Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Mathematical Engineering

An Advanced Signature Scheme: Schnorr Algorithm and its Benefits to the Bitcoin Ecosystem

Supervisors: Prof. Daniele Marazzina
Prof. Ferdinando M. Ametrano

Master thesis by:
Giona Soldati
ID: 874209

Academic year 2017-2018

Inserire frase memorabile.

Qualcuno di tosto.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Da scrivere.

Acknowledgements

Da scrivere.

Chapter 1

Introduction

1.1 Problem under analysis

Descrivi il problema considerato

1.2 Thesis structure

Descrivi la struttura della tesi.

1.3 Notation

- Prime numbers: the lowercase letter p is used to represent an odd prime number;
- Fields: for a general field the letter K is used, while the finite field of order $q = p^k$, where p is an odd prime and k an integer, are represented as $\mathbb{F}_q$;
- Elliptic curves: in general an elliptic curve over a field K is denoted by $E(K)$, which represents the set of points satisfying the generalized Weierstrass equation with coefficients in K plus the point at infinity. Lowercase letters are used to denote scalars, while the uppercase equivalent denotes the linked EC point, e.g. $qG = Q = (x_Q, y_Q)$ (G is reserved to the generator of the group). Whenever a second generator is needed we use the capital letter H: this does not constitute a conflict

with the cofactor since typically the two generators are NUMS (nothing up my sleeves), meaning that we do not know the discrete logarithm of one with respect to the other and viceversa;

- Elliptic curves' key pair: the pair of private and public key is denoted as $\{q, Q\}$, where $Q = qG$. Whenever a second point is needed we use the couple $\{r, R\}$; if more key pairs are needed subscripts are used, e.g. $q_1G = Q_1 = (x_1, y_1)$, $q_2G = Q_2 = (x_2, y_2)$ and $q_3G = Q_3 = (x_3, y_3)$. Notice that, for the sake of clarity, also the coordinate notations is adapted.

- Algorithms:

  - $||$ refers to byte array concatenation;
  - $a \leftarrow b$ refers to the operation of assignment;
  - $z \overset{\$}{\leftarrow} Z$ denotes uniform sampling from the set Z and assignment to z;
  - The function bytes(x), where x is an integer, returns the byte encoding of x;
  - The function bytes(Q), where Q is a point, returns $bytes(0\text{x}02 + (y_Q\&1)) \mid\mid bytes(x_P)$[1];
  - The function int(x), where x is a byte array, returns the unsigned integer whose most significant byte encoding is x;
  - The function hash(x), where x is a byte array, returns the hash of x. In particular, when dealing with Schnorr signature, it returns the 32 byte SHA-256 of x;
  - The function jacobi(x), where x is an integer, returns the Jacobi symbol $\left(\frac{x}{p}\right)$. In general we have:

$$\left(\frac{x}{p}\right) = \begin{cases} 0, & \text{if } \gcd(x, p) \neq 1 \\ \pm 1, & \text{if } \gcd(x, p) = 1 \end{cases}$$

  Moreover we have that if $\left(\frac{x}{p}\right) = -1$ then x is not a quadratic residue modulo p (i.e. has not a square root modulo the field order) and that if x is a quadratic residue modulo p and $\gcd(x, p) = 1$, then $\left(\frac{x}{p}\right) = 1$. However, unlike the Legendre symbol, if $\left(\frac{x}{p}\right) = 1$

[1]This matches the compressed encoding for elliptic curve points used in Bitcoin already, following the standard in [1].

then x may or may not be a quadratic residue modulo p. Fortunately enough, since p is an odd prime we have that the Jacobi symbol is equal to the Legendre symbol. Thus we can check only whether $\left(\frac{x}{p}\right) = 1$ to assess if x is or is not a quadratic residue modulo p.
Moreover, by Euler's criterion we have that $\left(\frac{x}{p}\right) = x^{\frac{p-1}{2}} \pmod{p}$, so that we have an efficient way to compute it.

Chapter 2

Mathematical background

The present study starts with an introduction to the mathematical foundations required for its understanding. The reader that is familiar with abstract algebra and elliptic curve cryptography may skip the first chapter to deal directly with the digital signature schemes presented in Chapter 4.
We start with a brief presentation of the group and field's structures, deferring the introduction to elliptic curves mathematics to the second section.

2.1 Groups and fields

Definition 2.1.1. (Group)*: A group is a non empty set G together with a binary operation $\bullet$ (called the group law of G) that satisfies:*

- *Closure: $\forall a, b \in G \implies a \bullet b \in G$;*
- *Associativity: $\forall a, b, c \in G \implies (a \bullet b) \bullet c = a \bullet (b \bullet c)$;*
- *Identity: $\exists e \in G \mid \forall a \in G,\ e \bullet a = a \bullet e = a$. Such an element e is unique;*
- *Invertibility: $\forall a \in G,\ \exists b \in G \mid a \bullet b = b \bullet a = e$. This element is unique and it is commonly denoted either as a^{-1} or $-a$, depending on the notation (multiplicative or additive).*

As a careful reader may have noticed, it is not required the property of commutativity. This means that, in general, the result of the operation may depend on the order of the operands. If this is not the case, meaning that the group operation is also commutative, the group is called abelian.
Consider now a group $(G, +)$: hereinafter we will stick with the additive

notation. If G is finite, i.e. it has a finite number of elements, we define the order of G as the number of elements in G. We can also define the order of an element $g \in G$ to be the smallest integer $k > 0$ such that $kg = 0$[1], where with 0 we denote the additive identity. If k is the order of g, then:

$$ig = jg \iff i \equiv j \pmod{k}.$$

Indeed:

$$ig = jg \iff (i-j)g = 0 \iff i - j = nk,\ n \in \mathbb{Z}\backslash\{0\},$$

from which the thesis.
Next we state one basic result in group theory, that will turn out to be very useful in the following:

Theorem 2.1.1. (Lagrange's theorem): *Let G be a finite group.*

1. *Let H be a subgroup[2] of G. Then the order of H divides the order of G;*
2. *Let $g \in G$. Then the order of g divides the order of G.*

For the cryptographic applications that we will study a particular family of groups turn out to be very important: cyclic groups. A cyclic group is a group that is generated by a single element: this means that it contains (at least) an element g such that every other element of the group can be obtained by repeatedly applying the group operation to g, i.e. $\forall h \in G$, $\exists n \in \{1, ..., k\} \mid h = ng$, where k as before denotes the order of g. All the group's elements satisfying this property are called generator of the group.
Two interesting properties of cyclic groups are the following:

1. Every cyclic group is abelian;
2. Every finite group has at least a cyclic subgroup.

Now let's get to the mathematical concept of field. A field is a non empty set on which are defined two binary operations, usually called addition and multiplication. It has to be an abelian group under addition (with 0 denoting

[1]In an additive group it is natural to define the scalar multiplication: given $g \in G$ and $k \in \mathbb{N},\ k \neq 0$, we have that $kg = g + g + ... + g$, where the summation is repeated $k-1$ times. In a similar fashion, it is natural to define exponentiation in the multiplicative case.

[2]Given a group $(G, +)$, a subset H is a subgroup of G if H endowed with the restriction of the group operation to $H \times H$ is a group.

the additive identity) and the non-zero elements have to form an abelian group under multiplication. Finally, the multiplication has to be distributive over addition.
More precisely we can give the following definition.

Definition 2.1.2. (Field): *A field is a set $K \neq \emptyset$ endowed with two binary operations $+$ and $*$, such that:*

- *$(K, +)$ is an abelian group, whose identity element is denoted by 0. This group is called the additive group of the field;*
- *$(K\backslash\{0\}, *)$ is an abelian group. This is called the multiplicative group of the field, sometimes denoted by $K^{\times}$ or by K^*;*
- *$\forall a \in K\backslash\{0\},\ \forall b, c:\ a * (b + c) = a * b + a * c.$*

Another key role in the following will be played by finite fields: a finite field is simply a field with a finite number of elements. It can be shown that a finite field of order q, denoted as $\mathbb{F}_q$, exists if and only if $q = p^k$, where p is a prime number and k is a positive integer. The field of a given order is unique, up to an isomorphism (this means that there are different representations of a unique mathematical object). In a field of order p^k, the characteristic of the field (as defined below) is p: thus adding p copies of any element results in the additive identity.
In particular we will deal with prime finite fields, i.e. the finite fields whose order is a prime number. Finite fields of order a prime p may be represented by the set $\{0, 1, ..., p-1\}$ with addition and multiplication defined through modular arithmetic, that is: given $a, b \in \mathbb{F}_p$ then $r = a + b$ and $s = a * b$ are integers in $[0, p-1]$, defined as the remainder of the divisions $(a + b)/p$ and $a * b/p$, respectively. This is usually written as: $a + b \equiv r \pmod{p}$ and $a * b \equiv s \pmod{p}$ or as $a + b \equiv r$ in $\mathbb{F}_p$ and $a * b \equiv s$ in $\mathbb{F}_p$. It is easy to see that in this case the additive identity is the integer 0, while the multiplicative identity is the integer 1.
Subtraction and division are defined via the additive and multiplicative inverses:

- Subtraction: given $a \in \mathbb{F}_p$, we know by definition that there is a unique element $-a \in \mathbb{F}_p$ such that $a + (-a) = 0 \pmod{p}$. Then we define subtraction as: $a - b = a + (-b) \pmod{p}$, $\forall a, b \in \mathbb{F}_p$;
- Division: $\forall a \in \mathbb{F}_p\backslash\{0\}$, $\exists a^{-1} \mid a * a^{-1} = 1 \pmod{p}$, again by the definition. Thus we can defined division as: $a/b = a * (b)^{-1}$, $\forall a, b \in \mathbb{F}_p\backslash\{0\}$.

Notice that not all the finite fields $\mathbb{F}_q$ are isomorphic to $\mathbb{Z}_q$: consider for example the case $q = 2^2 = 4$. Here $p = 2$ and $k = 2$. Since two is a prime number, we have that there exists a field with four elements. But since four is not a prime number, we cannot represent this field through the set $\mathbb{Z}_4 = \{0, 1, 2, 3\}$. Consider the element two: $2 * 0 = 0$ (mod 4), $2 * 1 = 2$ (mod 4), $2 * 2 = 0$ (mod 4), $2 * 3 = 2$ (mod 4). As we can see it does not admit a multiplicative inverse in $\mathbb{Z}_4$.

We end this section giving the definition of field characteristic, that will be useful when defining the equation of an elliptic curve.

Definition 2.1.3. (Field characteristic): *Given a field $(K, +, *)$, we denote by 0 and 1 the additive and multiplicative identities, respectively. The characteristic of K, denoted as char(K), is defined to be the smallest $n \in \mathbb{Z}\backslash\{0\}$ for which the following relation holds:*

$$\underbrace{1 + ... + 1}_{n\ times} = 0$$

In case this relation never holds, the field is said to have characteristic zero.

2.2 Introduction to elliptic curves

An elliptic curve over a field K is a non-singular cubic curve in two variables: it is defined by the equation $f(x, y) = 0$, where f is a polynomial in x and y of degree three with coefficients in K.
In the most general case, an elliptic curve is defined through the following equation: $y^2 + a_1xy + a_2y = x^3 + a_3x^2 + a_4x + a_5$, where $a_1, ..., a_5 \in K$. This form is called generalized Weierstrass equation, and can be simplified according to the characteristic of the field. In particular, if it is different from two and three we can write:

$$y^2 = x^3 + ax + b,\ a, b \in K.$$

This latter equation is called Weierstrass equation, and it is the most widely used.
We will denote an elliptic curve over a field K through the notation $E(K)$, that represents the set of points satisfying the defining equation. By construction, this set also contains a special point, called the point at infinity (its role will be clarified in the next section):

$$E(K) = \{\infty\} \cup \{(x, y) \in K \times K \mid y^2 = x^3 + ax + b\}.$$

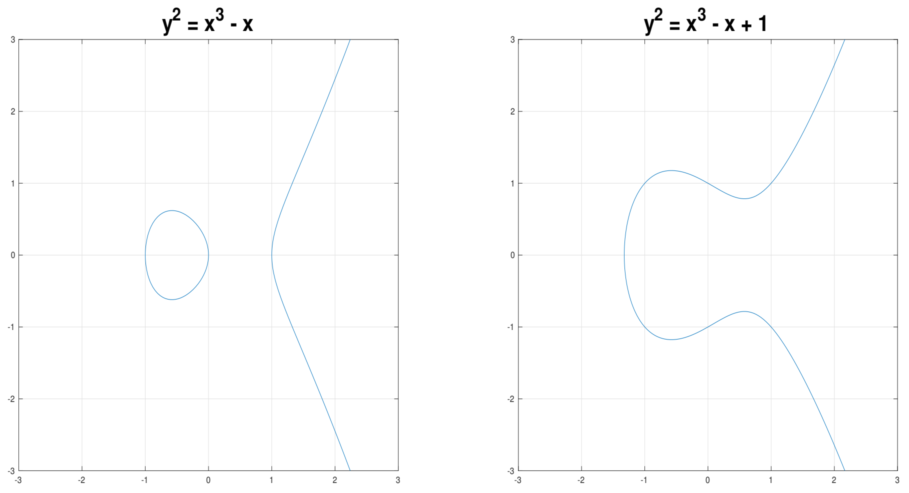


Figure 2.1: Two examples of elliptic curves over the real numbers.

To start familiarizing with elliptic curves, it could be useful to consider the case in which $K = \mathbb{R}$, the set of real numbers. In Figure 2.1 we can look at a couple of elliptic curves' real graph. As we can see, the elliptic curves are symmetric with respect to the x-axis, so that for every feasible x there are two y values satisfying the Weierstrass equation.
Typically the Weierstrass equation is coupled with the constraint that the discriminant of the cubic $x^3 + ax + b$ is different from zero. This is equivalent to ask that:

$$4a^3 + 27b^2 \neq 0,$$

in order to avoid multiple roots. Geometrically, this requirement means that there are no cusps, self intersections or isolated points[3]. We can also link the sign of the discriminant to the number of components of the curve: the real graph of a curve has two components if its discriminant is positive, and one component if it is negative, as exemplified in Figure 2.1.

We get back for a moment to the generalized Weierstrass equation, and show how to go from there to the more appealing Weierstrass form:

$$y^2 + a_1xy + a_2y = x^3 + a_3x^2 + a_4x + a_5,$$

[3]This requirement has also other implications: it can be shown that, without it, the elliptic curve $E_{ns}(K)$ defined over the non-singular points has an isomorphism with simpler algebraic structures, such as K itself or $K^\times$. This problem typically arises when dealing with elliptic curves modulo composites.

with $a_1, ..., a_5$ constants in K. We have said that this is the most generic form for an elliptic curve, but what does it exactly mean? It means that it is always possible to reduce to the Weierstrass equation, unless the characteristic of the field over which the curve is defined is two or three. In these two cases, we need to resort to the general formula.
However, if the characteristic of the field is not two we can divide by two[4] and complete the square:

$$\left(y + \frac{a_1 x}{2} + \frac{a_2}{2}\right)^2 = x^3 + \left(a_3 + \frac{a_1^2}{4}\right)x^2 + \left(a_4 + \frac{a_1 a_2}{4}\right)x + \left(\frac{a_2^2}{4} + a_5\right),$$

which can be written as

$$y_1^2 = x^3 + {a_1}'x^2 + {a_2}'x + {a_3}',$$

with $y_1 = y + \frac{a_1 x}{2} + \frac{a_2}{2}$ and new constants ${a_1}', {a_2}', {a_3}'$. Moreover, if the characteristic is neither three, then we can consider $x_1 = x + \frac{{a_1}'}{3}$:

$$y_1^2 = x^3 + {a_1}'x^2 + {a_2}'x + {a_3}' = \left(x_1 - \frac{{a_1}'}{3}\right)^3 + {a_1}'\left(x_1 - \frac{{a_1}'}{3}\right)^2 + {a_2}'\left(x_1 - \frac{{a_1}'}{3}\right) + {a_3}' =$$

$$= \left(x_1 - \frac{{a_1}'}{3}\right)\left(x_1^2 + \frac{{a_1}'^2}{9} - \frac{2}{3}{a_1}'x_1 + {a_1}'x_1 - \frac{{a_1}'^2}{3} + {a_2}'\right) + {a_3}' =$$

$$= \left(x_1 - \frac{{a_1}'}{3}\right)\left(x_1^2 + \frac{1}{3}{a_1}'x_1 - \frac{2}{9}{a_1}'^2 + {a_2}'\right) + {a_3}' =$$

$$= x_1^3 + \cancel{\frac{1}{3}{a_1}'x_1^2} - \frac{2}{9}{a_1}'^2 x_1 + {a_2}'x_1 - \cancel{\frac{1}{3}{a_1}'x_1^2} - \frac{1}{9}{a_1}'^2 x_1 + \frac{2}{27}{a_1}'^3 - \frac{1}{3}{a_1}'{a_2}' + {a_3}' =$$

$$= x_1^3 + \left({a_2}' - \frac{1}{3}{a_1}'^2\right)x_1 + \left(\frac{2}{27}{a_1}'^3 - \frac{1}{3}{a_1}'{a_2}' + {a_3}'\right).$$

Through a proper renaming of the constants we get finally to the well known Weierstrass form: $y_1^2 = x_1^3 + ax_1 + b$ for some constants a and b in K.
Finally we present the case of the equation $cy^2 = dx^3 + ax + b$ and how to transform it in Weierstrass form. We can multiply both sides by c^3d^2 and write $(c^2dy)^2 = (cdx)^3 + (ac^2d)(cdx) + bc^3d^2$. The change of variables $y_1 = c^2dy$ and $x_1 = cdx$ yields to the desired formulation of the equation.

