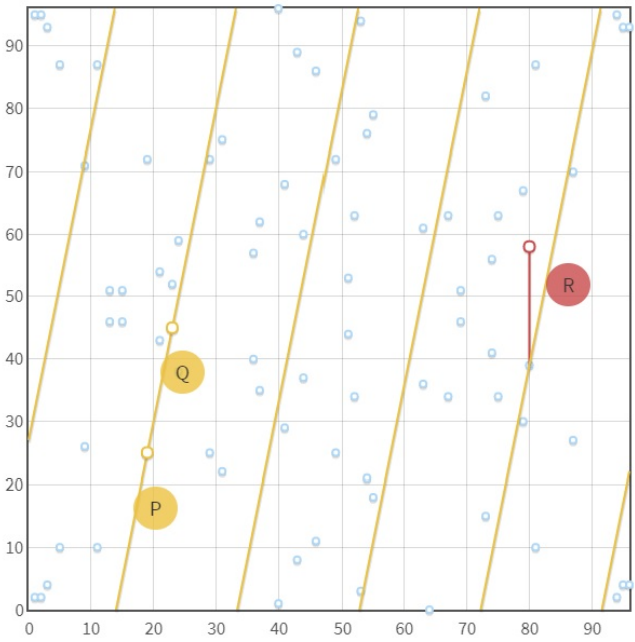

Figure 2.6: Point addition on an EC over a Finite Field[7].

Note that the line $y \equiv 4x + 83 (\text{mod } 127)$ that connects the points "repeats" itself in the plane.
Formally, we should compute:

$$m \equiv \frac{y_P - y_Q}{x_P - x_Q} (\text{mod } p), \tag{2.13}$$

$$x_R \equiv m^2 - x_P - x_Q \ (\text{mod } p), \tag{2.14}$$

$$y_R \equiv y_P + m(x_R - x_P) \ (\text{mod } p). \tag{2.15}$$

Domain Parameters Elliptic curve domain parameters yield a set of information to identify a certain elliptic curve group for use in cryptography. The domain parameters are the finite field $\mathbb{F}_p$, the coefficients a and b of the $Weierstrass$ equation, a base point $G \in E(\mathbb{F}_p)$, its order n, and finally the cofactor $h = \frac{\#E(\mathbb{F}_p)}{n}$.
The base point G generates a cyclic subgroup of order n in $E(\mathbb{F}_p)$ denoted by $\langle G \rangle$, i.e.: $\langle G \rangle = \{G, 2G, \ldots, (n-1)G\} \cup \mathcal{O}$.

Parameter	Explanation
p	A *prime* number specifying the underlying field $\mathbb{F}_p$
a	The *first* coefficient of Weierstrass equation
b	The *second* coefficient of Weierstrass equation
G	The *generator* point
n	The *order* of G
h	The *cofactor* of G

Table 2.1: Elliptic Curve domain parameters over $\mathbb{F}_p$[6].

2.2.2 Bitcoin's Elliptic Curve

Bitcoin uses the Koblitz curve $secp256k1$ [14], which has never been used before. It is characterised by the following parameters domain:

- $p = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ FFFFFC2F = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- $a = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000$
- $b = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000007$
- $G = 04\ 79BE667E\ F9DCBBAC\ 55A06295\ CE870B07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798\ 483ADA77\ 26A3C465\ 5DA4FBFC\ 0E1108A8\ FD17B448\ A6855419\ 9C47D08F\ FB10D4B8$ (uncompressed)
- $n = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ BAAEDCE6\ AF48A03B\ BFD25E8C\ D0364141$
- $h = 01$

So the EC over $\mathbb{F}_p$ is:
$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 :\ y^2 \equiv x^3 + 7 \pmod{p}\} \cup \{\mathcal{O}\}$.
Satoshi Nakamoto chose this curve because is the only one which was generated in a $non - random$ way, so it 30% faster than the others. Moreover, it is defined over a ring, not over a binary Galois field.

2.3 Double and Add Algorithm

In order to compute an elliptic curve point multiplication it is used the *Double* and *Add* algorithm. This is based on the idea that a *scalar* d can be written, in a binary representation, as:

$$d = d_0 + 2^1 d_1 + \cdots + 2^m d_m, \tag{2.16}$$

where $d_i \in \{0,1\} \forall i$.

Example 2.4. *Let's consider the elliptic point multiplication,* $947\ G$. *Starting from the binary representation of 947,* $947 = 2^0 + 2^1 + 2^4 + 2^5 + 2^7 + 2^8 + 2^9$, $947\ G = G + 2G + 4G + 16G + 32G + 128G + 256G + 512G$. *This operation requires only 9 doublings and 6 additions: much better than 947 additions.*

2.4 Discrete Logarithm

The Elliptic Curve Cryptography, known as ECC, that is cryptography based on EC, is particularly interesting because it takes advantage of the fact that the Discrete Logarithm Problem (DLP) for EC is proven to be very "hard" [6].

Definition 2.6. *Let G be a cyclic group of order n with generator g. The discrete logarithm of $h \in G$ to the base g, denoted by $\log_g h$, is the unique integer k, $0 \leq k \leq n-1$, such that $g^k = h$.*
Given g and h, the DLP is to find k.

The best known algorithms to break the elliptic curve discrete logarithm problem take steps proportional to $\sqrt{2^n} = 2^{\frac{n}{2}}$, where n is the number of bits of the key. $secp256k1$ uses 256 bit keys, so the number of steps needed to break it is 2^{128}.

2.5 Hash Function

Definition 2.7. *A hash function H is a one-way function that maps data of arbitrary size to a bit string of a fixed size, the hash value or digest.*

Hash functions have a lot of useful applications in cryptography, such as digital signatures, message authentication, fingerprinting, and so forth.
In order to be suitable for cryptography, an hash function has to satisfy the following requirements:

1. **Pre-image resistance**: given an hash value h it is computationally infeasible to find m such that $h = H(m)$.
2. **Second pre-image resistance**: given an input m_1 it is computationally infeasible to find another input $m_2 \neq m_1$, such that $H(m_1) = H(m_2)$
3. **Collision resistance**: it is computationally infeasible to find two different inputs m_1, m_2, such that $H(m_1) = H(m_2)$.