[4]Remember that in a field of characteristic two the number two acts as zero, the additive element that has not a multiplicative inverse.

2.3 The group law

This section is devoted to the definition of an addition operation between elements in $E(K)$: this operation turns out to be fundamental since it induces the structure of abelian group on the elliptic curve's point. Moreover, we will see why, in the definition of the curve, we had to consider the point at infinity. In particular we will consider K to be the set of real numbers, in order to give an easy graphical representation of the operation. This can be done without loss of generality, since the formulas that we will derive can be shown to be valid for every K with $\text{char}(K) \neq 2, 3$.

Loosely speaking we can say that:

- The points of the elliptic curve are the group's elements;
- The identity element is the point at infinity, denoted by ∞;
- The inverse of a point Q is the one symmetric about the x-axis: $Q = (x, y) \Longrightarrow -Q = (x, -y)$;
- Addition is given by the following rule: given three aligned points on the curve Q_1, Q_2 and Q_3 their sum is $Q_1 + Q_2 + Q_3 = \infty$.

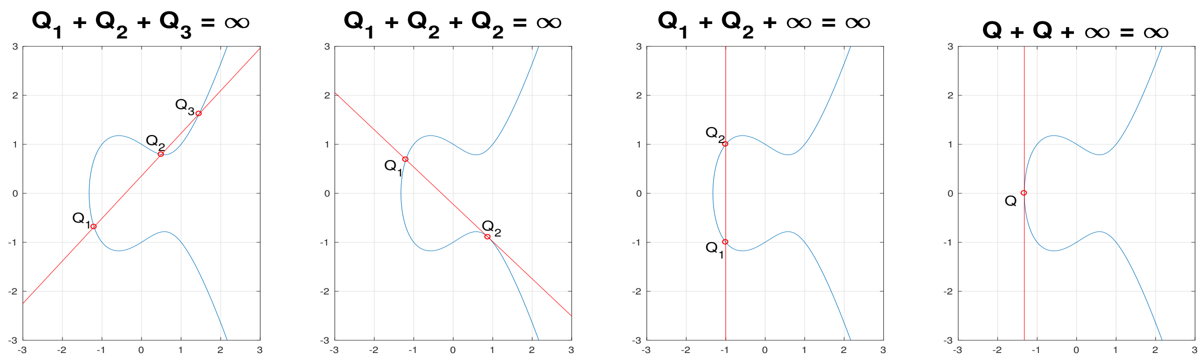


Figure 2.2: Geometrical interpretation of the addition of EC points.

In order to define properly the addition of EC points we need to consider five different cases. We start taking an elliptic curve E defined by the usual equation in Weierstrass form $y^2 = x^3 + ax + b$, $a, b \in \mathbb{R}$ and two points $Q_1 = (x_1, y_1), Q_2 = (x_2, y_2) \in E$. Unless otherwise specified, we assume also that $Q_1, Q_2 \neq \infty$.

1. $x_1 \neq x_2$: consider the line passing through Q_1 and Q_2. In most of the cases (i.e. when neither the points are tangent points) this line will

intersect the curve in a third point, that we denote as $-Q_3$. Then take the reflection with respect to the x-axis (i.e. change the sign of the y coordinate): denote this final point as Q_3.
The algorithm specified allows us to give meaning to the formula:

$$Q_1 + Q_2 = Q_3.$$

The one outlined above is the geometrical intuition, but we can obtain algebraic formulas for the operation. The line passing through Q_1 and Q_2 has equation $y = m(x - x_1) + y_1$, where $m = \frac{y_2 - y_1}{x_2 - x_1}$ is its slope. Notice that if $x_1 = x_2$ the line is vertical, but we will treat this case later on. We can find the intersections between the line and the curve through a substitution: $(m(x - x_1) + y_1)^2 = x^3 + ax + b$. We have $m^2(x^2 - 2x_1x + x_1^2) + 2my_1(x - x_1) + y_1^2 = x^3 + ax + b$; this formula can be rearranged in the form:

$$x^3 - m^2x^2 + (a + 2m^2x_1 - 2my_1)x + (b + 2my_1x_1 - m^2x_1^2 - y_1^2) = 0.$$

In general solving a cubic equation is non trivial, but we have already two roots, namely x_1 and x_2 since Q_1 and Q_2 are points on both the line and the curve. To find the third root we notice that, given a cubic polynomial $x^3 + ax^2 + bx + c$ with roots r, s and t, we can write:

$$x^3 + ax^2 + bx + c = (x - r)(x - s)(x - t) =$$

$$= x^3 - (r + s + t)x^2 + (rs + rt + st)x - rst.$$

Therefore $r + s + t = -a$, so that, if we know two roots r and s, we can find the third as $t = -a - r - s$.
In our case we have $x_1 + x_2 + x_3 = m^2 \implies x_3 = m^2 - x_1 - x_2$. Remembering to change the y coordinate of the intersection point due to the reflection, we get Q_3 as: $x_3 = m^2 - x_1 - x_2$ and $y_3 = m(x_1 - x_3) - y_1$.

2. $x_1 = x_2 \wedge y_1 \neq y_2$: the line through Q_1 and Q_2 is vertical, therefore it intersects E at ∞. Reflecting the point at infinity across the x-axis yields to the same point. Therefore, in this case $Q_1 + Q_2 = \infty$.
   This is in agreement with what we stated before: if $x_1 = x_2 \wedge y_1 \neq y_2$ it means that $Q_2 = -Q_1$, and since the point at infinity acts as the identity element we have exactly that $Q_1 + Q_2 = \infty$.

3. $Q_1 = Q_2 = (x_1, y_1) \wedge y_1 \neq 0$: when two points on a curve are very close one another, the line through them approximates a tangent line.

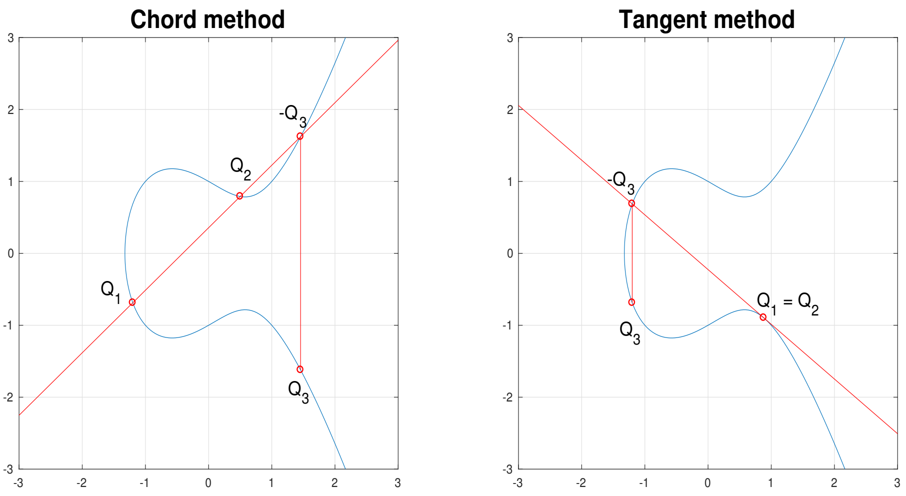


Figure 2.3: Point addition: graphical representation.

Therefore, when the two points coincide, we take the line through them to be the tangent line. Implicit differentiation allows us to find its slope m:

$$y^2 = x^3 + ax + b \implies \frac{d(y(x)^2)}{dx} = \frac{d}{dx}(x^3 + ax + b) \implies$$

$$\implies \frac{d(y^2)}{dy}\frac{dy}{dx} = 2y\frac{dy}{dx} = 3x^2 + a \implies$$

$$\implies m = \left(\frac{dy}{dx}\right)_{x_1} = \frac{3x_1^2 + a}{2y_1}.$$

Again, the equation of the line is $y = m(x - x_1) + y_1$. At this point we can proceed as before: the line intersects the curve in a second point. Substituting the equation of the straight line would result in a cubic equation. But we already know two roots, so that we get: $x_3 = m^2 - 2x_1$ and $y_3 = m(x_1 - x_3) - y_1$.

4. $Q_1 = Q_2 = (x_1, y_1) \wedge y_1 = 0$: as we can see from the previous point in this case the tangent line is vertical and we set $Q_1 + Q_2 = \infty$.

5. The last case we have to deal with is when one of the two points is ∞. Suppose, without loss of generality, that $Q_2 = \infty$; the line through Q_1 and ∞ is a vertical line that intersects E in the point $-Q_1$, which

is the reflection of Q_1 across the x-axis. When we reflect $-Q_1$ to get $Q_3 = Q_1 + Q_2$, we get back to Q_1. Therefore:

$$Q_1 + \infty = Q_1$$

for all points Q_1 on E. Of course, we extend this to include $\infty + \infty = \infty$.

Thanks to these algebraic formulas we can notice that when Q_1 and Q_2 have coordinates in a field K that contains a and b, then $Q_1 + Q_2$ also has coordinates in K. This means that $E(K)$ is closed under the defined addition operation.

Now we state a theorem showing formally that the couple $(E, +)$ is indeed an abelian group. The theorem with the proof can be found in [12].

Theorem 2.3.1. *The operation + as defined above on an elliptic curve E satisfies the following properties:*

1. *Commutativity:* $Q_1 + Q_2 = Q_2 + Q_1,\ \forall Q_1, Q_2 \in E$;
2. *Identity:* $Q + \infty = Q,\ \forall Q \in E$;
3. *Invertibility:* $\forall Q \in E,\ \exists Q' \in E \mid Q + Q' = \infty$. *This point will be denoted by* $-Q$;
4. *Associativity:* $(Q_1 + Q_2) + Q_3 = Q_1 + (Q_2 + Q_3),\ \forall Q_1,\ Q_2, Q_3 \in E$.

Although we won't prove the theorem, it can be useful to give a sketch of its proof. The commutativity is obvious, either from the formulas or from the fact that the line passing through Q_1 and Q_2 is the same as the line passing through Q_2 and Q_1. The identity property of the point at infinity holds by definition, as the existence of the inverse element.
The tricky part of the proof is related with associativity: either it can be proved working out the annoying computations or through a much more elegant and complex method based on projective coordinates, for which we again refer to the bibliography.

Now that we have clear in mind what does addition between EC points mean, we can define subtraction and scalar multiplication:

- Subtraction: given $Q, R \in E$, we set $Q - R = Q + (-R)$. We can do this since everything is perfectly defined in the right hand side of the equation;

- Scalar multiplication: $\forall k \in \mathbb{N}\backslash\{0\},\ kQ = Q + Q + ... + Q$, where the addition is repeated $k-1$ times.

Notice that to compute the scalar multiplication for a large k it is inefficient to add Q to itself repeatedly. There are several faster approaches, the simplest called the **double and add algorithm**. We stress this fact since, as we will see later on, scalar multiplication and its computational asymmetry are the key ingredients for elliptic curve cryptography: the fact is that it is easily computable[5] but its inversion, the so called discrete logarithm (DL), altough not provable to be hard, is believed to be so: this means that in general we have no efficient algorithm to compute it.

Double and Add Algorithm: To compute kQ, start with the binary representation of k: $k = k_0 + 2k_1 + 2^2k_2 + ... + 2^m k_m$, where $k_0, ..., k_m \in \{0, 1\}$.

Algorithm 2.1 Double and Add algorithm

```
1: procedure DOUBLE_AND_ADD($k$, $Q$)
2:     $R \leftarrow \infty$
3:     for $i \leftarrow 1, m$ do
4:         if $k_i = 1$ then
5:             $R \leftarrow Q + R$
6:         end if
7:         $Q \leftarrow Q + Q$
8:     end for
9:     return $R$
10: end procedure
```

The algorithm can be further sped up through the pre-computation of a vector of multiples of Q: this approach is not always feasible, since the point Q is not always fixed[6].

[5]The double and add algorithm requires polynomial time in the number of bits n representing the numbers we are dealing with, compared to the exponential time required by the naive approach; that is, assuming that the point doubling and the point addition require a constant time, we need $O(n^\alpha),\ \alpha > 1$ elementary operations, compared to $O(2^n)$. In particular on average it requires n multiplications and $0.5*n$ additions, since the binary expansion of a random integer has an equal number of 1s and 0s.

[6]The point Q is fixed, for example, when computing the public key from a private key. For a thorough exposition of these concept we refer to the chapter on elliptic curve cryptography.

Chapter 3

Elliptic curve cryptography

In this chapter we will analyse the cryptographic primitives and assumptions that will allows in the next chapter to introduce the digital signature schemes that are the core of the present work. The first argument will be the specialization of elliptic curves to finite fields, the mathematical structure underpinning the whole elliptic curve cryptography.

3.1 Elliptic curves over finite fields

Up to now we considered the field over which the curve is defined to be the set of real numbers: this is not at all the unique possibility. The typical choice in cryptography is $K = \mathbb{F}_p$, where $\mathbb{F}_p$ is the finite field with p elements. Since there are only finitely many pairs (x, y) with $x, y \in \mathbb{F}_p$, the group $E(\mathbb{F}_p)$ is finite. In practice we consider:

$$\{(x, y) \in \mathbb{F}_p^2 \mid y^2 \equiv x^3 + ax + b \pmod{p},\ 4a^3 + 27b^2 \not\equiv 0 \pmod{p}\} \cup \{\infty\}.$$

It can be shown that the formulas for point addition are the same derived in Section 2.3, but where the calculations are done modulo p[1].

[1]Notice that reduction modulo p can be executed much faster if the prime p is a Mersenne or a pseudo-Mersenne prime, i.e. if $p = 2^d - 1$ for some d in the first case or if $p \simeq 2^d$ in the second one.

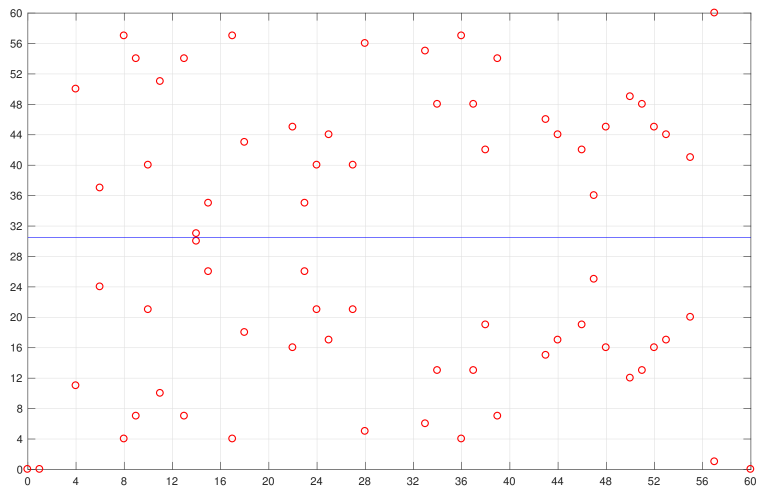


Figure 3.1: The curve $y^2 = x^3 - x$ over $\mathbb{F}_{61}$.

In Figure 3.1 it is represented the typical shape of an elliptic curve over a finite field: we can see that we are not anymore dealing with a smooth curve, but with a finite set of points scattered along the plane. We can notice a certain degree of symmetry with respect to the line $y = \frac{p}{2}$, with the exception of the points with y coordinate equal to zero, that correspond to the roots of the cubic $x^3 + ax + b$ in $\mathbb{F}_p$.
It is still possible to define a geometric method for point addition: the general idea is that we draw the line passing through the points considered. Since we are working in modular arithmetic, this line "repeats" itself along the plane. Once the line intersects a third point, we take the opposite one to be the result of the addition. The approach is presented in Figure 3.2, however the method can be somewhat counterintuitive, especially when dealing with point doubling, so we stick with the algebraic notation.

The number of points on an EC over $\mathbb{F}_q$ plays a central role in the cryptographic applications, so we state here a useful theorem that allows to give an estimate:

Theorem 3.1.1 (Hasse's theorem). *Let $\mathbb{F}_q$ be a finite field and let E be an elliptic curve defined over $\mathbb{F}_q$. Then the order of $E(\mathbb{F}_q)$ (i.e. the number of points) satisfies:*

$$|q + 1 - \#E(\mathbb{F}_q)| \leq 2\sqrt{q}$$

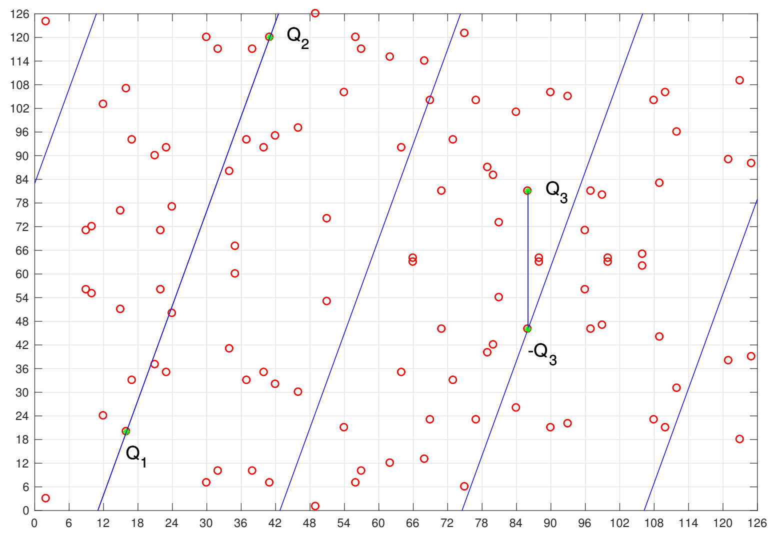


Figure 3.2: Geometric representation for the addition of points on the elliptic curve $y^2 = x^3 - x + 3$ over $\mathbb{F}_{127}$.

Loosely speaking, Hasse's theorem tells us that the number of points over $E(\mathbb{F}_q)$ increases linearly with the order of the field.
Another observation we want to do is that if $\#E(\mathbb{F}_p)$ is a prime number, then $E(\mathbb{F}_p)$ is a cyclic group: this is a direct consequences of Lagrange's theorem. Since the order of each subgroup is a divisor of the order of the group, we have that, if the order of the group is a prime number, there is no subgroup (except the trivial one, comprising only the point at infinity). This means that it had to exists a generator of the whole group. Suppose this is not the case, i.e. $\nexists G \in E(\mathbb{F}_p) \mid \forall Q \in E(\mathbb{F}_p)\ \exists k \in [1, ..., \#E(\mathbb{F}_p)]$ s.t. $Q = kG$. This means that, $\forall G \in E(\mathbb{F}_p)$, there exists at least a Q for which the relation $Q = kG$ does not hold. Fix G and start adding it repeatedly to itself. Of course we would reach different points in $E(\mathbb{F}_p)$ due to the closure of the addition. However we won't be able to reach Q, by definition. But eventually we would reach the point at infinity (when k gets equal to the order of G): then we would start again, meaning that we would have found a cyclic subgroup. But this is in contrast with Lagrange's theorem, so that we can conclude that the whole set of elliptic curve's points form a cyclic group if $\#E(\mathbb{F}_q)$ is a prime number.

3.1.1 Jacobian coordinates

We have told that for EC over finite fields the same formulas previously seen hold, with the calculations modulo p. We have not told that this comes with a downside: in particular we have now to deal with modular inversion, an operation that, although can be done efficiently, is up to two orders of magnitude slower than field's multiplication.
In the next chapter we will deal with digital signatures and we will see that ECDSA requires modular inversion. This traduces in poor performances when it comes to a system like Bitcoin in which signatures are (potentially) verified by each participant in the network to validate transactions. Therefore, it is sometimes advantageous to avoid inversion in the formulas for point addition. This can be done through a change of coordinates: the approach consists in writing all the points as points in a projective space, the key idea being to defer the divisions by multiplying them into a denominator, represented then as a new coordinate, instead of performing every division immediately. Only at the very end we perform a single division to convert from projective coordinates back to affine coordinates.

Jacobian coordinates are a modification of projective coordinates, typically used since they lead to a faster doubling procedure. In affine form, each elliptic curve point has two coordinates (x, y). In projective form, each point will have three coordinates (X, Y, Z), with the restriction that Z is never zero for points different from the point at infinity. The forward mapping is given by $(x, y) \mapsto (xz^2, yz^3, z)$, for any non zero z (usually chosen to be 1 for convenience). The reverse mapping is given by $(X, Y, Z) \mapsto (X/Z^2, Y/Z^3)$, as long as Z is non zero.
The elliptic curve $y^2 = x^3 + ax + b$ becomes:

$$Y^2 = X^3 + aXZ^4 + bZ^6.$$

The point at infinity now has the coordinates $\infty = (1 : 1 : 0)$: indeed, substituting $Z = 0$ in the equation defining the curve we get $Y^2 = X^3$; then we simply normalize.
Let $P_i = (X_i : Y_i : Z_i)$, $i = 1, 2$ be points on the elliptic curve $Y^2 = X^3 + aXZ^4 + bZ^6$. Then:

$$(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2),$$

where X_3, Y_3 and Z_3 are computed as follows:

- $P_1 \neq \pm P_2$: we have the affine test $x_1 = x_2$, that in jacobian coordinates correspond to the check $X_1/Z_1^2 = X_2/Z_2^2 \implies X_1Z_2^2 = X_2Z_1^2$.

We start again from the slope of the line passing through P_1 and P_2.

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{Y_2/Z_2^3 - Y_1/Z_1^3}{X_2/Z_2^2 - X_1/Z_1^2} = \frac{Y_2/Z_2^3 - Y_1/Z_1^3}{X_2/Z_2^2 - X_1/Z_1^2} \cdot \frac{Z_1^3 Z_2^3}{Z_1^3 Z_2^3} =$$

$$= \frac{Y_2 Z_1^3 - Y_1 Z_2^3}{X_2 Z_1^3 Z_2 - X_1 Z_1 Z_2^3}.$$

Defining $T = Y_1 Z_2^3$, $U = Y_2 Z_1^3$, $W = U - T$, $R = X_1 Z_2^2$, $S = X_2 Z_1^2$ and $V = S - R$ we can write:

$$m = \frac{U - T}{S Z_1 Z_2 - R Z_1 Z_2} = \frac{W}{V Z_1 Z_2}.$$

Now consider:

$$x_3 = m^2 - x_1 - x_2 = \frac{W^2}{V^2 Z_1^2 Z_2^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2} = \frac{W^2}{V^2 Z_1^2 Z_2^2} - \frac{X_1}{Z_1^2} \cdot \frac{V^2 Z_2^2}{V^2 Z_2^2} - \frac{X_2}{Z_2^2} \cdot \frac{V^2 Z_1^2}{V^2 Z_1^2} =$$

$$= \frac{W^2 - V^2 X_1 Z_2^2 - V^2 X_2 Z_1^2}{V^2 Z_1^2 Z_2^2} = \frac{W^2 - V^2 R - V^2 S}{V^2 Z_1^2 Z_2^2} = \frac{W^2 - V^2(S + R)}{V^2 Z_1^2 Z_2^2} =$$

$$= \frac{W^2 - V^2(S - R + 2R)}{V^2 Z_1^2 Z_2^2} = \frac{W^2 - V^3 - 2RV^2}{V^2 Z_1^2 Z_2^2}.$$

$$y_3 = m(x_1 - x_3) - y_1 = \frac{W}{V Z_1 Z_2} \left(\frac{X_1}{Z_1^2} - \frac{W^2 - V^3 - 2RV^2}{V^2 Z_1^2 Z_2^2} \right) - \frac{Y_1}{Z_1^3} =$$

$$= \frac{W}{V Z_1 Z_2} \left(\frac{V^2 X_1 Z_2^2 - W^2 + V^3 + 2RV^2}{V^2 Z_1^2 Z_2^2} \right) - \frac{Y_1}{Z_1^3} =$$

$$= \frac{W(RV^2 - [W^2 - V^3 - 2RV^2]) - V^3 Y_1 Z_2^3}{V^3 Z_1^3 Z_2^3} =$$

$$\frac{W(RV^2 - [W^2 - V^3 - 2RV^2]) - TV^3}{V^3 Z_1^3 Z_2^3}.$$

Thus, we can write:

$$X_3 = W^2 - V^3 - 2RV^2,\ Y_3 = W(RV^2 - X_3) - TV^3,\ Z_3 = V Z_1 Z_2,$$

where

$$T = Y_1 Z_2^3,\ U = Y_2 Z_1^3,\ R = X_1 Z_2^2,\ S = X_2 Z_1^2,\ V = S - R,\ W = U - T.$$

- $P_1 = P_2$: now we have to check if $y = 0$. This is equivalent to check if $Y/Z^3 = 0 \implies Y = 0$, since $Z \neq 0$.

  $$m = \frac{3x^2 + a}{2y} = \frac{3(X/Z^2)^2 + a}{2Y/Z^3} = \frac{3X^2/Z^4 + a}{2Y/Z^3} \cdot \frac{Z^4}{Z^4} = \frac{3X^2 + aZ^4}{2YZ} = \frac{W}{2YZ},$$

  if we define $W = 3X^2 + aZ^4$.

  $$x_3 = m^2 - 2x = \frac{W^2}{4Y^2Z^2} - 2\frac{X}{Z^2} = \frac{W^2 - 8XY^2}{4Y^2Z^2} = \frac{W^2 - 2V}{4Y^2Z^2},$$

  once we define $V = 4XY^2$.

  $$y_3 = m(x - x_3) - y = \frac{W}{2YZ}\left(\frac{X}{Z^2} - \frac{W^2 - 2V}{4Y^2Z^2}\right) - \frac{Y}{Z^3} =$$

  $$= \frac{W(4XY^2 - [W^2 - 2V])}{8Y^3Z^3} - \frac{Y}{Z^3} = \frac{W(4XY^2 - [W^2 - 2V]) - 8Y^4}{8Y^3Z^3} =$$

  $$= \frac{W(V - [W^2 - 2V]) - 8Y^4}{8Y^3Z^3}.$$

  Finally we can write:

  $$X_3 = W^2 - 2V,\ Y_3 = W(V - X_3) - 8Y^4,\ Z_3 = 2YZ,$$

  where

  $$W = 3X^2 + aZ^4,\ V = 4XY^2.$$

- Once again we can consider jointly the separate cases $P_1 = P_2 \wedge y = 0$ and $P_1 = -P_2$: indeed, by definition, we have $P_1 + P_2 = \infty$.

Looking at the formulas we can see that no inversion is involved in the calculation. Moreover, a further speed up is possible when doubling if we set $a = -3^2$: we have $W = 3X^2 - 3Z^4 = 3(X^2 - Z^4) = 3(X + Z^2)(X - Z^2)$, which can be computed via one squaring and one multiplication rather than via three squarings.

We implemented this approach in the library that can be found at `https://github.com/dginst/BitcoinBlockchainTechnology` to improve the efficiency of the curve's operations. The speed up checks have been performed considering the curves secp192k1, secp192r1, secp224k1, secp224r1, secp256k1, secp256r1, secp384r1 and secp521r1 as described in [2], resulting in a scalar multiplication that is from six to seven times faster in all the cases.

[2]This is the reason why some standardized curves make this choice for the parameter a .

3.1.2 Elliptic curve domain parameters

In order to rely on ECC, the parties involved in a scheme need to agree on a set of parameters called elliptic curve domain parameters. A typical choice is to rely on standardized curves (e.g. those defined in [2]); however, it is still possible to choose other parameters, but this has to be done carefully, as we will explain in this section. We start defining EC parameters, following closely [1].

Elliptic curve parameters are defined as a sextuple $T = (p, a, b, G, n, h)$, where:

- The integer p specifies the prime finite field $\mathbb{F}_p$;
- The two elements $a, b \in \mathbb{F}_p$ specifies the elliptic curve $E(\mathbb{F}_p)$ through the Weierstrass equation;
- G is a point on $E(\mathbb{F}_p)$;
- n is the order of G, i.e. the number of elements of the cyclic subgroup generated by G (that can coincide with the whole $E(\mathbb{F}_p)$);
- $h = \frac{\#E(\mathbb{F}_p)}{n}$ is an integer[3] called the cofactor.

There are different ways in which we can choose these parameters. For example, they can be generated at random or not. Random generation is a conservative choice, since it offers a probabilistic guarantee against future special purpose attacks. In this case it could be better to use verifiable random parameters, meaning that (a, b) and/or G are obtained as output from a secure hash function[4], applied to some seed S: in this way we have the assurance that the parameters cannot be predetermined in order to expose the curve to some special attack. When choosing verifiable random parameters, the sextuple T should be accompanied by the seed value S to allow parameters' validation. However, other choices may be more efficient. This is why a good compromise can be to rely on standardized curves: this approach has the advantage to underpin interoperability.
Now that we explained what they are and how they can be chosen, we will show some constraints and the validation process for the sextuple T.

Parameters' selection:

[3]We can deduce that h is an integer directly from Lagrange's theorem.

[4]An hash function in general is a function that takes as input strings of arbitrary length and outputs in a fixed length sequence of bytes. For a discussion on the security properties of an hash function we refer to the appendix of [1].

1. Select the approximate security level in bits, t^5;
2. Choose a prime p such that $\lceil log_2 p \rceil = 2t$ if $80 < t < 256$, such that $\lceil log_2 p \rceil = 521$ if $t = 256$ and such that $\lceil log_2 p \rceil = 192$ if $t = 80$;
3. Select a and b in $\mathbb{F}_p$ such that:
   - $4a^3 + 27b^2 \neq 0 \pmod{p}$;
   - $\#E(\mathbb{F}_p) \neq p$;
4. Select $G \in E(\mathbb{F}_p)$ such that:
   - $p^B \neq 1 \pmod{n}$, $\forall 1 \leq B < 100$;
   - $h \leq 2^{\frac{t}{8}}$;
   - $n - 1$ and $n + 1$ should each have a large prime factor r, which is large in the sense that $log_n r > \frac{19}{20}$.

These requirements may look strange, but are all needed in order to avoid special attacks. In particular, the two requirements at point 3 are needed in order for the curve to be non singular and to avoid Smart's attack[6], respectively.
The three requirements at point 4 instead are needed to ensure that the curve is robust against MOV and Cheon's attack, and to ensure that the order of G is sufficiently high.
Notice however that the choice of the parameters does not secure against all possible attacks. Indeed other kind of attack may be possible:

- Side channel attacks (timing and power analysis): this attacks exploit the differences in the point addition and doubling operations, that lead to different timings. Possible solutions are a change of coordinates or the use of Edwards curves, a special family of elliptic curves for which addition and doubling can be done with the same operation;

- Cryptographic experts have expressed concerns that the NSA has inserted a backdoor into at least one elliptic curve-based pseudo random generator.

[5]The security level should be chosen such that solving the ECDLP requires 2^t curve's operations. The SEC standard that we are following restricts to $t \in \{80, 112, 128, 192, 256\}$.

[6]The points on the curve are mapped to the elements of the additive group of $\mathbb{F}_p$, so that the discrete logarithm problem is solvable in polynomial time through the **Extended Euclidean Algorithm**.

Parameters' validation: Given $T = (p, a, b, G, n, h)$ and t, the parameters are deemed valid if:

1. p is a prime such that $\lceil log_2 p \rceil = 2t$ if $80 < t < 256$, or such that $\lceil log_2 p \rceil = 521$ if $t = 256$, or such that $\lceil log_2 p \rceil = 192$ if $t = 80$;
2. a, b, x_G and y_G are integers in $[0, ..., p-1]$;
3. $4a^3 + 27b^2 \neq 0 \pmod{p}$;
4. $y_G^2 = x_G^3 + ax_G + b \pmod{p}$;
5. n is a prime number;
6. $h \leq 2^{\frac{t}{8}}$ and $h = \lfloor (\sqrt{p} + 1)^2 / n \rfloor$;
7. $nG = \infty$;
8. $p^B \neq 1 \pmod{n}$, $\forall 1 \leq B < 100$ and $n \neq p$.

If the parameters are verifiably random, it should also be checked that (a, b) and/or G have been correctly derived from the seed S.
These checks are needed to verify that: the curve has the required difficulty level, it is defined over $\mathbb{F}_p$, it is non singular, G is a point of the curve, the DL is difficult, h is effectively the cofactor of G, G has order n and neither MOV nor Smart's attacks are possible.

3.1.3 Elliptic curve key pairs

In this section we would like to briefly analyse the core of the public key cryptography based on elliptic curves: the concept of elliptic curve key pairs. Shortly, public key cryptography is a cryptographic system that relies on pairs of keys, a public and a private key, with different roles. The asymmetry of the keys is the reason behind the fact that public key cryptography is sometimes called also asymmetric cryptography. We talk about asymmetry since public keys can be published, while private keys must be kept secret, as the names suggest. This accomplishes two functions: authentication, since everybody with the public key can verify that the holder of the paired private key sent a signed message, and encryption, where the public key is used for encryption, but only the owner of the private counterpart can decrypt.