It is important to notice that *Collision resistance* implies *Second pre-image resistace*, but does not imply *Pre-image resistance.*

Example 2.5. *One of the most diffused hash function is SHA-1 (produces a 160-bit (20-byte) hash value). It was designed by the United States National Security Agency, and is a U.S. Federal Information Processing Standard. Since 2005 SHA-1 has not been considered secure against well-funded opponents and, in 2017, CWI Amsterdam and Google announced they had performed a collision attack against SHA-1.*
So, it is not collision resistant ⇒ it is not a good hash function.

Actually, Bitcoin uses SHA-256, which is one of the successor hash functions to SHA-1, and is one of the strongest hash functions currently available.

Properties As we said before, the size of the possible hash values is smaller than the size of possible input data. Therefore, many input data points will share a single hash value output ⇒ collisions do exist. But you have to try 2^{130} randomly chosen inputs in order to have 99.8% chance that two of them will collide. So, it is computationally unfeasible.
Hence, if we know $h(x) = h(y)$, it is safe to assume that $x = y$.

Another important property is **Puzzle-friendliness**: Given x and a target set Y, to find r from high min-entropy distribution such that $H(x||r) \in Y$, there is no solving strategy better than trying random values of r. (Brute force).
Min-entropy measures how predictable an output is. If this probability is p, then the min-entropy is defined as -$\log_2 p$.
For example, for a fair coin toss, you'd have $p = 0.5$, giving a min-entropy of 1 bit. A uniformly random 256-bit string would have -$\log_2 2^{-256} = 256$ bits of min entropy.
If a distribution has an high min-entropy, it means that the distribution is very spread out. In this context, if the input is chosen from a high min-entropy distribution, it is possible to say that the output belongs to a uniform distribution.
This property is important, because if r is chosen from a high min-entropy distribution, then the likelihood to find x knowing $H(r||x)$ at first guess is 2^{-256}.
An important application of secure hashes is verification of message integrity. Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).
For this reason, most digital signature algorithms only confirm the authenticity of a hashed digest of the message to be "signed". Verifying the authenticity of a hashed digest of the message is considered proof that the

message itself is authentic.

2.6 Elliptic Curve Cryptography

In 1985, Neal Koblitz [10] and Victor S. Miller [13] independently proposed to use the elliptic curves in cryptography. This brought to the birth of Elliptic-curve cryptography (ECC) which is a type of public-key cryptography based on the difficulty of computing discrete logarithms in the group of points on an elliptic curve defined over a finite field.

Definition 2.8. *(**Public-key cryptography**) Public key cryptography, also known as asymmetric cryptography, is any cryptographic system that uses pairs of keys: public keys which can be disclosed, and private keys which must be known only by the owner of the keys.*

2.6.1 Key pair generation

Inputs: The elliptic curve domain parameters $(\overline{p},\ a\ ,\ b,\ G,\ n,\ h)$.
Actions: The following actions are performed:

1. The **private key** is: $p = random(1, 2, \ldots, n-1)$;
2. The **public key** is: $P = p \times G$.

Output: (p, P)

P is a point on the elliptic curve; while p is an integer, the number of additive steps from the generator point G to arrive at point P. Using the *Double* and *Add* algorithm permits to compute P in a polynomial time.
Here it becomes clear why it is fundamental that DLP is such a hard problem over elliptic curves! Indeed, the *private key* must be secret, only the owner should know it; while the *public key* can be known by everyone. That is also the reason why the *prime* number of E should be selected in a proper way: $\overline{p}$ should be a high prime number, so that the order n is high. So this means that it is unfeasible to know and try every single number in order to steal the *private* key.

2.6.2 Signature Protocol

Until now, we have been focused on the mathematical structure that allows us to approach this type of cryptogrphy, but how does a signature algorithm work?
The scenario is the following: Alice wants to sign a message m with her private key (p_A), and Bob wants to validate the signature using Alice's public

key (P_A). Nobody but Alice should be able to produce valid signatures. Everyone should be able to check signatures. They are using the same domain parameters.
First, Alice generates her key pair and hashes the message, $h = H(m)$. With her private key, she generates her signature over h and sends it to Bob.
This last computes h and uses it, together with Alice's public key, in order to verify the validity of the received signature.
In the figure below it is represented this process.

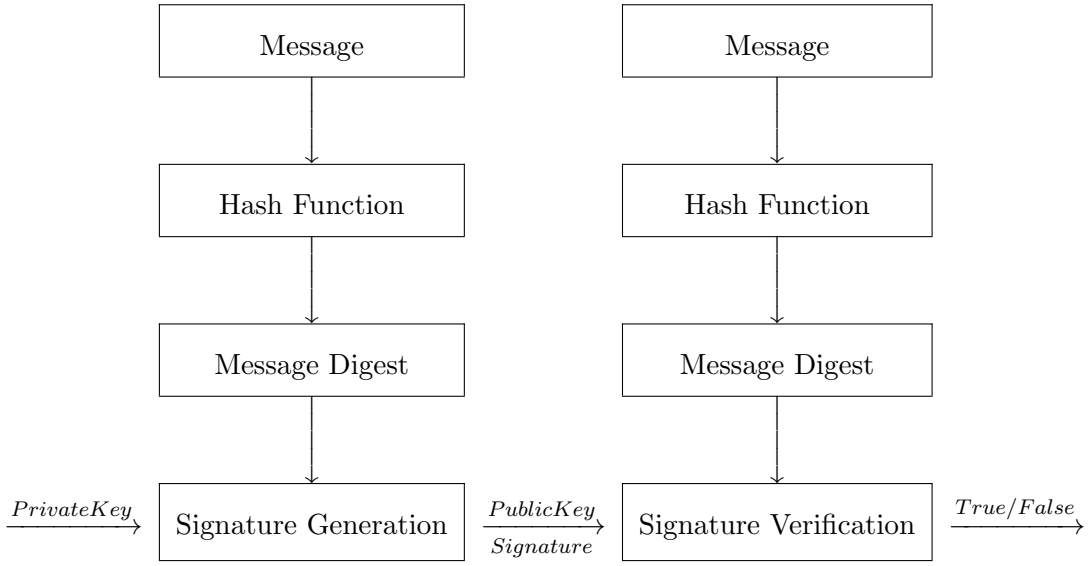

Figure 2.7: Signature Process.