Given some elliptic curve domain parameters $T = (p, a, b, G, n, h)$, an elliptic curve key pair $\{q, Q\}$ associated with T consists of an elliptic curve secret key

q which is an integer in $[1, n-1]$ and an elliptic curve public key $Q = (x_Q, y_Q)$, which is the point $Q = qG$.
The choice of the colours is intended to make clear what is secret and what can be made public.
To prevent some attacks (e.g. small subgroup attacks) it is desirable to validate the public key. This can be achieved following these steps:

1. Check that $Q \neq \infty$;
2. Check that x_Q and y_Q are integers in $[0, p-1]$ and that $y_G^2 = x_G^3 + ax_G + b \pmod{p}$;
3. Check that $nQ = \infty$.

Through this algorithm we are checking that Q is effectively a point on the curve different from the point at infinity and that it belongs to the cyclic subgroups generated by G. Indeed, if the relation $Q = qG$ holds we have that $nQ = n(qG) = q(nG) = q\infty = \infty$. If this would not be the case, the third check would fail.

3.1.4 The Bitcoin curve: secp256k1

Since this work is mainly focused on applications that can be deployed in Bitcoin, we would like to present the elliptic curve over a finite field used there. This could also suggest the shape of the curves used in practice.

The curve is named secp256k1: the naming is not casual, and its analysis could help a better understanding of the curve's properties. It begins with **sec** to denote "Standards for Efficient Cryptography", the documentation from which it is taken; then it follows a **p**, denoting the use of parameters over a prime field $\mathbb{F}_p$, in contrast with the so called binary fields $\mathbb{F}_{2^m}$, denoted by the letter **t**; the **p** is followed by a number, 256, denoting the length in bits of the field size p, that suggests the difficulty of solving the DL on the curve; then in comes a **k** to denote parameters associated with a Koblitz curve[7], to be distinguished from an **r**, that would denote the use of verifiably random parameters; at last we find the sequence number **1**, meaning that this curve is the first, actually the unique, with all these characteristics.
We have already discussed the benefits of using a random curve; however, secp256k1 was constructed in a special non random way to ensure efficient computations.
Here follows the sextuple T defining the secp256k1 curve:

[7]The name Koblitz curve used here has the same meaning as in [2].

- The finite field $\mathbb{F}_p$ is defined by the pseudo-Mersenne prime number:

  $p =$ FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFF FFFFFFFFE FFFFFFC2F =

  $= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1.$

- The defining equation E: $y^2 = x^3 + ax + b$ is determined by:

  $a =$ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 =

  $= 0;$

  $b =$ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007 =

  $= 7.$

  Hence E: $y^2 = x^3 + 7$.

- The point G in compressed form is[8]:

  $G =$ 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,

  and in uncompressed form:

  $G =$ 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.

[8]The starting bytes 0x02 and 0x04 means exactly that the first expression of G is in compressed form, while the second one in uncompressed. For what concerns the compressed form we need to have informations about the y coordinate. 0x02 is used when it is even, while 0x03 is used when it is odd: if y is a square root in $\mathbb{F}_p$, we know that also $-y \pmod{p} = p - y$ is a square root. Since p is an odd prime, we are sure that one of the roots is odd while the other is even.

- Finally, the order n of G and the cofactor are:

  $n =$ FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6

  AF48A03B BFD25E8C D0364141,

  $h = 01$

3.2 The discrete logarithm problem

In this section we will deal more in depth with the obscure concept of discrete logarithm problem.
In general the DLP is defined over a group $\mathbb{G}$. For the moment write $\mathbb{G}$ in multiplicative notation and consider $x, y \in \mathbb{G}$ such that y is in the subgroup generated by x. The DLP is the problem of determining an integer $k \geq 1$ such that $x^k = y$. This explain the name, since it would mean to find $k = log_x y$.
Typically, the cryptosystems whose security is based on the ECC, depend on the difficulty of solving the DLP defined over the EC (ECDLP). One way of attacking the DLP is simple brute force: try all possible values of k until one works. This is impractical when the answer k is an integer of several hundred digits, which is a typical size used in cryptography (indeed this approach is fully exponential in the number of bits representing k[9]). Therefore, better techniques are needed.
More formally, we can state the DLP through the following definition.

Definition 3.2.1. *Let $\mathbb{G}$ be a cyclic group of order n and let g be a generator. An algorithm $\mathcal{A}$ is said to (t, ϵ)-solve the DLP in $\mathbb{G}$ if on input a random $h \in \mathbb{G}$, it runs in time at most t and returns $k \in \{0, ..., n-1\}$ such that $h = g^x$ with probability at least ϵ.*

The most general algorithms that works in any group has a time upper bound of $O(|\mathbb{G}|^{\frac{1}{2}})$. But it turns out that the DLP is significantly easier in some groups than it is in others. In order of difficulty:

[9]Assume for example to work with secp256k1, the Bitcoin curve. This means that $p \simeq 2^{256}$, and the same holds true for n. By brute force we try all possible values of k, that ranges from 1 to $n-1$. Trying all possible k would lead to a number of steps of the order $O(2^{256})$, exponential in the number of bits representing the order of the group.

1. The additive group of $\mathbb{F}_q$: the problem here can be stated as finding k such that $kx = y$, i.e. we only need to find the inverse of x, and we have already stated that this can be done efficiently through the extended euclidean algorithm (in particular this requires $O(log_2(q))$ elementary operations, meaning that the problem can be solved in polynomial time);

2. The multiplicative group $\mathbb{F}_q^{\times}$: the problem here is finding k such that $x^k = y$. It can be shown that there exist some algorithms that work in sub-exponential time (the so called index calculus methods);

3. Elliptic curves over finite fields: the fastest known procedure to solve the ECDLP on general curves are collision algorithms. Those algorithms are fully exponential and this is the primary attraction for using elliptic curves in cryptography.

The last point is linked to the primary benefit of ECC, that is a smaller key size: since the problem is harder, in order to have the same level of security we can rely on smaller keys, reducing storage and transmission requirements.

The two most general algorithms are known as **Baby Step, Gian Step Algorithm** and as the **Pollard's ρ Method** with its variant[10]: as explained above both runs in time $O(|\mathbb{G}|^{\frac{1}{2}})$, but the first is exponential also in the memory required to run, making it absolutely impractical. But let's focus on the run time of the algorithms on the Bitcoin's curve: assuming that the scalar multiplication requires constant time on the order of microseconds, solving the ECDLP would require time proportional to $10^{-3} * 2^{128} \simeq 10^{35}$ seconds or around 10^{28} years. It is easy to see that even relying on optimizations, super computers or parallelization is computationally infeasible to solve the ECDLP.

[10]For a detailed but still easily understandable presentation of the algorithms we refer to [6].

Chapter 4

Digital signature schemes

In this chapter we will present the general idea behind digital signatures: what they are and why they are used. Then we will discuss the Elliptic Curve Digital Signature Algorithm (ECDSA), the one actually used in Bitcoin. After having delved into its problems, we will present the adaptation of the Schnorr signature to elliptic curve cryptography (there is currently a discussion about its adoption in Bitcoin). The comparison will turn out to be pitiless.
For a detailed explanation of how signatures are actually used in Bitcoin we refer to Appendix A.

To understand what a digital signature scheme is, we start considering the situation in which Alice wants to sign a document that she aims sending to Bob, who wants assurance that the document comes from Alice and has not been tampered with. As for real life signatures, the aim of a proper digital signature scheme should be to prove the authenticity of a message or document, in the digital realm. Thus, we can list some properties that a proper scheme should provide:

- Authentication: gives a recipient reasons to believe that the message was created by a known sender;
- Non repudiation: the sender cannot deny having sent the message;
- Integrity: ensures that the message has not been altered.

A naive approach to this problem would be to digitalize Alice's signature: but in this case it would be too easy for an eavesdropper to simply copy her signature and append it to another document.
In order to achieve the listed properties, as first thing we should tie the

signature to the document, so that it could not be used again: this would also solve the problem of integrity. If the message changes, the signature is not valid anymore. Moreover, it should be possible for someone to verify that the signature is valid, and it should be possible to show that Alice must have been the person who signed the document: this would ensure the authentication and non repudiation properties.
In cryptography usually some security models are introduced for the so called unforgeability, the property that prevents adversaries from being able to forge a signature that seems to come from Alice. In particular we consider the security model known as existentially unforgeability under chosen message attack, where forgery means winning at the following game:

1. The signer (Alice) gives her public key to the adversary (also known as forger or attacker);

2. The adversary sends messages m_i to the challenger (Alice) and receives valid signature σ on the message m_i for the given public key. He may do this as many times as he likes;

3. The adversary produces a message $m \neq m_i$, $\forall i$ (i.e. the message has not been queried before) along with a valid signature on it.

In this model, unforgeability means that no computationally bounded adversary is able to forge a signature except with negligible probability: in this setting a forgery consists in a signature for a message/public key pair never queried. This hypothesis can be relaxed, excluding from the winning conditions only the triplets message/public key/signature obtained from a query, leading to the concept of strong unforgeability under chosen message attack (SUF-CMA).

Now that we have presented the security model, we can look more formally at the scheme: a signature scheme is a triplet of algorithms $(KeyGen, Sign, Ver)$, where the first two are randomized algorithms, while the third one is deterministic. To sign a document or message, the signers proceeds as follows:

- He runs the key generation algorithm with no inputs[1] to generate a key pair: $(q, Q) = KeyGen()$;

- He runs the signing algorithm on message m and private key q, resulting in a signature σ: $\sigma = Sign(m, q)$;

[1]Some authors consider as input a security parameter, but we take it as given.

- A verifier can easily check the validity of the signature running the verification algorithm as follows: $Ver(\sigma, m, Q) \in \{0, 1\}$. If $Ver(\sigma, m, Q) = 1$ then the signature is valid, otherwise it is not.

A proper signature scheme should satisfy the following consistency equation: $Ver(Sign(m, q), m, Q) = 1, \ \forall(q, Q)$ returned from the key generation algorithm.
A possibility to construct a signature scheme relies on the difficulty of discrete logarithms on elliptic curves. In the next sections we will analyse deeply merits and flaws of two schemes constructed on this hypothesis.

4.1 Elliptic curve digital signature algorithm (ECDSA)

Assume that Alice wants to sign a message M and send it to Bob. The first thing to do for the parties involved is to agree on an elliptic curve determined by the set of parameters T and on a cryptographically secure hash function. This function is used to get a digest from M: indeed, a digital signature can only sign small amounts of data. Applying a proper hash function to a message produces a digest which is small enough and that can act as digital fingerprint if the output space is large enough. Now Alice chooses a secret integer $q_A \in [1, ..., n-1]$ (the private key) and computes her public key $Q_A = q_A G$, that sends to Bob who has to validate it (Alice shouldn't be able to repudiate the signature due to the use of an invalid public key).
The public key can be made public thanks to the difficulty of the ECDLP; moreover, the fact that it is public allows anyone to verify Alice's signature, constituting the non repudiation property.
Now that we have talked about the setup procedure we can give a closer look to the signing and verification algorithms.

It is necessary for k not only to be secret, but to be selected differently each time: otherwise the secret key would be recoverable. Indeed, let's consider two signatures on different messages (r, s) and (r, s') derived using the same k. After having converted r, s and s' to integers, we have:

$$s = k^{-1}(z + rq_A) \pmod{n}, \ s' = k^{-1}(z' + rq_A) \pmod{n} \implies$$

$$\implies s - s' = k^{-1}(z - z') \pmod{n} \implies k = (z - z')(s - s')^{-1} \pmod{n}.$$

Algorithm 4.1 ECDSA: signing algorithm

```
1: procedure ECDSA_SIG($M,\ q$)
2:     $m \leftarrow \mathrm{hash}(\mathrm{bytes}(M))$
3:     Let $z$ be the ${L_n}^2$ leftmost bits of $m$. $z \leftarrow \mathrm{int}(z)$
4:     $k \overset{\$}{\leftarrow} [1, ..., n-1]$
5:     $K = kG = (x_K, y_K)$
6:     $r = x_K \pmod{n}$. If $r = 0$ go back to step 4
7:     $s = k^{-1}(z + rq_A) \pmod{n}$. If $s = 0$ go back to step 4
8:     return $(\mathrm{bytes}(r), \mathrm{bytes}(s))$
9: end procedure
```

Algorithm 4.2 ECDSA: verification algorithm

```
1: procedure ECDSA_VER($(r, s), M, Q$ )
2:     $r \leftarrow \mathrm{int}(r)$, $s \leftarrow \mathrm{int}(s)$
3:     if $r \notin [1, ..., n-1]$ or $s \notin [1, ..., n-1]$ then
4:         return False
5:     end if
6:     $m \leftarrow \mathrm{hash}(\mathrm{bytes}(M))$
7:     Let $z$ be the $L_n$ leftmost bits of $m$. $z \leftarrow \mathrm{int}(z)$
8:     $u_1 \leftarrow zs^{-1} \pmod{n}$, $u_2 \leftarrow zrs^{-1} \pmod{n}$
9:     $K \leftarrow u_1 G + u_2 Q_A$
10:    if $K = \infty$ then
11:        return False
12:    end if
13:    return $r = x_K$ (mod $n$)
14: end procedure
```

From just two signatures we were able to recover the common k and now we can derive the private key from the definition of s or from the definition of s':

$$q_A = r^{-1}(ks - z) \pmod n.$$

To ensure that k is unique for each message one may bypass random number generation completely, since it is one of the main sources of problems in cryptosystems, and generate deterministic signatures by deriving k from both the message and the private key as: $k = \text{hash}(q \mid\mid m)$.
Moreover there are some ways to speed up the verification procedure: through the Shamirs trick, the sum of two scalar multiplication can be calculated faster than the two isolated scalar multiplication at point nine. Finally, we can speed up the verification procedure further, by making K efficiently recoverable from r. In this case one may verify that $sR = zG + rQ_A$. This can be done by including the point R itself in the signature, either in compressed (x_R coordinate and a byte telling the sign of y_R, 0x02 or 0x03 if is is even or odd, respectively) or uncompressed form.

Let's give a look now at the proof of the verification algorithm correctness.
Proof of correctness: We have to prove that $kG = K = u_1G + u_2Q_A$.

- From the definition of Q_A: $u_1G + u_2Q_A = u_1G + u_2q_AG$;
- From the definition of u_1 and u_2: $u_1G + u_2Q_A = s^{-1}(z + rq_A)G$;
- From the definition of s: $u_1G + u_2Q_A = (k^{-1}(z + rq_A))^{-1}(z + rq_A)G = k(z + rq_A)^{-1}(z + rq_A)G = kG$.

$\square$

At the beginning of the chapter we have said that the comparison between ECDSA and Schnorr will be pitiless, so it is clear that ECDSA is far from being perfect. Let's discuss some of its problems:

- Malleability: the signature (r, s) may be replaced with $(r, -s \pmod n)$, because this is an equivalent signature. This means that a third party, without access to the private key can alter an existing valid signature for a given public key and message into another signature that is valid for the same key and message. However this is not regarded as a forgery on the scheme since the message is the original one, but this possibility prevented the deployment of layer two solutions for the Bitcoin scalability problem (e.g. Lightning network). However in Bitcoin this problem has been solved through the soft fork SegWit.

Augmenting the verification equation with the check $s \leq (n-1)/2$ we can make ECDSA strongly unforgeable (which prevents the malleability problem);

- The signing operation needs the calculation of the modular inverse of k and as we have seen it is a slow operation. Moreover this computation can be done only if n is a prime number. Indeed we have a theorem stating that $x \in \mathbb{Z}_n$ is invertible if and only if the greatest common divisor of x and n is one[3]. This obviously holds true for every $x \in \mathbb{Z}_n$ if and only if n is a prime number. However this is not at all a problem, but just a remark: n has to be a huge prime number for the ECDLP to be hard;

- In order for ECDSA to be secure, it can be shown that we need a small cofactor h. This is due to the presence of attacks on the conversion function used in the sixth step of the signing operation, the one that entails taking the x coordinate of K and reducing it modulo n. It is told in [1] that this function has to be almost bijective for ECDSA to be secure, which means that an attacker cannot find an r for which a random K has non negligible probability of satisfying $r = x_K \pmod{n}$. Indeed we have that the integers $x_K = r + jn$ for $j \in \{0, 1, 2, ..., h\}$, if $h = 1$ or 2 as recommended in [2], are linked usually to one valid candidate x coordinate.

4.1.1 ECDSA: multi-signature

Multi-signature schemes allow a group of users to cooperate (interactively or not) to sign a single message or document, usually producing a joint signature that is more compact than a collection of distinct signatures from all users. Verification usually requires the message m and the set of public keys of the signers.
Before we analyse ECDSA multi-signature, let's see formally what these schemes are.

A multi-signature algorithm is a triplet of algorithms $(KeyGen, Sign, Ver)$. To sign jointly a document or message m, each participant i of the scheme should proceed as follows:

[3]`https://www.coursera.org/learn/crypto/lecture/2YWK8/notation`, from minute 7.39.

- She generates a public key pair through the key generation algorithm: $\{q_i, Q_i\} = KeyGen()$;
- She sends her public key to all the other participants, so that every user can gather the same multiset L of public keys: $L = \{Q_1, Q_2, ..., Q_n\}$;
- She runs the signing algorithm on message m, secret key q_i and multiset of public key L: $\sigma = Sign(m, q_i, L)$;
- A verifier can check the validity of the signature through the verification algorithm: $Ver(\sigma, m, L) \in \{0, 1\}$. If it returns 1 then the signature is valid, otherwise it is not.

A proper multi-signature scheme should output to every signer the same signature σ that satisfies the following consistency equation:

$$Ver(Sign(m, q_i, L), m, L) = 1, \ \forall q_i \text{ s.t. } q_i G = Q_i \in L.$$

It is easy, given a signature scheme, to extend it to the multi-user setting in a naive way: each signer signs the message with its own private key, the final signature being the concatenation of the partial signatures. This is the approach used today in Bitcoin with ECDSA: the problem is that the signature size increase linearly in the number of participants. Ideally, the size of the signature should be independent of the number of participants, leading to higher fees for the users and a bloat of the blockchain size that affects every participant in the network.
ECDSA multi-signatures requires multiple public keys in order to validate multiple signatures, i.e. despite the name, multi-signature in Bitcoin is just a tuple of distinct users' signatures.
In literature, the name multi-signature is associated to schemes that, given n participants, require the collaboration of everybody to produce a valid signature. In Bitcoin this notion is extended to the so called threshold schemes: if before we talked about n-of-n schemes, these can be though of as m-of-n, where at least $m <= n$ participants have to collaborate to sign.

4.1.2 ECDSA: public key recovery

Given an ECDSA signature (r, s) and EC domain parameters T, it is generally possible to determine the public key Q associated to the private key used to generate the signature. This is useful in bandwidth constrained environments, when transmission of public keys cannot be afforded. Potentially,

Algorithm 4.3 ECDSA: public key recovery

```
1: procedure ECDSA_RECOVERY($(r, s), M$)
2:     $keys \leftarrow \{\}$
3:     $r \leftarrow \text{int}(r),\ s \leftarrow \text{int}(s)$
4:     for $j \leftarrow 0, h - 1$ do
5:         $x \leftarrow r + jn$
6:         $K \leftarrow \text{EC\_point}(x)$. If $K =$ False or $nK \neq \infty$ go back to step 3
7:         $m = \text{hash}(\text{bytes}(M))$
8:         Let $z$ be the $L_n$ leftmost bits of $m$. $z \leftarrow \text{int}(z)$
9:         for $k \leftarrow 1, 2$ do
10:            $Q \leftarrow r^{-1}(sK - zG)$
11:            $keys \leftarrow keys + Q$
12:            $K \leftarrow -K$
13:        end for
14:    end for
15:    return $keys$
16: end procedure
```

several candidate public keys can be recovered from a signature (their number is linked to the cofactor). The algorithm is presented in Algorithm 4.3.
In the set *keys* there is the correct key. Authenticity could be checked against a public key in some certificate or directory.
Through public key recovery, the verification of a signature becomes implicit: you first recover Q from (r, s). Then, the signature is deemed valid once the public key Q has been authenticated. This step is necessary since, in general, from every signature we can extract a public key.

4.2 Schnorr signature algorithm

In this section we finally pass to analyse the core of the present work, following closely the recent Bitcoin Improvement Proposal by Pieter Wuille [13] on the standardization of the Schnorr signature[4].
As stated in the BIP, ECDSA is standardized, while the Schnorr signature, due to the presence of a patent in past years, is not. But the first one has some downsides compared to the latter over the same curve:

[4]The BIP proposes a standardization of Schnorr but does not deal with the possible implementation in Bitcoin.

- Security proof: the security of Schnorr signature is proved in the random oracle model[5] assuming the ECDLP is hard. Such a proof does not exist for ECDSA, since its security proof requires the generic group model. The two proofs can be found in [7] and [5], respectively. The provably secure construction offered by Schnorr is important since it could prevent attacks from ECDSA in the future[6];
- Non-malleability: ECDSA, as already showed, is malleable. On the other hand, Schnorr signatures are provably non-malleable[7];
- Linearity: Schnorr verification algorithm is linear in both the terms of the signature, so that multiple parties can collaborate to produce a signature that is valid for the sum of their public keys (native multi-signature construction, discussed later on with its flaws).

Thanks to these characteristic, the introduction of Schnorr in Bitcoin would result in privacy and efficiency improvements. Moreover, thanks to the version system introduced by SegWit, Schnorr can be deployed "easily" through a soft fork. Schnorr signature could moreover enable easier implementation of higher level protocols, such as general payment channels and atomic swaps, for which we refer to a later chapter.

The fact that Schnorr is not standardized, allows us to make design choices in order to implement other features in addition to the native ones:

- Batch validation: the specific formulation of ECDSA signature that is standardized cannot be validated more efficiently in batch compared to individually. Switching to Schnorr offers the opportunity to choose a formulation that allows batch validation. This can be an important feature in Bitcoin since when validating a block we need to verify multiple signatures; moreover, we are interested that all of them are valid: if the verification fails, we do not care about which signature caused the failure. We simply reject the whole block;
- Fixed size: the proposed Schnorr standardization is of fixed size, 64 bytes. This is in contrast with the 70-72 bytes long ECDSA (caused

[5]In the random oracle model (ROM) hash functions are modelled as truly random functions whose outputs are uniformly random and can be computed by querying a public oracle.

[6]This is a major achievement since Schnorr provides better security based on the same hypothesis: we do not need to introduce stronger conditions to improve the security.

[7]Segregated witness (SegWit) solves the known malleability problems of ECDSA, but we do not know if others exist. On the other hand, Schnorr is strongly unforgeable under chosen message attacks (SUF-CMA).

by the DER encoding). This would lead to a reduction of on-chain transactions' size.

Even if we decided to be compliant with the BIP, we remark the fact that when signing with Schnorr there are two possible shape for the signature. EC Schnorr signatures for the message m and public key Q involve a point K and integers e and s which satisfy $e = \text{hash}(K \mid\mid m)$ and $sG = K + eQ$. The two different verification equation depend on whether we decide to include e or K in the signature.

- The signature is (e, s) with $e = \text{hash}(sG - eQ \mid\mid m)$. This choice avoids the difficulty of encoding an EC point in the signature;
- The signature is (K, s) with $sG = K + \text{hash}(K \mid\mid m)Q$. This formulation supports batch validation since there are no elliptic curve operations inside the hashes.

The choice falls on the second option, since it supports batch validation. Batch validation is important also because it would speed up considerably the syncing of new Bitcoin nodes that have to download the entire blockchain and validate every transaction.
However, using the second validation rule directly comes with a downside: it is possible for a third party to convert a signature (K, s) for key Q into a signature $(K, s + a * \text{hash}(K \mid\mid m))$ for key $Q + aG$ and the same message, for any integer a. Indeed, the signature (K, s) is verified checking that $sG = K + \text{hash}(K \mid\mid m)Q$, thus: $(s + a * \text{hash}(K \mid\mid m))G = sG + a * \text{hash}(K \mid\mid m)G = K + \text{hash}(K \mid\mid m)Q + a * \text{hash}(K \mid\mid m)G = K + \text{hash}(K \mid\mid m)(Q + aG)$. This is not a concern for the Bitcoin protocol itself, due to the difficulty of the ECDLP[8]. However it may be better to avoid this problem, since in some cases, such as unhardened key derived using BIP32, if Bob knows Alice's master public key, he would be able to transform Alice's signature under a parent public key into another for any public key derived from the same master public key. For this reason, the BIP suggests to use key prefixed Schnorr, changing the equation to $sG = K + \text{hash}(K \mid\mid Q \mid\mid m)Q$. Because any change in the public key would produce unpredictable changes in the hash, Bob cannot use an existing signature to do anything but verify it. Notice that this choice prevents the possibility for public key derivation, a construction that is typically incompatible with batch validation[9].

[8]An adversary could convert the signature to a public key of his choice only knowing the secret key of the signer: in this way he would be able to compute $Q_{adv} = (q + a)G$, choosing properly the integer a.

[9]`https://bitcoin.stackexchange.com/questions/77234/schnorr-vs-ecdsa`.

Let's give a deeper look at how this formulation can be exploited in case of unhardened derivation using BIP32. Without entering in the details, in BIP32 it is specified how to derive key pairs from a unique master key pair, easing incredibly the backup procedure: the major achievement is that the whole tree of key pairs is recoverable from a special number called seed, that can be safely stored in efficient ways. We label the parent key pair as (q_P, Q_P) and the child key pair as (q_C, Q_C). The scheme for unhardened derivation is represented in Figure 4.1 and works as follows: we use the hash function HMAC, that outputs in the space of 512 bits, fed with the parent public key, the parent chain code (for our purposes the only thing that matters is that it is derived as half of the output of the HMAC function, i.e. it is a 256 bit string) and the child index. The result is divided in two half, an offset and the child chaincode (that will be needed for further derivations). The offset is added to the parent private key (modulo the order of the curve) to obtain the child private key. Thus, denoting the offset as f, we have the following relation: $q_C = f + q_P \pmod{n} \implies Q_C = fG + Q_P$.

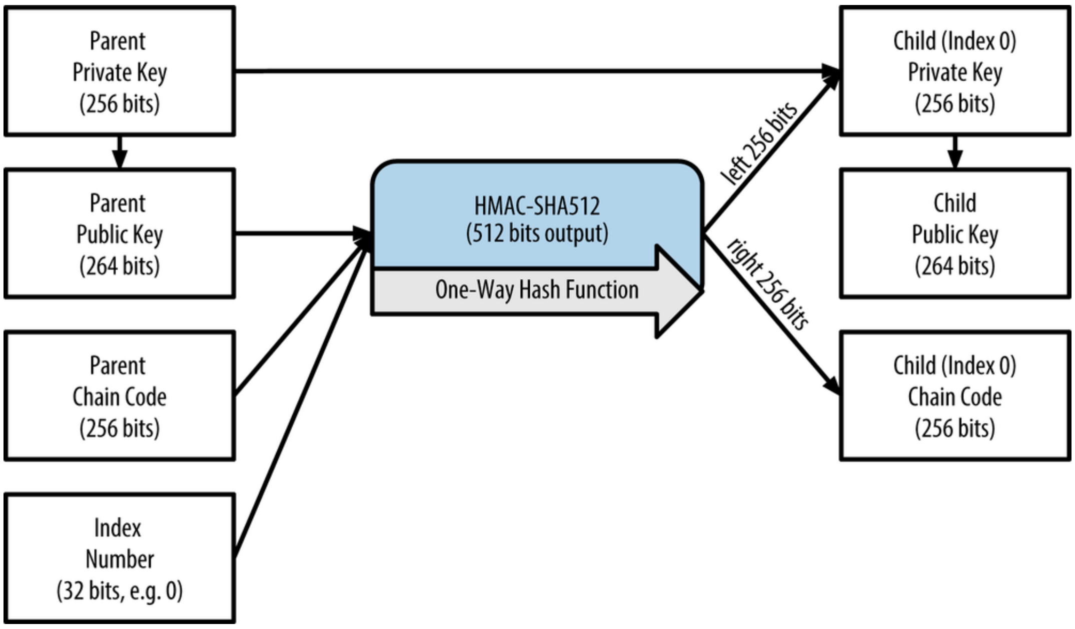

Parent Private Key (256 bits)
Parent Public Key (264 bits)
Parent Chain Code (256 bits)
Index Number (32 bits, e.g. 0)
HMAC-SHA512 (512 bits output)
One-Way Hash Function
left 256 bits
right 256 bits
Child (Index 0) Private Key (256 bits)
Child Public Key (264 bits)
Child (Index 0) Chain Code (256 bits)

Figure 4.1: BIP32's unhardened derivation, source: [3].

Now, imagine an attacker is given a valid signature (K, s) for public key Q_P and message m. Relying on the previous relation, he is able to forge a valid signature for public key Q_C and for the same message m as $(K, s + \text{hash}(K \mid\mid m)f)$. Indeed we have:

$$(s + \text{hash}(K \mid\mid m)f)G = sG + \text{hash}(K \mid\mid m)fG =$$

$$= K + \text{hash}(K \mid\mid m)Q_P + \text{hash}(K \mid\mid m)fG = K + \text{hash}(K \mid\mid m)(Q_P + fG) =$$
$$= K + \text{hash}(K \mid\mid m)Q_C.$$

When substituting sG with $K + \text{hash}(K \mid\mid m)Q_P$ we relied on the fact that (K, s) was a valid signature for public key Q_P and message m. Notice that the attacker only needs a valid signature for the parent public key, the related public key and message, the parent chaincode and the child index.
This is a particular case of the previously presented weakness, that now can be exploited since the relation between the two public keys is known. The hardened derivation also would prevent the forgery since, in place of the parent public key, the parent secret key is given as input to the hash function. To learn the relation between the two public keys, i.e. the offset, the forger would need to know also the parent private key. But this would mean that the attacker has direct access to all the funds secured by the subtree generated by the compromised parent node: the simple forgery is not anymore a concern.

The first problem we have to face is how to encode efficiently the point EC point in the signature, as we chose the K option. Several possibilities exist:

1. Encoding the full point through x and y coordinates, resulting in a 96 bytes signature (32 for each coordinate and another 32 for the integer s);

2. Encoding the x coordinate and use an additional byte to encode whether y is odd or even, like for compressed public keys: this would result in a 65 bytes signature;

3. Encoding only the x coordinate making the point implicit, leading to a 64 bytes signature.

In the BIP, the choice falls on the third option, since compactness is prioritized. However we cannot have ambiguity about the y coordinate, so we need break the symmetry, i.e. we need to make the whole point K implicit in its x coordinate. Again, we have multiple possibilities:

1. Select the y coordinate in the lower half of the plane;

2. Select the y coordinate that is even;

3. Select the y coordinate that is a quadratic residue[10].

As directly stated in the BIP "the third option is slower at signing time but a bit faster to verify, as the quadratic residue of the y coordinate can be computed directly for points represented in Jacobian coordinates. The two other options require a possibly expensive conversion to affine coordinates first. This would even be the case if the sign or oddness were explicitly coded". This is the statement with which the choice of the third option is justified.

Hereinafter hash is the SHA-256 function and the elliptic curve on which everything is worked out is intended to be secp256k1. Now we pass to study the signing and verification algorithms.

Algorithm 4.4 Schnorr: signing algorithm

```
 1: procedure SCHNORR_SIG($M$, $q$)
 2:     $m \leftarrow \text{hash}(\text{bytes}(M))$
 3:     $k \leftarrow \text{int}(\text{hash}(\text{bytes}(q) \mid\mid m)) \pmod n$
 4:     $K \leftarrow kG$
 5:     if jacobi($y_K$) $\neq 1$ then
 6:         $k \leftarrow n - k$
 7:     end if
 8:     $e \leftarrow \text{int}(\text{hash}(\text{bytes}(x_K) \mid\mid \text{bytes}(qG) \mid\mid m) \pmod n$
 9:     $s \leftarrow k + eq \pmod n$
10:     return $(\text{bytes}(x_K), \text{bytes}(s))$
11: end procedure
```

We chose k in such a way that it changes each time the signature algorithm is applied to a different message; indeed even with Schnorr the predictability of this secret value can be used to recover the private key. Given (r_0, s_0) and (r_1, s_1) we have, by definition of s_i:

$$k = s_0 - e_0 q = s_1 - e_1 q \pmod n \implies q = (s_0 - s_1)(e_0 - e_1)^{-1} \pmod n.$$

[10]It can be proved that the product of two numbers is a quadratic residue when either both or none of the factors are quadratic residues. Since we have that the two y coordinates are one the negation of the other, and since -1 is not a quadratic residue when $p = 3 \pmod 4$ (as for secp256k1), we conclude that exactly one of the two roots is a quadratic residue. Notice that this choice for symmetry breaking prevents a real standardization: this formulation of Schnorr signature would not work for curves in which $p = 1 \pmod 4$.

Algorithm 4.5 Schnorr: verification algorithm

```
procedure SCHNORR_VER($(r, s), M, Q$)
    if not isOnCurve($Q$) or $Q = \infty$ then
        return False
    end if
    $r \leftarrow \text{int}(r)$, $s \leftarrow \text{int}(s)$
    if $r \notin [1, ..., p-1]$ or $s \notin [1, ..., n-1]$ then
        return False
    end if
    $m \leftarrow \text{hash}(\text{bytes}(M))$
    $e \leftarrow \text{int}(\text{hash}(\text{bytes}(r) \mid\mid \text{bytes}(Q) \mid\mid m) \pmod n)$
    $K \leftarrow sG - eQ$
    if $K = \infty$ or $\text{jacobi}(y_K) \neq 1$ or $x_K \neq r$ then
        return False
    end if
    return True
end procedure
```

Proof of correctness: Given the signature (r, s), we need to prove that the elliptic curve point K, recoverable from the integer r, equals $sG - eQ$. But by definition of s and by definition of public key, we have:

$$sG - eQ = (k + eq)G - eqG = kG = K.$$

□

Batch verification: it is pretty common for a system to verify a large number of signatures, particularly now with crypto-currency widespread adoption. Thus, faster validation for a batch of signatures traduces in efficiency enhancements.
When validating a signature in the formulation that we chose, the most expensive operations are the two elliptic curve scalar multiplications involved. Using the (K, s) representation, we can verify the signature doing the hash operation first, and then perform elliptic curve operations. This is the key ingredient that allows to introduce batch validation.