Chapter 3

Schnorr Signature Algorithm

As we said in the introduction, Claus Peter Schnorr presented his idea in 1989, and he promptly patented it. Since the beginning of 2008, when the patent has expired, a lot of possible algorithms for the generation of Schnorr signature have been proposed, but still today there is no standard.
In this chapter, we present the one we think performs better: Schnorr Signature Algorithm over an Elliptic Curve.
First of all we illustrate the steps through which we can generate and verify this type of signature. Then, we continue analyzing each codeline in order to deeply understand why we choose this implementation out of the various proposed by different developers. Finally, we introduce the important benefits of this signature.

3.1 Scheme

Given an elliptic curve over a finite field $\mathbb{F}_p$, a user generates himself a private key p, which is a random number $\in \{1,\ 2,\ldots,\ n\text{-}1\}$, where n is the group order. The corresponding public key P is $p \times G \bmod\ n$.
In order to sign a given message m, the user has to choose another random number $k \in \{1, 2, \ldots, n-1\}$, the *ephemeral private key* (or *nonce*), and compute the *Ephemeral public key*, $K = k \times G$.
The signature consists of a pair of two integers (r, s). The first is computed as the first coordinate of the Ephemeral key; while the second integer, s, is computed as:

$$s = (k - h \times p)(\bmod\ n), \tag{3.1}$$

where $h = H(m||r)$ is the hash value of the concatenation of the message and the first coordinate of the ephemeral key.
The pair (r, s) is published as the signature.

A proceeds as follow to generate the EC-Schnorr signature (r, s) on the message m.

Inputs: The following informations are required as inputs.

1. A's private key p_A and the elliptic curve domain parameters $(\overline{p},\ a\ ,\ b,\ G,\ n,\ h)$.
2. The message m to be signed.

Actions: The following actions are performed:

1. $k = random(1, 2, \ldots, n-1)$
2. $K = k \times G$
3. if K_y is odd:
   a. $k = n - k$
   b. $K_y = p - K_y$
4. $r = K_x$
5. $h = H(m||r)$
   If $h = 0 \mod n$ goto 1.
6. $s = k - h \times p \bmod n$
   If $s = 0$ goto 1.

Output: The ECSSA signature (r, s) over m

3.1.1 Verification

The verification process is very important, because through it we can be sure that the message has been signed by the owner of the private key.
The protocol is simple, it consists of few steps:

- Compute
$$V = K - h \times P; \tag{3.2}$$
- If
$$V = s \times G, \tag{3.3}$$
the signature is verified!

Proof of correctness We want to show that the verification protocol is mathematically correct.

Proof. We can start analysing (3.3). It is immediately visible that if we substitute (3.2) and (3.1) in it, we obtain

$$K - h \times P = (k - h \times p) \times G. \tag{3.4}$$

Knowing that $K = k \times G$,

$$(k - h \times p) \times G = (k - h \times p) \times G \tag{3.5}$$

□

Given a ECSSA signature (r, s) over a message m, the verification procedure is the following:
Inputs: The following informations are required as inputs.

1. A's authentic public key P_A and the elliptic curve domain parameters *($\overline{p}$, a , b G, n, h)*.
2. The message m to be signed.
3. The ECSSA signature (r, s).

Actions: The following actions are performed:

1. if $s \geq order$: `False`;
2. if $r \geq prime$ number: `False`;
3. $h' = H(r||m)$;
4. if $h' = 0$: `False`;
5. $K' = h' \times P + s \times G$;
6. if K'_y is odd: `False`;
7. if $K'_x = r$: `True`.
   Else: `False`

Output: `True`, if the signature is valid, and `False` otherwise

3.2 Step by step analysis

We continue our dissertation analysing each step, trying to highlight the reasons behind our decisions.
First of all it is very important that k is always a random number, otherwise it is possible to find out the *private key*.

Example 3.1. *Lets say that Alice wants to sign two messages, m_1 and m_2, using the same ephemeral nonce, k.*
Everyone can see (r_1, s_1), (r_2, s_2), and the messages. So Bob, who wants to steal Alice's private key, is able to compute:
$h_1 = H(m_1||r_1)$ and $h_2 = H(m_2||r_2)$.
From (3.1)*, Bob can easily trace back to p:*

$$p = \frac{s_2 - s_1}{h_2 - h_1} \bmod n \tag{3.6}$$

In this simple way, everyone can find out Alice's private key and sign in lieu of her.
Going on into our analysis, why do we have to hash the message?
The reasons behind this step have a practical nature: on one hand, in order to include the message in the signature mechanism, we must have an integer, because in (3.1) h multiplies another integer p; on the other hand, using a hash function h we can reduce the amount of bytes; moreover, while the message can vary in its dimension, h has a standard length.

The next step is the generation of the signature s. Schnorr [17] inferred it from the ElGamal signature algorithm:

$$sk = m + xK \pmod{p-1}, \tag{3.7}$$

where x is the *private key*, K is the *ephemeral key* and the k is the *ephemeral nonce*.
Replacing K by $h = H(m, K)$, m can be eliminated. Another semplification comes from the replacements of sk and $p-1$, respectively, by $s-k$ and the prime number q. This transforms (3.7) into:

$$s = k + hp \pmod{q}, \tag{3.8}$$

obtaining a much shorter signature.
This one is slightly different from the (3.1), there is $-p$ in place of p. It would not be a problem except that, in this case, you should compute the *public key* as $-p \times G$ instead of $p \times G$; but they require basically the same computational effort.
We now have understood why to hash the message, but why do we introduce the concatenation of message and x coordinate of the *ephemeral key*?
The reason is strictly linked to the most important property of this signature: *linearity*.
As we will explain in the next section and in the next chapter, Schnorr signature is additive, so the signature of the sum is the sum of the signatures. This brings to amazing benefits, such as supporting naive multisignature, but it has some downside: Schnorr signature does not natively commit to the public key, which means that you can sign message m_1 but you can also declare that you had signed m_2 instead. Nobody can controll this!

Example 3.2. *Let's say that Alice signs a transaction T_B, through which she gives money to Bob. So, she randomly peaks her ephimeral nonce k_B, $K_B = k_B \times G$, if $h_B = H(T_B)$, then $s_B = k_B - p_B h_B$, so the signature is (K_{B_x}, s_B).*
At the same time, Alice can pretend to sign a transaction T_C to Carl.
She can simply choose $k_C = k_B - (h_B - h_C)p$ as her ephimeral nonce, where

$h_C = H(T_C)$; she can keep K_B as Ephimeral key, while the signature becomes:
$s_C = k_C - h_C p = k_B - (h_B - h_C)p - h_C p = k_B - p_B h_B = s_B$

The last step is the output of the algorithm: (K_x, s). Why do we use only the x coordinate of the *Ephemeral key* to sign the message? Why not completely adhere to Schnorr's proposal?
Indeed, he thought about a different output: (h, s), which occupies three-quarters of ours' space.
We chose K_x instead of h because otherwise it would not have been possible to make such an important processesas the *key recovery*.

Definition 3.1 (**Key Recovery**). *The Key recovery is the process through which is possible to trace back to a public key starting from one coordinate.*

This process is very important, because it permits everyone to control the validity of the signature.
Given (r, s), through key recovery we can obtain K and then we can apply the validation process.
We could not have used the y coordinate instead of the x one, because the key recovery would require a higher computational effort. Moreover, there would be three x coordinates among which we should find our key recovered rather than the two y coordinates.
We obtain a 64-bytes signature:

- a 32-byte integer K_x
- a 32-byte integer s

Scheme	**Public key**	**First component**	**Second component**	**Signature size**
[Sc91]	$-p \times G$	$H(m, K)$	$k + ph$	$b + 2b$
EC-SDSA	$-p \times G$	$H(K_x \|\| K_y \|\| m)$	$k + ph$	$2b + 2b$
EC-SDSA-opt	$-p \times G$	$H(K_x \|\| m)$	$k + ph$	$2b + 2b$
EC-FSDSA	$-p \times G$	$K_x \|\| K_y$	$k + ph$	$4b + 2b$
EC-Schnorr	$p \times G$	$H(K_x \|\| m)$	$k - ph$	$2b + 2b$
EC-SSA	$p \times G$	K_x	$k - ph$	$2b + 2b$

Table 3.1: Characteristics of the proposed schemes.

In the table above are reported the different schemes among which we chose ours.
[Sc91] is the one that most conforms to Schnorr's paper, and has the lowest signature's size: $b + 2b$.
The following three schemes are an alteration of the EC-DSA, which is the current signature algorithm that is used by Bitcoin. We will discuss about it in the fifth chapter.
We rejected the first five ides because the first component does not allow the key recovery. In particular, the concatenation of the coordinates of the *Ephemeral key* can be either a component of a point over the eliptic curve or not, so it is unusable; while the other schemes have an hash value as firts component, so no one besides *ECSSA* supports the key recovery.
Moreover, EC-FSDSA has a very big signature.
So, currently the best possible scheme is EC-SSA.

3.3 Benefits

Here we want to highlight some feature of this signature.
The main reason behind the hype around Schnorr signature is the **additivity**.
What is it? And what does it involve?
This allows us to sum up two signatures creating just one valid signature.
Consequently, it brings to Multisignature, which we will thoroughly discuss in the following chapter.

The *non-malleability* is another important feature of this signature.

Definition 3.2. *A signature scheme is malleable if, given a signature s on a message m, one can efficiently derive a signature s' on a message m' = T(m) for an allowed transformation T.*

Consequently, if a signature scheme is non-malleable, a third party that does not have access to private key can not modify the signature without invalidating it.

Another important benefit of our choice is the possibility to implement *batch validation*, which derives from the property of additivity.

Definition 3.3 (**Batch validation of signatures**)**.** *Let l be the security parameter. Suppose (Gen,Sign,Verify) is a signature scheme, $k, n \in poly(l)$, and $(pk_1, sk_1), \ldots, (pk_n, sk_n)$ are generated indipendently according to Gen(1^l). Let PK=$pk_1, \ldots, pk_n$. Then we call batch a batch verification algorithm when the following conditions hold:*

- *If* $pk_{t_i} \in PK$ *and Verify*$(pk_{t_i}, m_i, \sigma_i)$*=1 for all* $i \in [1 \dots n]$*, then Batch*$((pk_{t_1}, m_1, \sigma_1), \dots, (pk_{t_n}, m_n, \sigma_n))$*=1.*
- *If* $pk_{t_i} \in PK$ *and Verify*$(pk_{t_i}, m_i, \sigma_i)$*=0 for some* $i \in [1 \dots n]$*, then Batch*$((pk_{t_1}, m_1, \sigma_1), \dots, (pk_{t_n}, m_n, \sigma_n))$*=0.*

The idea behind batch validation is that you can have multiple sets of combinations of keys and messages and you can verify them all at once.

Chapter 4

Multisignature

In this chapter we want to analyse the most important feature of Schnorr Signature: multisignature.
More often than not, n people are asked to sign the same document or transaction. In these cases, it is used a *multisignature* scheme. It is supported by also others digital signatures, but the amazing thing of Schnorr Signature is that it supports a *native multisignature* scheme.
Firstly, we focus on *additivity*, which is the property that leads to the main core of our work.
We continue showing the scheme that we have implemented and, finally, we analyse the reasons behind every choice taken.

4.1 Key Aggregation

The biggest innovation brought by Schnorr signature is the *Key Aggregation*, which, according to [11], means that the joint signature can be verified exactly as a standard Schnorr signature with respect to a single "aggregated" public key which can be computed from the individual public keys of the signers.
Key aggregation is closely related to *additivity*.

4.1.1 Additivity

Additivity is a very important property of Schnorr signature. In [8], Greg Friedman introduces the connection between additivity and additive signature: given two manifolds M_1, M_2 glued together along a common boundary.