We established that a signature (K, s) is valid if $K = sG - \text{hash}(K \mid\mid Q \mid\mid m)Q$. Therefore, a couple of valid signatures will verify:

$$K_0 + K_1 = s_0G - \text{hash}(K_0 \mid\mid Q_0 \mid\mid m_0)Q_0 + s_1G - \text{hash}(K_1 \mid\mid Q_1 \mid\mid m_1)Q_1 \implies$$

$$K_0 + K_1 = (s_0 + s_1)G - \text{hash}(K_0 \mid\mid Q_0 \mid\mid m_0)Q_0 - \text{hash}(K_1 \mid\mid Q_1 \mid\mid m_1)Q_1.$$

We are able to factorize multiplication by G summing all the s values of the signatures: this approach reduces the number of scalar multiplication to one per signature, plus one for the aggregated s. Sadly, this is not secure at all: an attacker can produce a set of signatures that cancel each others. This could be a problem if the signatures are invalid. The attacker would convince the network that they are valid thanks to the batch validation that would succeed. There is also a simpler example that could help clarifying why this construction is not secure: imagine we have a set $(K_i, s_i),\ i \in \{1, ..., n\}$ of valid signatures. Obviously the naive batch validation would succeed. However it would be possible to switch the terms between the signatures, invalidating them all: nonetheless, the batch validation would still succeed. To work around this, we will introduce a random factor per signature. Not knowing the random factor for each of his signatures, the attacker is now unable to have them cancel each others:

$$a_0K_0+a_1K_1 = (a_0s_0+a_1s_1)G-a_0\text{hash}(K_0 \mid\mid Q_0 \mid\mid m_0)Q_0-a_1\text{hash}(K_1 \mid\mid Q_1 \mid\mid m_1)Q_1.$$

Usually it is possible to rely on another trick at this point, namely we can write:

$$a_0K_0 + a_1K_1 = (a_0 - a_1)K_0 + a_1(K_0 + K_1)$$

By keeping a sorted list of points and factors, we could recursively pick the two largest values of a and associated points, do the above mentioned transformation, and reinsert the two new factors and associated points in the list. Every time $a_0 - a_1$ is zero, a point can be removed from the list. Doing so could lead to less scalar multiplication in the end; obviously, this would come at the cost of having way more point additions to do, but the more a values you have, the easier it becomes to find two that cancel each others, and the more multiplications you save. This quickly becomes profitable with the number of signatures.
However, this approach is not feasible in Bitcoin. This is linked to the fact that batch validation is made on a block basis, meaning that we would have some thousands of signatures. But the a values are generated pseudo-randomly in $[1, ..., n-1]$, where n is the order of the curve. This number is around 2^{256}, so that it would be very unlikely to end up with two factors that cancel out.

Algorithm 4.6 presents the batch verification for a number u of signatures $(r_1, s_1), ..., (r_u, s_u)$, associated to the messages $M_1, ..., M_u$ and to public keys $Q_1, ..., Q_u$.
We remark the fact that the algorithm, taken from the BIP, is constructed to work on secp256k1, as clearly shown by the 14^{th} step, where we can see

the defining equation. However it can be easily extended to other curves satisfying the relation $p = 3 \pmod 4$: this property is used in the 15th step, that we try here to clarify. The y coordinates are the square roots of c and can be computed as $y = \pm c^{\frac{p+1}{4}}$, if they exist, due to a lemma by Lagrange[11], applicable when $p \equiv 3 \pmod 4$. Euler's criterion tells us that, given an odd prime p and an integer c coprime to p, we have $c^{\frac{p-1}{2}} = \pm 1 \pmod p$, depending whether c is a quadratic residue or not. The same criterion applied to y yields to $y^{\frac{p-1}{2}} \pmod p = \pm c^{\frac{p+1}{4}\frac{p-1}{2}} \pmod p = \pm (c^{\frac{p-1}{2}})^{\frac{p+1}{4}} \pmod p = \pm 1^{\frac{p+1}{4}} \pmod p = \pm 1 \pmod p$. Therefore, $y = c^{\frac{p+1}{4}} \pmod p$ is a quadratic residue, while $-y \pmod p$ is not: this approach is thus used to comply with the chosen symmetry breaking method.

[11]`https://en.wikipedia.org/wiki/Quadratic_residue#Prime_or_prime_power_modulus`.

Algorithm 4.6 Schnorr: batch verification algorithm

```
1: procedure SCHNORR_BATCH($u, \{(r_1, s_1), ..., (r_u, s_u)\}, \{M_1, ..., M_u\}, \{Q_1, ..., Q_u\}$)
2:     $RHS \leftarrow \infty$
3:     $mult \leftarrow 0$
4:     for $i \leftarrow 1, u$ do
5:         if not isOnCurve($Q_i$) or $Q_i = \infty$ then
6:             return False
7:         end if
8:         $r_i \leftarrow \text{int}(r_i)$, $s_i \leftarrow \text{int}(s_i)$
9:         if $r_i \notin [1, ..., p-1]$ or $s_i \notin [1, ..., n-1]$ then
10:            return False
11:        end if
12:        $m_i \leftarrow \text{hash}(M_i)$
13:        $e_i \leftarrow \text{int}(\text{hash}(\text{bytes}(r_i) \mid\mid \text{bytes}(Q_i) \mid\mid m_i) \pmod n)$
14:        $c \leftarrow r^3 + 7 \pmod p$
15:        $y \leftarrow c^{\frac{p+1}{4}}$
16:        if $y^2 \neq c$ then
17:            return False
18:        end if
19:        $K_i \leftarrow (r_i, y)$
20:        if $i \neq 1$ then
21:            $a_i \xleftarrow{\$} [1, ..., n-1]$
22:        else
23:            $a_i \leftarrow 1$
24:        end if
25:        $mult \leftarrow mult + a_i s_i$
26:        $RHS \leftarrow a_i K_i + (a_i e_i) Q_i$
27:    end for
28:    if $multG \neq RHS$ then
29:        return False
30:    end if
31:    return True
32: end procedure
```

Chapter 5

Schnorr's application

In this chapter we will present some of the applications that Schnorr would make deployable in Bitcoin: mostly they are intended to adapt utilities already present in Bitcoin, such as multi-signatures and threshold signatures.

5.1 Multi-signature: MuSig ($\mu\Sigma$)

Usually Schnorr is presented with an implicit multi-signature scheme: given n users that want to sign a single message m, they can sign it on their own, the final signature being the sum of the so called partial signatures. This signature can then be verified against the sum of the public keys. An appealing property of this scheme is non-interactivity.
Let's study it through an example: Alice and Bob have the key pairs (q_A, Q_A) and (q_B, Q_B), respectively. If both participants are honest, they will proceed as follows: they exchange their public keys, computing the aggregated one $Q = Q_A + Q_B$. Then each one of them proceed as usual, producing K_A and K_B and defining $K = K_A + K_B$, after having exchanged also the public nonces. The signature would then be (K, s), with $s = s_A + s_B = k_A + \text{hash}(K \mid\mid Q \mid\mid m)q_A + k_B + \text{hash}(K \mid\mid Q \mid\mid m)q_B \pmod n = (k_A + k_B) + \text{hash}(K \mid\mid Q \mid\mid m)(q_A + q_B) \pmod n$. Verification would require: $sG = K + \text{hash}(K \mid\mid Q \mid\mid m)Q$, just as usual.
This sounds great, except for the fact that it is a completely insecure scheme: we assumed that both the participants were honest, a deadly hypothesis for every cryptosystem. Imagine that it is Bob that wants to cheat. He could simply says that its public key is $Q'_B = Q_B - Q_A$. Then, if someone sends money to the address associated to $Q = Q_A + Q'_B = Q_A + Q_B - Q_A = Q_B$, clearly Bob can control the funds by himself, being in possess of the associated private key.

This kind of attack is called rogue key attack and is a serious concern for multi-signature schemes: given n participants, a subset of $1 \leq t < n$ dishonest signers use public keys that are functions of the public keys of honest signers, allowing them to forge a signature without the aid of the honest signers for the whole set of public keys. There are certain ways to prevent such an attack: for example by ensuring that the participants own the private keys associated with the alleged public keys (now it is not possible for Bob to cheat, since it would imply breaking the ECDLP), a setting that takes the name of KOSK (knowledge of secret key).

In this section we will present a provably secure multi-signature scheme of the type n-of-n. But before delving into its technicalities, it could be better to give a look into the future: deploying innovations in Bitcoin is a long procedure, due to its decentralized consensus protocol. Since it could take years to take a new feature to it, we should think about properties that would enhance Bitcoin in the long term. Today Bitcoin is missing some important properties in order to be a good method of payment: it is missing both fungibility[1] and privacy. It is missing fungibility due to the fact that it is missing privacy: Bitcoin is pseudonymous, not anonymous, in the sense that an address is not directly linked to a physical person, but every single transaction is on the public ledger open to (possibly) every node in the network. Low privacy means that bitcoins, not being interchangeable, could be treated differently: think about the bitcoins possessed by the creator of Bitcoin, Satoshi Nakamoto, and not moved since the creation of Bitcoin. Obviously they have not the same appeal of newly minted coins.
Fortunately enough, the lack of some properties is not everlasting: for such a reason, when introducing a new feature in Bitcoin we should try to fix these problems. So, we will give now a look to some properties that, in a long term view, a new multi-signature scheme should have.

1. Accountability: this property refers to m-of-n multi-signature schemes (also referred to as threshold schemes) and deals with the fact that for the participants of the scheme should be possible to know who signed and to show to others that they have not;

2. Usability: the ease of use is important. If an interactive scheme requires a huge number of rounds, it won't be used by anyone;

3. Privacy: third parties should learn as little about the policy of the scheme as possible (particular kind of policies could identify your transactions, leading to various problems, like censorship by miners).

[1]Fungibility is the property of a good whose individual units are interchangeable.

5.1.1 MuSig ($\mu\Sigma$)

Up to now we have talked a lot about multi-signature schemes, but we have not seen what they are in detail. With this term we refer to a mathematical scheme that allows to n participants to cooperate to sign a single message m. Verification usually requires the message m and the set of public keys of the signers.
Formally, we have that a multi-signature algorithm is a triplet of algorithms $(KeyGen, Sign, Ver)$. To sign jointly a document or message m, then each participant i of the scheme should proceed as follows:

- She generates a public key pair through the key generation algorithm: $(d_i, P_i) = KeyGen()$;
- She sends her public key to all the other participants, so that every user can gather the same multiset L of public keys: $L = \{P_1, P_2, ..., P_n\}$;
- She runs the signing algorithm on message m, secret key s_i and multiset of public key L: $\sigma = Sign(m, d_i, L)$;
- A verifier can check the validity of the signature through the verification algorithm: $Ver(\sigma, m, L) \in \{0, 1\}$. If the signature scheme returns 1 then the signature is valid, otherwise it is not.

A proper multi-signature scheme should output to every signer the same signature σ that satisfies the following consistency equation: $Ver(Sign(m, d_i, L), m, L) = 1$, for every secret key d_i associated with a public key P_i in L.
It is easy, given a signature scheme, to create the multisig equivalent in a naive way: each signer signs the message with its own private key, the final signature being the concatenation of the partial signatures. This is the approach used today in Bitcoin with ECDSA: the problem is that the signature size increase linearly in the number of participants. Ideally, the size of the signature should be independent of the number of participants.

MuSig is an interactive (meaning that the scheme comprehends different rounds of communication) multi-signature scheme, based on Schnorr signature. MuSig has some very attractive properties, namely:

- The size of the signature is equal to the single user case;
- It is provably secure in the plain public key model[2];

[2]The signers are only required to have a public key: they do not have to prove ownership of it, i.e. knowledge of the associated private key.

These properties, although being appealing, are not original: MuSig shares them with others scheme, in particular with the Bellare-Neven (BN) multisig scheme. The novelty introduced by the authors is that they recovered key aggregation, meaning that to the scheme can be associated a unique joint public key, leading to a verification algorithm that is equal to the single user case: the multi-signature can be verified with respect to a single aggregated public key, leading to greater privacy.
The Bellare-Neven scheme prevents rogue key attacks relying on a particular algorithm to compute the partial signatures, avoiding a trusted setup: $s_i = k_i + c_i d_i,\ c_i = \text{hash}(\langle L\rangle \mid\mid P_i \mid\mid R \mid\mid m)$, where $R = \sum_{i=1}^{n} R_i$, m is the message to be signed, P_i the public key of the i-th participant and $\langle L\rangle$ a unique encoding of the multiset of public keys $L = \{P_1, P_2, ..., P_n\}$. The verification equation then becomes: $sG = R + \sum_{i=1}^{n} c_i P_i,\ s = \sum_{i=1}^{n} s_i$. We can think about MuSig as a variant of BN that recovers key aggregation: indeed, from the verification equation we can see that BN requires the entire multiset of public keys. The setting in which the two schemes are defined is the same: both can be proven to be secure in the plain public key model under the discrete logarithm assumption[3], modeling the hash functions involved as a public random oracle[4].
Security is to be intended in the sense that it is infeasible for an adversary to forge multi-signatures involving at least an honest participant, that is: the adversary is not able to produce on its own a signature valid for the set of public keys containing the one of the honest signer.
We stress the fact that, from the applicative point of view, key aggregation is a fundamental property: if the scheme is usable (few interactive rounds), then we get for free privacy and accountability (MuSig is an n-of-n scheme, so that it is possible to generate a valid signature only if all the participants agree). Indeed, thanks to key aggregation, verifiers will only see an aggregated public key $\tilde{P}$: they wouldn't even know that it is indeed aggregate, since it is indistinguishable from a normal public key. This is also important from the point of view of efficiency: in Bitcoin every single node has the possibility of validating each transaction. This means that verification efficiency and signature size are very important, more than the timing of the signing algorithm. This is why, although there are multisig schemes with fewer interaction rounds, we prefer MuSig: the benefits of key aggregation are improved bandwidth (no need for communication of multiple public keys),

[3]The discrete logarithm assumption requires the DL to be hard on the selected group. This means that if the DL is hard, then the scheme is secure.

[4]In the random oracle model, hash functions are substituted with a black box that responds in random way to every unique query. Moreover, these random oracles are publicly accessible.

privacy (aggregated public key indistinguishable from a normal one) and validation efficiency (as efficient as a normal Schnorr's verification, slightly faster than the ECDSA case).
Moreover if the aggregated public key is not given to the verifier, it is still possible to recover it just from the set of public keys of the participant, without interaction with the signers.

The plain public key setting plays a crucial role when trying to enable multi-signatures across multiple inputs of a Bitcoin transaction. Indeed, MuSig can go beyond what we have seen at input level, where it reduces the signatures number to one per input: it is possible to go further and get a single signature per transaction. Citing directly the original paper, doing so would require a change in the verification procedure of the expandability of UTXO, since the validity of separate inputs is no longer independent, thus the outputs locking scripts cannot be modeled as predicates in this case. We can think of them as functions returning a boolean value and a set of zero or more public keys. The validity of the transaction requires all returned boolean values to be true and a multi-signature of the transaction with L to be the union of the returned keys. The amazing fact is that this can be implemented in a backward compatible way providing an alternative to the signature checking opcode OP_CHECKSIG and related opcodes in the Bitcoin scripting language. Instead of returning the result of an actual ECDSA verification, they always return true, but additionally add the public key with which the verification would have taken place to a transaction-wide multiset of keys. Finally, after all inputs are verified, a multi-signature present in the transaction is verified against that multiset. In case the transaction spends inputs from multiple owners, they will need to collaborate to produce the multi-signature, or choose to only use the original opcodes.
To see why the plain public key model is fundamental think about an attacker that identifies some outputs he want to steal, corresponding to a set $\{P_1, P_2, ..., P_{n-t}\}$ of public keys. Then he could try to identify another set of keys $\{P_{n-t+1}, ..., P_n\}$ such that he can sign for the aggregated public key $\tilde{P}$. He would be able to steal the coins just by sending a small amount of his own moneys to outputs corresponding to the keys he found and finally creating a transaction with inputs the outputs he want to steal and the newly created outputs in his possession: by construction he is able to forge a signature on his own for this transaction.
In the case of multi-signatures across inputs, theft can occur by being able to forge a signature over a set of keys that includes at least one key not controlled by the attacker: this is exactly what the plain public key model considers a win for the attacker.