Additivity holds when the signature is additive with respect to this decomposition.
In order to understand this statement, Friedman introduces bilinear forms. In particular, given a bilinear form on finite dimensional $\mathbb{R}$-vector spaces

$$\phi : V \otimes V \to \mathbb{R},$$

it is symmetric if $\phi(v,u) = \phi(u,v)\ \forall v,u \in V$. The matrix representation is $M_{ij} = \phi(e_i, e_j)$.
Two considerations the author does in the paper are very important for our implementation:

- $(V_1, \phi_1),\ (V_2, \phi_2)$ produces $\phi_1 \boxplus \phi_2$ on $V_1 \oplus V_2$:

$$\begin{pmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{pmatrix}$$

  [1] The signature of the sum is the sum of the signatures.

- On $V_1 \bigotimes V_2$, there is a natural form. The signature $\sigma(\phi_1 \otimes \phi_2) = \sigma(\phi_1)\sigma(\phi_2)$[2]

Both the considerations have an important implication:
since the elliptic curve is a finite dimensional $\mathbb{R}$-vector space, Schnorr signature is additive.
We have taken advantage of this property and we have implemented a *multisignature* scheme.

4.2 Scheme

This scheme consists of 3 stages and one preparatory step (Stage 0).

Stage 0 Given the elliptic curve domain parameters *($\overline{p}$, a, b, G, n, h)* and The message m to be signed, each user i has to generate his key pair (p_i, P_i).

Inputs: Every signer i has to provide his public key P_i.

Actions: The following actions are performed:

1. The hash value of P_i, $h'_i = H(P_i)$.
2. $P_{All} = \sum_i h'_i \times P_i$.

Output: P_{All}.

[1] $\boxplus$ symbolises the *free additive convolution*
[2] $\otimes$ symbolises the *tensor product*

Stage 1 Every signer i has to choose his *ephemeral private key* k_i, compute the *Ephemeral public key* K_i and give to the other signers the x coordinate K_{x_i}

Stage 2 In this stage everyone computes his own signature s_i and the *Ephemeral public key* related to the *multisignature*. Everyone has to follow this process.

Inputs: Every signer i has the following informations as inputs.

1. The message m.
2. His private key p_i.
3. His *ephemeral key pair* (k_i, K_i).
4. The x coordinate of the *ephemeral public key* of the other signers, K_{x_j}, $\forall j \neq i$

Actions: Every user, indipendently from the others, performs the following instructions:

1. $p'_i = p_i \times h'_i$
2. if K_i is odd:
   a. $k_i = n - k_i$
   b. $K_{y_i} = \overline{p} - K_{y_i}$
3. $\forall j \neq i$, i recovers K_j
4. $K_{All_i} = \sum_{j \neq i} K_j + K_i$
5. if K_{All_i} is odd:
   a. $k_i = n - k_i$
   b. $K_{All_{i_y}} = \overline{p} - K_{All_{i_y}}$
6. $h_i = H(m||K_{All_{i_x}})$
   If $h_i = 0 \mod n$ goto 1.
7. $s_i = k_i - h_i \times p'_i \mod n$
   If $s_i = 0$ goto 1.

Output: $(K_{All_{i_x}}, s_i)$

Stage 3 In this stage, takes place the combination of the previous outputs, in order to compute the final signature.

Inputs: Every signer i has to provide the following informations as inputs.

1. $K_{All_{i_x}}$.
2. His signature s_i.

Actions: The following instructions are performed:

1. if $K_{All_{i_x}} = K_{All_{j_x}}$ $\forall i, j$: $K_{All_x} = K_{All_{i_x}}$.
2. else: fail.
3. $s_{All} = \sum_i s_i$.
   If $s_{All} = 0$ or $s_{All} \geq n$: fail.

Output: The Multisignature (K_{All_x}, s_{All}) over m

We want to highlight that along the process the signers have to communicate each other some informations, such as between stage 1 and 2: among the inputs of stage 2 there are the x coordinate of the *ephemeral public key* of the other cosigners.

4.2.1 Verification

Through the process explained above, we have obtained one signature in place of N single signatures. The most fascinating property is that this signature acts as a single one, so the verification process is the same presented in Chapter 3, but in this case the inputs are slightly different: P_{All} in place of the single P_i.

- Compute
$$V = K_{All} - h \times P_{All}; \tag{4.1}$$
- If
$$V = s_{All} \times G, \tag{4.2}$$
the signature is verified!

Proof of correctness We want to show that the verification protocol is mathematically correct.

Proof. Knowing that $s_{All} = \sum_i s_i$, it is immediately visible that if we substitute (4.1) and (3.1) in (4.2), we obtain

$$\sum_i K_i - h_i \times (P_i \times h'_i) = \sum_i (k_i - h_i \times p'_i) \times G. \tag{4.3}$$

But $K_i = k_i \times G$, and $h_i = h \ \forall i$ because $h_i = H(m||K_{All_x})$. Moreover, $p'_i = p_i \times h'_i$, where $h'_i = H(P_i)$. Thus,

$$\sum_i k_i \times G - h \times (P_i \times h'_i) = \sum_i (k_i - h \times p_i \times h'_i) \times G. \tag{4.4}$$

□

4.3 Step by step analysis

Now we want to highlight the purpose of each step delineated above.
We should start with stage 1, where signers have to communicate the x coordinate of their *Ephemeral public key*, K_x. This step is fundamental because in stage 2 everyone has to recover the *ephemeral public keys* of the cosigners in order to compute K_{All_i}. Otherwise it would not be possible to generate a key aggregation using the additivity property. Actually, the users could send their entire K_i, but it would duplicate the cost of communication obtaining the same result. Not by chance, we ask to control the parity of their own key before sending it: in this way we know that every key is positive.
Going on in the second stage, the signatures are generated following the usual scheme.
The last steps we should focus on are:

1. $P_{All} = \sum_i h_i \times P_i$
2. $p_i = p_i \times h_i$,

where $h_i = H(P_i)$.
In the original idea there were not the h_i, but they are fundamental. Without them the *Rogue key attack* would be possible, see next section.

4.3.1 Cancellation attack

The *Rogue Key attack*, also known as *Cancellation attack*, is an attack made by malicious users who manipulate public keys in order to produce forgeries of the set of public keys.
Since everyone knows the various public keys P_i and $P_{All} = \sum_i P_i$, the corrupted signers can easly compute their P_i as a combination of all the public keys so that $P_{All} = \sum_j P_j$ where j are the malicious signers.