Now we will see in general the description of MuSig. The scheme is parameterized by the cyclic group $\mathbb{G}$ denoted in additive notation, its order n, a generator of the group G and three hash functions H_{com}, H_{agg} and H_{sign}: $\{0,1\}^* \to \{0,1\}^l$. The bit length of the order n is denoted by l and assumed to be a security parameter.

Key Generation: Each participant of the scheme generates a random private key $d \in \mathbb{Z}_n$ and computes the corresponding public key $P = dG$.

Signing algorithm: We split the signing algorithm accordingly to the interaction rounds of the scheme.

1. Let (d_1, P_1) be the key pair of a specific signer, let m be the message to be signed, let $P_2, P_3, ..., P_n$ be the public keys of the other cosigners and let $\langle L \rangle$ be a unique encoding of the multiset of public keys $L = \{P_1, P_2, ..., P_n\}$: the indices are local references to cosigners. For $i \in \{1, 2, ..., n\}$ the signer computes:

   $$a_i = H_{agg}(\langle L \rangle, P_i).$$

   and then the aggregated public key $\tilde{P} = \sum_{i=1}^{n} a_i P_i$. Then the signer generates a random $k_1 \in \mathbb{Z}_n$, computes $R_1 = k_1 G$, $t_1 = H_{comm}(R_1)$ and sends t_1 to all other cosigners.

2. Upon reception of the commitments $t_2,\ t_3, ...,\ t_n$ from other cosigners, the signer sends R_1.

3. Upon reception of $R_2,\ R_3, ...,\ R_n$ from other cosigners, it checks that $t_i = H_{comm}(R_i),\ \forall i \in \{2, ..., n\}$ and aborts the protocol if this is not the case; otherwise, he computes:

   $$R = \sum_{i=1}^{n} R_i,$$

   $$c = H_{sig}(\tilde{P}, R, m),$$

   $$s_1 = k_1 + c a_1 d_1 \pmod{n}.$$

   Then he sends s_1 to all other cosigners.

4. Finally, upon reception of $s_2, s_3, ..., s_n$ from other cosigners, the signer can compute $s = \sum_{i=1}^{n} s_i \pmod{n}$. The signature is (R, s).

Verification algorithm: Given a multiset of public keys $L = \{P_1, P_2, ..., P_n\}$, a message m and a signature $\sigma = (R, s)$, the verifier computes $a_i = H_{agg}(\langle L \rangle, P_i)$ for $i \in \{1, 2, ..., n\}$, $\tilde{P} = \sum_{i=1}^{n} a_i P_i$ and $c = H_{sig}(\tilde{P}, R, m)$. Then he accepts the signature if $sG = R + \sum_{i=1}^{n} c a_i P_i = R + c\tilde{P}$.

Proof of correctness: We have to prove that $sG = R + c\tilde{P}$.

$$sG = \left(\sum_{i=1}^{n} s_i\right) G = \left(\sum_{i=1}^{n} (k_i + c a_i d_i)\right) G = \sum_{i=1}^{n} (k_i G + c a_i d_i G) =$$
$$= \sum_{i=1}^{n} k_i G + \sum_{i=1}^{n} c a_i d_i G = \sum_{i=1}^{n} R_i + c \sum_{i=1}^{n} a_i P_i = R + c\tilde{P}.$$

□

For the security proof we refer to [5]. Here the verification algorithm is constructed getting as input the multiset of public keys L. To improve both efficiency and privacy it is immediate to modify it in order to take as input the aggregate signature $\tilde{P}$, simply by relying on the standard Schnorr verification algorithm.
We have seen in the single user case that there was the possibility to derandomize the signature algorithm without loss of security, by generating the random nonce k through a deterministic function. This is done since pseudo random generation is one of the major sources of problems in cryptography. But in the multi user setting it is necessary to ensure to use different random values when the other signers change their R values in repeated signing attempts, otherwise secret key recovery would be possible. We show this referring to an example coming from [5]: assume Alice and Bob have key pairs (d_A, P_A) and (d_B, P_B), respectively. They want to jointly produce a signature. Alice generates k_A and sends $R_A = k_A G$ to Bob. In a first attempt, Bob responds with R_B. Alice then computes:

$$R = R_A + R_B,$$

$$c = H_{sig}(\tilde{P}, R, m),$$

$$s_A = k_A + c a_A d_A \pmod{n},$$

and sends s_A to Bob. Bob is trying to cheat on Alice, and decides not to produce a valid s_B, and thus the protocol fails. A new signing attempt takes place, and Alice again sends the same R_A. Bob responds with $R'_B \neq R_B$. Alice compute $c' = H_{sig}(\tilde{P}, R_A R'_B, m)$ and $s'_A = k_A + c' a_A d_A \pmod{n}$ and sends s'_A to Bob. Now Bob is able to derive Alice's private key:

$$s_A - s'_A = (c - c')a_A d_A \pmod{n} \implies d_A = (c - c')^{-1} a_A^{-1}(s_A - s'_A) \pmod{n}.$$

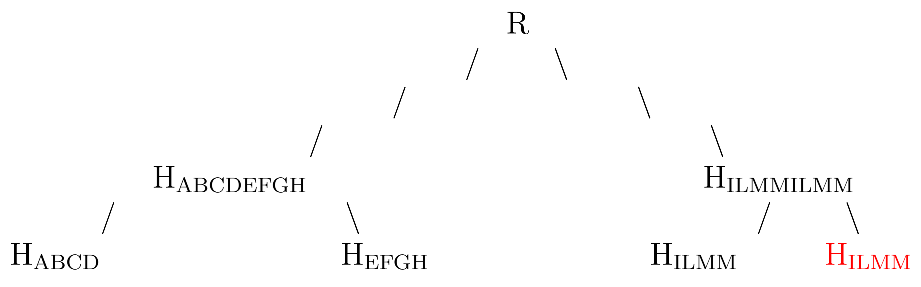
To avoid this problem, each signer must ensure that whenever any R value sent by other cosigners or the message m changes, his k_i changes as well.

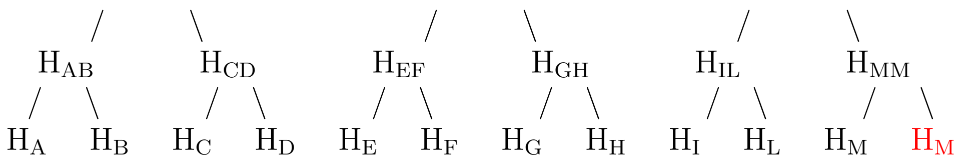
5.1.2 Threshold signatures

With the name threshold signatures we refer to policies of the kind m-of-n, where it is necessary that at least m participants of the scheme decide to collaborate to produce a valid signature. This kind of policy is very popular in Bitcoin, since it is flexible and has many applications: for example, a single user could use such a policy to improve security, storing the keys on different machines, at the same time defeating the risk of loss of some keys.
We will analyse two possibilities to bring threshold signature schemes based on Schnorr to Bitcoin. One is more general and combines Schnorr signatures with Pedersen Secret Sharing, while the other is more Bitcoin oriented since it requires Bitcoin's scripting language and is called signatures tree.

Tree signatures

Tree signatures are based on the concept of Merkle tree, a binary tree able to uniquely identify a possibly huge amount of data with a much smaller identifier, the so called Merkle root, so that it is possible to prove ownership of a piece of data to the tree with a proof that increases logarithmically with the amount of data. In the leaves of the tree are stored the hashes of the data; these hashes are then coupled (if the number of elements at the current level is odd the last one is concatenated with itself and hashed), concatenated and hashed again and this procedure is repeated until there remains a unique hash, the Merkle root of the tree.
It follows an example of Merkle tree with eleven leaves that becomes twelve when duplicating the last one (the duplicates are highlighted in red).

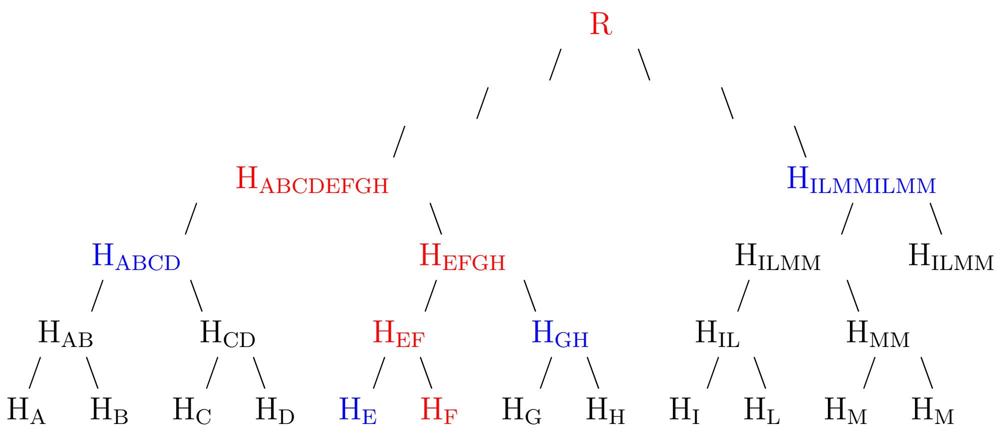




Now we would like to exemplify the proof of ownership of the data associated to the leaf H_F. In order to do it we need what is called a Merkle proof, that, in this specific case, consists of the nodes $\{H_E,\ H_{GH},\ H_{ABCD},\ H_{ILMMILMM}\}$ and of the piece of data F. Given the Merkle proof the validation proceed as follows:

- Take F and compute hash(F) = H_F;
- Take H_E, concatenate it with H_F and compute hash(H_E || H_F) = H_{EF};
- Take H_{GH} from the Merkle proof and compute hash(H_{EF} || H_{GH}) = H_{EFGH};
- Take H_{ABCD} and compute hash(H_{ABCD} || H_{EFGH}) = $H_{ABCDEFGH}$;
- Finally, the Merkle root is compared with hash($H_{ABCDEFGH}$ || $H_{ILMMILMM}$). If the two values coincide we have the assurance that the hash value correspondent to the data F belongs to the tree.

The path of the proof is showed in the following figure, where in blue it is presented the Merkle proof and in red the proof of membership:



In order to implement the procedure a good idea could be to associate to each node a boolean value to assess whether it is a right or left child of the upper level node.

This idea can go further relying on the Bitcoin scripting language: it is possible to associate to each leaf a different redeem script. Then, thanks to the Merkle proof, only the redemption script that is actually used to spend the bitcoins goes on the blockchain, while all the others are hidden. This approach leads to the so called MASTs (Merkleized Abstract Syntax Trees), that could lead a great improvement in privacy. But now let's get back to the combination of tree signatures with Schnorr.

To bring an m-of-n policy through Schnorr we could associate to each leaf of the Merkle tree an m-of-m Schnorr public key (the aggregated key): the number of the leaves would be $\binom{n}{m}$, the number of ways, disregarding the order, in which m signers can be chosen from n participants.
For example, think of a 3-of-4 policy. Denoting the set of signers as {A, B, C, D}, the possibilities to produce a valid signature are the following: {A, B, C}, {A, B, D}, {A, C, D} and {B, C, D}. We would need to compute the four aggregated public keys $\mathrm{P}_{\mathrm{ABC}}$, $\mathrm{P}_{\mathrm{ABD}}$, $\mathrm{P}_{\mathrm{ACD}}$ and $\mathrm{P}_{\mathrm{BCD}}$. Then the four associated hash values become the leaves of a Merkle tree:

$$\begin{tikzcd} & & & \mathrm{R} \arrow[dll, dash] \arrow[drr, dash] & & & \\ & \mathrm{H}_1 \arrow[dl, dash] \arrow[dr, dash] & & & & \mathrm{H}_2 \arrow[dl, dash] \arrow[dr, dash] & \\ \mathrm{H}_{\mathrm{ABC}} & & \mathrm{H}_{\mathrm{ABD}} & & \mathrm{H}_{\mathrm{ACD}} & & \mathrm{H}_{\mathrm{BCD}} \end{tikzcd}$$

The root R would lock the output of a transaction. To spend the associated bitcoins it would be necessary to provide a Merkle proof and a signature valid for the chosen aggregated key. Assuming that the signers are A, B and D, the Merkle proof consists of $\{\mathrm{P}_{\mathrm{ABD}},\ \mathrm{H}_{\mathrm{ABC}},\ \mathrm{H}_2\}$, while the actual redeem script in Bitcoin scripting language would comprehend the four public keys. Although the privacy benefits is obvious (one aggregated public key instead of the four public keys), it may not be obvious that also from the point of view of the efficiency this approach would bring some benefits. This is linked with the fact that the actual implementation of the threshold signature in Bitcoin scales linearly in the number of participants (the public keys of all the participants would be necessary), while the Merkle proof is logarithmic in the number of combinations (the number of leaves). Moreover, while this approach would require only one expensive CHECKSIG operation, the standard Bitcoin approach requires a CHECKSIG operation per signer.
Looking back to the desirable properties of a multi-signature scheme, we

could say that this approach satisfies them all: it is private, usable and accountable, since the participant are able to recognize the signers, having to store the whole tree. However it has some downsides: it is feasible only when the number of leaves is not too big, although this is a problem only for the participants, since the verifiers see only the Merkle proof. Another major problem is linked to the fact that the participants have to pre-compute the keys associated to every combination. This problem could be avoided if, instead of the aggregated key, in the leaves we put the hash associated to the concatenation of the public keys. This means that the unlocking script contains as usual the Merkle proof and a signature, but now the Merkle proof embeds the whole set of public keys used to compute the signature, giving up privacy. In the end it is up to the verifier the computation of the aggregated public key needed for the verification of the signature. Moreover, even if the Merkle proof has always size $log_2\binom{n}{m}$, this method is linear in m for what concerns the unlocking script since there is the need for the set of public keys of all the signers.
Other benefit provided by the tree signatures approach is that it is very generic, since it works for any subset of combinations of keys. The privacy improvement is so remarkable since from the outside the policy is not even recognizable: indeed adding dummies has a very low cost.

Pedersen secret sharing

In this section we will give a look to another construction that allows to build Schnorr based threshold signature schemes. The scheme that will be analysed is provably secure, meaning that it can be shown to be as secure as the Schnorr signature itself. For the security proof we refer to [8]. This scheme is constructed on top of the Pedersen Verifiable Secret Sharing Scheme (VSSS) and Pedersen's multi-party protocol to generate a random shared secret, hence we start the section presenting these two schemes.

8.2.2.1 Verifiable secret sharing scheme Typically, in an m-of-n secret sharing scheme, a trusted dealer distributes a secret s to n players $P_1, ..., P_n$ in such a way that any subgroup of at least m members can recover the secret, while any subgroup of cardinality strictly less than m learns nothing about it. A verifiable secret sharing scheme moreover prevents the dealer from cheating, since each participant can verify that his share of the secret is consistent with the others. The novelty introduced by Pedersen is that his scheme is not interactive in the verification and does not require trust between the parties involved. Still there is a trusted dealer, a figure we would like to get rid of: this will be done in a while. Here follows the VSSS proposed by Pedersen.

Fix an elliptic curve over a prime finite field $E(\mathbb{F}_p)$, characterized by the EC domain parameters $T = (a, b, p, G, n, h)$ and fix another generator H of the same cyclic group generated by G. We require that these two generators are "nothing up my sleeves" (NUMS), meaning that we do not know the discrete logarithm of one with respect to the other, and vice versa (this property is required by the Pedersen commitment, see below).
Assume that the dealer has a secret value $s \in \mathbb{Z}_p$ and a number $s' \in \mathbb{Z}_p$ generated at random. He commits to the couple (s, s') through the so called Pedersen commitment $C_0 = sG+s'H$[5]. The NUMS property is needed to prevent the person who commits from lying about the values he committed to. Assume indeed that the dealer knows the discrete logarithm of H with respect to G: $H = r_H G$. In this case she could write: $C_0 = sG+s'H = (s+s'r_H)G = (s \pm ar_H+s'r_H)G = (s-ar_H)G+(s'r_H+ar_H)G = (s-ar_H)G+(s'+a)H,\ \forall a \in \mathbb{Z}_n$.
After having broadcasted the commitment, the secret s can be shared among $P_1, ..., P_n$ through the following protocol:

The dealer:

1. Chooses a couple of random polynomials of degree $t-1$:

$$f(u) = s + f_1 u + ... + f_{t-1}u^{t-1},\ \ f'(u) = s' + f'_1 u + ... + f'_{t-1}u^{t-1},$$

   where s and s' are the committed values, while $f_i, f'_i \in \mathbb{F}_p$ are randomly chosen for every $i \in \{1, ..., t-1\}$;

2. Computes $(s_i, s'_i) = (f(i), f'(i))$ for $i \in \{1, ..., n\}$;

3. Sends secretly (s_i, s'_i) to $P_i, \forall i \in \{1, ..., n\}$;

4. Broadcasts the values $C_j = f_j G + f'_j H,\ \forall j \in \{1, ..., t-1\}$.

Each participant P_i:

1. Verifies the consistency of its share of the secret as:

$$s_i G + s'_i H = \sum_{j=0}^{t-1} i^j C_j.$$

   If this check fails he broadcasts a compliant against the dealer;

[5]Loosely speaking a commitment scheme is a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden to others, with the ability to reveal the committed value later, but without the possibility of lying about it.