Example 4.1. *Let Alice and Bob want to sign the same message m through the multisignature scheme where $P_{All} = \sum_i P_i$.*
Alice is honest, while Bob wants to have the control of the signature all alone.
Bob can pretend that his public key is $P'_B = P_B - P_A$; it is important to highlight that he can compute it because P_A is known.
$\implies P_{All} = P'_B + P_A = P_B - P_A + P_A = P_B$
The signature is valid and the verifiers are not able to demonstrate that Bob has ripped off Alice and P'_B is not his real public key.

This type of attacks was the reason why the first proposals failed and were abandoned. The first solution to this problem came from Micali, Ohta, and

Reyzin in 2001 [12], but it was based on an interactive key generation protocol.
Another way to prevent Rogue-key attacks is to require the proof of knowledge of the secret key during public key registration, idea proposed by Thomas Ristenpart and Scott Yilek in their paper published in 2007 [16].
Both the solutions were highly expensive. We implemented the idea explained in [11], because it does not require additional interactions between the signers, much more cheaper than the previouse ideas.

4.4 Benefits

We have analyzed *multisignature* in every step, but why have we implemented it?
This scheme allows us to save a lot of space, because in place of n signatures we end up with only one!
This is fundamental, for example, in Bitcoin system. Bitcoin [14] is a p2p electronic cash system in which all participants (are able to) validate transactions. These transactions consist of outputs, which have a verification key and amount, and inputs which are references to outputs of earlier transactions not spent yet (known as UTXOs). Each input contains a signature. In fact, some outputs even require multiple signatures to be spent. Transactions spending such an output are often referred to as m-of-n multisignature transactions, and the current implementation corresponds to the concatenation of the individual signatures. For each transaction are required from one to m signatures. These transactions are stored in a distributed ledger, known as blockchain, composed by connected blocks where the transactions are stuffed into.
It is obvious that through a *multisignature* scheme, the signature sizes would be lowered to the standard single signature of 64 bytes, regardless of the number of cosigners. Consequently, the transaction's size would diminish lowering the blocks' weight.
In the figure below it is represented the actual blockchain size and the one that it would have had if there had been a multisignature scheme since 2008.

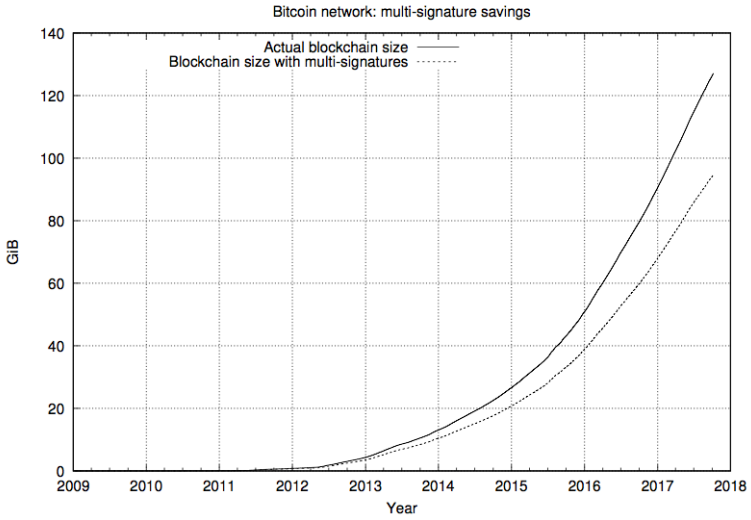

Figure 4.1: Size of the Bitcoin blockchain with and without multi-signatures.

Chapter 5

DSA vs SSA

In this chapter, we introduce the Digital Signature Algorithm, which is the current signature used in Bitcoin.
We analyze the algorithm and we compare it with Schnorr Signature Algorithm.

5.1 Digital Signature Algorithm

Proposed in August 1991 by the U.S. National Institute of Standards and Technology (NIST) and become a U.S Federal Information Processing Standard in 1993, DSA was the first signature scheme accepted legally by the U.S. government [9].
Particularly interesting is the application of DSA to Elliptic Curve Cryptography: ECDSA.
ECDSA works in the group of elliptic curve $E(\mathbb{Z}_P)$. In the early 00s it has been standardized by many standard committees such as ISO, ANSI, IEEE and FIPS. So, when Satoshy Nakamoto decided which standardized signature algorithm to adopt in Bitcoin, ECDSA was the best in circulation. In facts, it has some downsides that, as we are going to show you, could be solved adopting ECSSA.

5.1.1 Scheme

Given an elliptic curve over a finite field $\mathbb{F}_p$, a user generates himself a private key p, which is a random number $\in \{1, 2, \ldots, n\text{-}1\}$, where n is the group order. The corresponding public key P is $p \times G \bmod n$.
In order to sign a given message m, the user has to choose another random number k, the *ephemeral private key*, and compute the *Ephemeral public*

key, $K = k \times G$.
The signature consists of a pair of two integers (x_K, s). The first is computed as the first coordinate of the Ephemeral key; while the second integer, s, is computed as:

$$s = (h - x_K \times p)k^{-1} \bmod n \tag{5.1}$$

where $h = H(m)$ is the hash value of the message.
The pair (x_K, s) is published as the signature.

A proceeds as follow to generate the ECDSA signature (x_K, s) on the message m.

Inputs: The following informations are required as inputs.

1. A's private key p_A and the elliptic curve domain parameters *(p, a , b, G, n, h)*.
2. The message m to be signed.

Actions: The following actions are performed:

1. $k = random(1, 2, \ldots, n-1)$
2. $K = k \times G$
3. $x_K = K_x \bmod n$
4. $h = H(m)$
   If $h = 0 \bmod n$ goto 1.
5. $s = (h + x_K \times p)k^{-1} \bmod n$
   If $s = 0$ goto 1.

Output: The ECDSA signature (x_K, s) over m

5.1.2 Verification

The verification process is very important, because through it we can be sure that the message has been signed by the owner of the private key.
The protocol consists of the following steps:

- Compute:

  1.
  $$u = hs^{-1} \bmod n \tag{5.2}$$

  2.
  $$v = x_K s^{-1} \bmod n \tag{5.3}$$

  3.
  $$(x, y) = u \times G + v \times P \tag{5.4}$$

- If:
$$x = x_K \bmod n \tag{5.5}$$
the signature is verified!

Proof of correctness We want to show that the verification protocol is mathematically correct.

Proof. We can start noticing that (5.5) is true if
$$u \times G + v \times P = K \tag{5.6}$$
Since P is the public key and K is the ephemeral key:
$$(u + vp) \times G = k \times G \tag{5.7}$$
Considering (5.2) and (5.3),
$$(hs^{-1} + x_K s^{-1} p) \times G = k \times G \tag{5.8}$$
$$(h + x_K p)s^{-1} \times G = k \times G \tag{5.9}$$
Since s is the signature,
$$(h + x_K p)(h + x_K p)^{-1} k \times G = k \times G \tag{5.10}$$
$\implies k \times G = k \times G$ identity. □

Given a ECDSA signature (x_K, s) over a message m, the verification procedure is the following:
Inputs: The following informations are required as inputs.

1. A's authentic public key P and the elliptic curve domain parameters *($\overline{p}$, a , b G, n, h)*.
2. The message m to be signed.
3. The ECDSA signature (x_K, s).

Actions: The following actions are performed:

1. if $s \geq n$: `False`;
2. $v = x_K s^{-1} \bmod n$;
3. $h' = H(m)$;
4. if $h' \neq 0$ or $h' < n$:
   - $u = h's^{-1} \bmod n$;
   - $(K' = u \times G + v \times P) \bmod n$;
5. if $K'_x = x_K$: `True`.
   Else: `False`.

Output: `True`, if the signature is valid, and `False` otherwise

5.2 Comparison

Here we want to compare the scheme we propose to the one currently used in Bitcoin: ECSSA vs ECDSA.
We can start analysing (3.1) and (5.1), the generation of s:
they are both quite simple, the second requires a little bit more effort because it uses the multiplication by the inverse of k; anyway, the signature is a couple of integer. They differ in the use of the hash function:
in Schnorr, as we have demonstrated in *Chapter* (3), it must be $h = H(m||K_x)$; while in ECDSA, it can easily be $h = H(m)$. This is because of the linearity, a property that the latter does not have. ECDSA, indeed, is not additive. Why?
The explanation lies in the fact that ECDSA works with just the x-coordinate of the *Ephemeral key*, while Schnorr uses directly the points, so the latter exploits the $shifting\ property$:

Definition 5.1. ***(Shifting Property)*** *Let P be a point on an Elliptic Curve with gerator G, e be an integer, then:*

$$Q = P + e \times G$$

is an EC point. Moreover if $P = x \times G$ then $Q = (x + e) \times G$.

Operating with only one coordinate it is not possible to use this property, because:
$P_x + G_x = Q_x \nRightarrow P + G = Q$.
Looking at the left side, Q_x could be the coordinate of an EC point, but it could also not be in the curve: we are not in a *Cartesian plane*!
Indeed, Schnorr signature supports native multisignature, while it is not possible to implement such a scheme in DSA. Actually, multisignature implies the use of N different signatures. Thus, using the former there would be a considerable save of space! (*see figure* (4.1))

Currently, Bitcoin system uses an operator to check the signatures, OP_CHECKSIG, which requires DER encoding. This adds 6 bytes in each signature, and it is composed by:

- $0x30$ to indicate the a DER encoded signature follows
- 1 byte for length of signature
- $0x02$ to indicate the a integer follows
- 1 byte for length of x_K

The integration of Schnorr signature could bring to the use of a new operator which does not require DER encoding.

Schnorr signature saves up not only space, but also time. Looking at the *verification* process, indeed, in Schnorr is faster and simpler than in DSA. In this contex, it is important to remind what we said in *Chapter* (3), which is that SSA supports *Batch Validation*. It is an amazing feature, because it permits to verify at the same time a set of signatures all together.

Furthermore, DSA has some others drawbacks, such as *mellability*.
So, for example, if (x_K, s) is a valid signature of h, also $(x_K, n - s)$, where n is the *order*, is a valid signature adn everyone can use it.
This cannot happen with SSA.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Throughout this dissertation we have analysed *Schnorr Signature*, a digital signature proposed in 1989 by Schnorr, a German mathematician and cryptography. It has been particularly in the spotlight during the last few years, due to its properties, benefits and features.
Firstly, we have introduced the foundations on which this signature is based, starting from *modular arithmetic* and *groups* and *finite field theory*, continuing with *Elliptic Curve theory*, concluding with *hash functions* and *Discrete Logarithm Problem*.
Then, we have focused on the scheme of Schnorr signature ($Key-Generation$, $Signing$, $Verification$) applied to an Elliptic Curve and on our implementation of it, which we chose among the various proposals. We have explained in every detail our design choices, showing every benefit. Particularly interesting is the property of *additivity*, which enables us to say that "the signature of the sum is the sum of the signatures".
This leads us to the core of this thesis: *multisignature*, an amazing feature of this signature. We have implemented its scheme and analysed it in details, concluding that it would lead to considerable benefits, for example a significant reduction of the transactions' size (see Figure (4.1)).
Finally, we have introduced ECDSA (*Digital Signature Algorithm applied to an Elliptic Curve*) and compare the two signature schemes in order to highlight the advantages of Schnorr signature and to stress the importance of its standardisation.

6.2 Future Work

Our implementation is one of the various proposals that can be found on the internet. Of course, a signature cannot be used if there is not an unique implementation. Thus, in the foreseeable future Schnorr signature should be standardised.
Bitcoin core developers are working on it, so it is conceivable that they might release their algorithm in the next few months. Probably, they will introduce it in Bitcoin system within the end of this year.
As we said in Chapter 3, one of the possible features of Schnorr signature is *Batch Validation*, which may have considerable beneficial effects on the validation of blocks of transactions, almost halving its time. So it will probably be implemented using Schnorr signature.

Bibliography

[1] Bitcoin core. `https://bitcoincore.org`.

[2] Crypto stackexchange. `https://crypto.stackexchange.com`.

[3] Encyclopædya britannica. `https://www.britannica.com`.

[4] Inquiers journal. `http://www.inquiriesjournal.com`.

[5] Wikipedia. `https://en.wikipedia.org`.

[6] *Elliptic Curve Cryptography*. Bundesamt fur Sicherheit in der Informationstechnik, 2007.

[7] Andre corbellini. `http://andrea.corbellini.name`, 2015.

[8] Greg Friedman. Novikov additivity. Department of Mathematics, Texas Christian University, Fort Worth, Texas 76129, USA, 2009.

[9] Don B. Johnson and Alfred J. Menezes. *Elliptic Curve DSA (ECDSA): An Enhanced DSA*. 1998.

[10] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

[11] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. 2018.

[12] Kazuo Micali, Silvio Ohta and Leonid Reyzin. Accountable-subgroup multisignatures. page 245–254. ACM Conference on Computer and Communications Security - CCS 2001, Michael K. Reiter and Pierangela Samarati, editors, 2001.

[13] Victor S. Miller. *Use of Elliptic Curves in Cryptography*. Springer-Verlag, 1986.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.

[15] Jan Paar, Christof Pelzl. *Understanding Cryptography*. Springer, 2010.

[16] Thomas Ristenpart and Scott Yilek. *The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks.* Moni Naor, editor, Advances in Cryptology - EUROCRYPT 2007, volume 4515 of LNCS, pages 228–245, 2007.

[17] Claus Pieter Schnorr. *Efficient Signature Generation by Smart Cards*, volume 4(3), page 161–174. J. Cryptology, 1991.

Ringraziamenti

Vorrei ringraziare i professori Ferdinando Ametrano e Daniele Marazzina, che durante questo percorso di ricerca mi hanno aiutato e consigliato, rendendosi sempre disponibili nei momenti di bisogno. Ringrazio anche Leonardo Comandini, per aver sempre pazientemente risposto alla mie domande e per avermi aiutata con le funzioni usate. Gli faccio un grande in bocca al lupo per la sua tesi e per la sua carriera.

Inoltre ringrazio di cuore i miei genitori e tutti coloro che mi hanno supportato in questi faticosi tre anni; ma soprattutto Chiara, Lorenzo e Stefano, che mi hanno sopportato nei momenti peggiori.

Appendix A

Multisignature

```
# %% combined public key date le chiavi private prv1 e prv2
prv1 = decode_prv('0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d92ad1d
prv2 = decode_prv('0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d

# Inputs
Q1 = pointMultiply(prv1, ec_G)  #moltiplica G per prv1
Q2 = pointMultiply(prv2, ec_G)

# Steps
HQ1 = hash_to_int(sha256(ec_point_to_bytes(Q1, False)))
HQ2 = hash_to_int(sha256(ec_point_to_bytes(Q2, False)))
Q_All = pointAdd(pointMultiply(HQ1, Q1), pointMultiply(HQ2, Q2))

# stage 1
msg = '9788fd27b3aafd1bd1591a1158ce2d8bdc37ab4040dddb64e64d17616e69ce2b'
msg = decode_msg(msg)
m = sha256(msg).digest()

eph_prv1 = 0x012a2a833eac4e67e06611aba01345b85cdd4f5ad44f72e369ef0dd640424dbb
eph_prv2 = 0x01a2a0d3eac4e67e06611aba01345b85cdd4f5ad44f72e369ef0dd640424dbdb

## Steps
R1 = pointMultiply(eph_prv1, ec_G)
if R1[1] % 2 == 1: #must be even
eph_prv1 = ec_order - eph_prv1
R1 = pointMultiply(eph_prv1, ec_G)
R1_x = R1[0]

R2 = pointMultiply(eph_prv2, ec_G)
```

```
if R2[1] % 2 == 1: #must be even
eph_prv2 = ec_order - eph_prv2
R2 = pointMultiply(eph_prv2, ec_G)
R2_x = R2[0]


## stage 2

## steps
prv1 = HQ1* prv1
prv2 = HQ2* prv2



R2_y_recovered = ec_x_to_y(R2_x, 0)
R2_recovered = (R2_x, R2_y_recovered)
R1_All = pointAdd(R1, R2_recovered)

if R1_All[1] % 2 == 1:      # must be even
eph_prv1 = ec_order - eph_prv1
R1_All_x = int_to_bytes(R1_All[0], 32)

e1 = hash_to_int(sha256(R1_All_x + m))
assert e1 != 0 and e1 < ec_order, "sign fail"
s1 = (eph_prv1 - e1 * prv1) % ec_order



R1_y_recovered = ec_x_to_y(R1_x, 0)
R1_recovered = (R1_x, R1_y_recovered)
R2_All = pointAdd(R2, R1_recovered)

if R2_All[1] % 2 == 1:
eph_prv2 = ec_order - eph_prv2
R2_All_x = int_to_bytes(R2_All[0], 32)

e2 = hash_to_int(sha256(R2_All_x + m))
assert e2 != 0 and e2 < ec_order, "sign fail"
s2 = (eph_prv2 - e2 * prv2) % ec_order

## combine stage 2 signatures into a full signature

assert R1_All_x == R2_All_x, "sign fail"
R_All_x = R1_All[0]
```

```
s_All = (s1 + s2) % ec_order
ssasig = (R_All_x, s_All)


#verification
v = ecssa_verify(msg, ssasig, Q_All, hasher = sha256)
print(v)



def ecssa_verify(msg, ssasig, pub, hasher = sha256):
msg = decode_msg(msg)
check_ssasig(ssasig)
pub = decode_pub(pub)
hashmsg = hasher(msg).digest()
return ecssa_verify_raw(hashmsg, ssasig, pub, hasher)

def ecssa_verify_raw(hashmsg, ssasig, pub, hasher):
R_x, s = int_to_bytes(ssasig[0], 32), ssasig[1]
e = hash_to_int(hasher(R_x + hashmsg))
assert e != 0 and e < ec_order, "sign fail, invalid e value"
add1, add2 = pointMultiply(e, pub), pointMultiply(s, ec_G)
assert add1[0] != add2[0], "sign fail, point at infinity"
R_rec = pointAdd(add1, add2)
assert R_rec[1] % 2 == 0, "sign fail, R.y odd"
return R_rec[0] == ssasig[0]
```