2. For each compliant from a player i, the dealer defends himself by broadcasting the values $(s_i, s'_i) = (f_i, f'_i)$ that satisfies the checking equation at point 1;

3. Aborts the protocol if:
   - The dealer received more than t compliants in step 1;
   - She answered to a compliant in step 2 with values that violates again the checking equation.

Pedersen proved that any coalition of less than t players cannot get any information about the shared secret, provided that the discrete logarithm in $E(\mathbb{F}_p)$ is hard[6]. For the proof we refer to the original paper [*Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing* by Torben Pryds Pedersen]. Although we do not look at the proof, it may still be of interest to check why the verification procedure should succeed:

$$s_iG + s'_iH = f(i)G + f'(i)H = \sum_{j=0}^{t-1} f_j i^j G + \sum_{j=0}^{t-1} f'_j i^j H =$$
$$\sum_{j=0}^{t-1} i^j (f_jG + f'_jH) = \sum_{j=0}^{t-1} i^j C_j.$$

We used the convention that $f_0 = s$ and $f'_0 = s'$. Remembering that C_0 commits to the secret and that the other C_j commits to the polynomials, we have the assurance that the dealer is not cheating. Indeed there is one and only one polynomial of degree at most $t-1$ satisfying $f(i) = s_i$, respectively $f'(i) = s'_i$, for t values of i.
This is also the key property that allows the reconstruction of the secret value from any group $\mathcal{P}$ of t participants. Indeed the members in $\mathcal{P}$ can recover the polynomial f through the Lagrange's interpolation formula, that given a set of t points $(i, s_i = f(i))$ returns the lowest degree polynomial (in this case a $t-1$ degree polynomial) that in each value i assumes the value s_i:

$$f(u) = \sum_{i\in\mathcal{P}} f(i)\omega_i(u), \text{ where } \omega_i(u) = \prod_{j\in\mathcal{P},\ j\neq i} \frac{u-j}{i-j} \pmod p.$$

Since it holds that $s = f(0)$ by definition, the group $\mathcal{P}$ can directly reconstruct the secret as:

$$s = f(0) = \sum_{i\in\mathcal{P}} f(i)\omega_i, \text{ where } \omega_i = \omega_i(0) = \prod_{j\in\mathcal{P},\ j\neq i} \frac{j}{j-i} \pmod p.$$

[6]This is important since we are not adding cryptographic assumptions.

8.2.2.2 Protocol for the generation of a random shared secret For the key generation phase of the signature scheme it is necessary to generate a random shared secret key in a distributed way. Thus we need to get rid of the trusted dealer. This can be done relying on the following protocol:

Each participant P_i:

1. Chooses $r_i, r'_i \in \mathbb{Z}_n$ at random and verifiably shares (r_i, r'_i) acting as the dealer according to the Pedersen's VSSS described above. Let the sharing polynomials of participant i be $f_i(u) = \sum_{j=0}^{t-1} a_{ij}u^j$, $f'_i(u) = \sum_{j=0}^{t-1} a'_{ij}u^j$, where $a_{i0} = r_i$ and $a'_{i0} = r'_i$. The public commitments are $C_{im} = a_{im}G + a'_{im}H$ for $m \in \{0, ..., t-1\}$;

2. Sets $H_0 = \{P_j \mid P_j$ is not detected to be cheating at step $1\}$. The distributed secret value r is equal to $\sum_{i\in H_0} r_i$. Each participant P_i sets his share of the secret at $s_i = \sum_{j\in H_0} f_j(i) \pmod n$ and set the value $s'_i = \sum_{j\in H_0} f'_j(i) \pmod n$.

3. Each player in H_0 broadcasts $Y_i = r_iG$ via Feldman's VSSS:

   (a) Each player P_i in H_0 broadcasts $A_{ik} = a_{ik}G$ for $k \in \{0, ..., t-1\}$;

   (b) Each player P_j verifies the values broadcasted by the other players in H_0: for each $P_i \in H_0$, P_j checks:

   $$f_i(j)G = \sum_{k=0}^{t-1} j^k A_{ik}.$$

   If the check fails for an index i, P_j complaints against P_i broadcasting the values $(f_i(j), f'_i(j)$ that satisfy the checking equation of the Pedersen's VSSS but not the one at point (b);

   (c) For players P_i who received at least one valid complaint, the other players run the reconstruction phase of Pedersen's VSSS to compute r_i, $f_i(u)$ and A_{ik} for $k \in \{0, ..., t-1\}$. All participants in H_0 set $Y_i = r_iG$.

After the execution of the protocol the following equations hold:

$$Y = \sum_{i\in H_0} Y_i = \sum_{i\in H_0} r_iG = rG,$$

$$f(u) = \sum_{i\in H_0} f_i(u) = r + a_1u + ... + a_{t-1}u^{t-1}, \text{ where } a_i = \sum_{j\in H_0} a_{ji},$$

$$f(i) = s_i.$$

We introduce the following notation:

$$(s_1, ..., s_n) \xleftrightarrow{\text{(t, n)}} (r|Y, a_iG, H_0),\ i \in \{1, ..., t-1\}.$$

It means that s_j is the share of secret key r belonging to P_j for $j \in H_0$. The values a_iG are the public commitments of the sharing polynomials $f(u)$ and (r, Y) is the key pair that can be reconstructed by any subgroup of H_0 composed of at least t participants.
Before we pass to analyse the actual signature scheme, let's give a look at the checking equation at point (b) in the previous protocol.

$$f_i(j)G = \sum_{k=0}^{t-1} a_{ik} j^k G = \sum_{k=0}^{t-1} j^k A_{ik}.$$

8.2.2.3 Threshold signature scheme Now that we have defined the primitives on which it is built, we can discuss the protocol that implements the t-of-n threshold signature scheme.

Key generation: All n participants have to cooperate to generate a public key Y and a share of the secret key for each participant P_j. This can be done relying on the protocol presented in the previous paragraph named **Protocol for the generation of a random shared secret**. The output of the protocol is:

$$(\alpha_1, ..., \alpha_n) \xleftrightarrow{\text{(t, n)}} (x|Y, b_iG, H_0),\ i \in \{1, ..., t-1\}.$$

The α values denote the secret key share belonging to P_j. They will be used to generate a partial signature for the key pair (x, Y).

Signing algorithm: Let m denote the message to be signed. Suppose that a subset $H_1 \subseteq H_0$ wants to issue a signature. The members of H_1 proceed as follows:

1. If $|H_1| < t$, aborts. Otherwise, the subset H_1 generates a random shared secret following again the protocol presented in the paragraph about the generation of a random shared secret. We denote the output as:
$$(\beta_1, ..., \beta_n) \xleftrightarrow{\text{(t, n)}} (e|V, c_iG, H_2),\ i \in \{1, ..., t-1\}.$$
2. If $|H_2| < t$, aborts. Otherwise, each $P_i \in H_2$ reveals
$$\gamma_i = \beta_i + \text{hash}(V \mid\mid Y \mid\mid m)\alpha_i.$$

3. Each $P_i \in H_2$ verifies that:

$$\gamma_k G = V + \sum_{j=1}^{t-1} c_j k^j G + \text{hash}(V \mid\mid Y \mid\mid m) \left(Y + \sum_{j=1}^{t-1} b_j k^j G \right), \ \forall k \in H_2.$$

   Let $H_3 = \{P_j \mid P_j \text{not detected to be cheating at step 3}\}$.

4. If $|H_3| < t$, aborts. Otherwise each $P_i \in H_3$ selects an arbitrary subset $H_4 \subseteq H_3$ with $|H_4| = t$ and computes σ satisfying $\sigma = e + \text{hash}(V \mid\mid m)x$, where:

$$\sigma = \sum_{j \in H_4} \gamma_j \omega_j, \text{ where } \omega_j = \prod_{h \in H_4,\ h \neq j} \frac{h}{h-j}.$$

   The signature is (V, σ). To verify the signature, the same formula as in Schnorr scheme applies:

$$\sigma G = V + \text{hash}(V \mid\mid Y \mid\mid m)Y.$$

This concludes the presentation of the protocol. Nonetheless there are some formulas that deserves greater attention.

- Checking formula at point 3:

$$\gamma_k G = (\beta_k + \text{hash}(V \mid\mid Y \mid\mid m)\alpha_k)G = \beta_k G + \text{hash}(V \mid\mid Y \mid\mid m)\alpha_k G =$$

$$= \left(e + \sum_{j=1}^{t-1} c_j k^j \right) G + \text{hash}(V \mid\mid Y \mid\mid m) \left(x + \sum_{j=1}^{t-1} b_j k^j \right) G =$$

$$= V + \sum_{j=1}^{t-1} c_j k^j G + \text{hash}(V \mid\mid Y \mid\mid m) \left(Y + \sum_{j=1}^{t-1} b_j k^j G \right).$$

- Formula used to compute σ: we defined $\gamma_i = \beta_i + \text{hash}(V \mid\mid Y \mid\mid m)\alpha_i$, $\forall i \in H_2$. In particular the equation holds for every $i \in H_4$, with $|H_4| = t$. The α and β values are defined to be the pointwise evaluation of the sharing polynomials created during the two iterations of the Pedersen protocol for the generation of a random shared secret, that we denotes by $F_1(u)$ and $F_2(u)$. These polynomials have degree $t-1$, so that we can write:

$$F_3(u) = F_2(u) + \text{hash}(V \mid\mid Y \mid\mid m)F_1(u) \implies$$

$$\implies F_3(0) = F_2(0) + \text{hash}(V \mid\mid Y \mid\mid m)F_1(0) =$$
$$= e + \text{hash}(V \mid\mid Y \mid\mid m)x = \sigma.$$

At this point we can apply the Lagrange's interpolation formula, since we know that $F_3(u)$ satisfies by construction $F_3(i) = \gamma_i$:

$$F_3(u) = \sum_{j \in H_4} \gamma_j \omega_j(u), \text{ where } \omega_j(u) = \prod_{h \in H_4,\ h \neq j} \frac{u-h}{j-h}.$$

Thus, σ can be directly computed as:

$$\sigma = F_3(0) = \sum_{j \in H_4} \gamma_j \omega_j, \text{ where } \omega_j = \prod_{h \in H_4,\ h \neq j} \frac{h}{h-j}.$$

- Verification formula:

Moreover the scheme is robust, meaning that a corrupt signer who does not follow the protocol will be detected. The validity of the γ_i is tested at step 3.

Appendix A

Signatures in Bitcoin

In this appendix we will analyse the role of ECDSA in Bitcoin, starting with a general overview of the protocol, focusing our attention on transactions and blocks. Then we will give a low level description of Bitcoin scripting language, some spending policies and their relation with signatures. More complex subjects, such as SegWit, being out of the scope of the present work will not be presented. We refer the interested reader to the dedicated literature (starting from [9], [11], [8] and [10]). An easily understandable and complete introduction to Bitcoin can be found in [3].

In its basic form, a transaction is a transfer of bitcoins from a participant of the Bitcoin network to another user. Multi-signature constructions can be easily understood once the basic transaction is mastered. However is important to point out that these are not at all the unique possibilities: the presence of a scripting language, despite its simplicity, allows very flexible and complex constructions. Every transaction is broadcasted to each node in the network and each one, separately, validate it. Obviously one node propagate a transaction only if it is deemed valid: here is when signatures come in. Typically they are used to prove to the other users in the network ownership of the funds we are attempting to spend. Groups of transactions are collected into a data structure called block, by special nodes in the network called miners. The action of building blocks is very expensive from a computational point of view (requiring much energy this traduces in high costs): for this reason, miners are rewarded for their work with the issuance of new bitcoins. These blocks are chained one another including in the header of a block the hash of the previous one, creating the so called blockchain: an append only data structure. This means that altering a transaction requires not only the modification of the block containing it, but also of all the subsequent blocks. This traduces in the alleged immutability of the Bitcoin blockchain. In a

nutshell, this is how it is possible to achieve consensus about an economic history among a distrustful set of participants.
Now let's get back to the transaction data structure and give it a deeper look. A Bitcoin transaction consists of some inputs and some outputs: the inputs being references to some unspent outputs of previous transactions, and the outputs representing the new ownership of bitcoins[1]. These outputs are locked by the so called locking script (also referred to as scriptPubKey due to historical reasons), a script written in a special stack based, turing incomplete language (the Bitcoin scripting language). To unlock the funds (i.e. to spend them), the owner has to provide a proof of ownership in the form of an unlocking script (scriptSig), that usually comprise a digital signature. The difference in bitcoin value from inputs and outputs is collected by the miner who mines the block, and is called transaction fee (obviously the sum of the outputs' values cannot exceed the sum of the inputs' values).
Before we pass to the real life example, we would like to stress the fact that bitcoins exist only as unspent transaction output, and the so called wallets actually store the private keys needed to unlock these outputs, and not bitcoins. A proper renaming would be keychain.

Here follows an example of Bitcoin transaction, taken from block number 286731 (determined by the block hash 0000000000000000bf3856e067ec21f4c30a8a859cc7ed7f2de9a2b579200639[2]):

```
{
”hash”:”90b18aa54288ec610d83ff1abe90f10d8ca87fb6411a72b2e56a169fdc9b0219”,
”ver”:1,
”vin_sz”:1,
”vout_sz”:2,
”lock_time”:0,
”size”:226,
”in”:[
{
”prev_out”:{
”hash”:”18798f8795ded46c3086f48d5bdabe10e1755524b43912320b81ef547b2f939a”,
”n”:0
},
”scriptSig”:”3045022100c1efcad5cdcc0dcf7c2a79d9e1566523af9c7229c78ef71ee8
              b6300ab59aa63d02201fe27c3e6374dd3a5425a577d9ca6ad8ff079800
              175ef9a4475bc98bcef21cf01023b027d54ce8b6c730e0d5833f73aec6a
```

[1]The inputs of a transaction are completely spent by it. This implies that if the amount is greater than the one we intend to transfer, we have to create also an output returning the difference to ourselves.

[2]`https://blockexplorer.com/block/0000000000000000bf3856e067ec21f4c30a8a859cc7ed7f2de9a2b579200639`.

```
            5bae4efe04f57d2864a6a7df2af56e46"
}
],
"out":[
{
"value":"5.93100000",
"scriptPubKey":"OP_DUP OP_HASH160 4b358739fc7984b8101278988beba0cc00867adc
    OP_EQUALVERIFY OP_CHECKSIG"
},
{
"value":"1678.06900000",
"scriptPubKey":"OP_DUP OP_HASH160 55368b388ccfe22a3f837c9eee93d053460db339
    OP_EQUALVERIFY OP_CHECKSIG"
}
]
}
```

This is a representation of the transaction in human readable terms. The first voice is the transaction hash value, uniquely determining the transaction. Then we have the version number, the number of inputs, the number of outputs, the lock time[3] and the size of the transaction in bytes.
Then we have the inputs and the outputs: for what concerns the unique input we have the hash defining the transaction from which we take the unspent transaction output (UTXO) number 0, as determined from the subsequent voice n; that is, we look for the transaction determined by the hash value "18798...2f939a" and take its first output. The scriptSig is the script needed to unlock the funds, that we will study in a while.
The outputs are determined by the value and by the scriptPubKey, that contains a mathematical puzzle that the owner has to solve to spend the coins. The scriptPubKey has this name since originally it contained the public key of the receiver: due to privacy and security reasons this is not anymore the case, and we can find the address[4] related to the receiver in it.

Now let's analyse ScriptSig and ScriptPubKey more in detail. For the transactions of type Pay-to-PubKeyHash (P2PKH, like the ones in the example) the couple of scripts has the following form:

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
    OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

[3] Part of a transaction which indicates the earliest time or earliest block when that transaction may be added to the blockchain.

[4] An address is the result of the operator OP_HASH160 applied to the public key: first it is applied the SHA-256, then the RIPEMD160, another hash function that outputs in the space of 160 bits, or 20 bytes

Getting back to our example we can see that the scripts in there are written exactly in this way. It is easy to notice for the two scripPubKey's, since they have exactly the same structure, with two addresses of 20 bytes.
It is not that immediate for what concerns the scriptSig. But we know that it ends with a public key, that consist in 33 bytes (in compressed form), thus we can easily separate the signature from the public key:

```
sig : 3045022100c1efcad5cdcc0dcf7c2a79d9e1566523af9c7229c78ef71ee8b6300ab59aa63d
02201fe27c3e6374dd3a5425a577d9ca6ad8ff079800175ef9a44475bc98bcef21cf01
publickey: 023b027d54ce8b6c730e0d5833f73aec6a5bae4efe04f57d2864a6a7df2af56e46
```

Once the transaction is received by a node, it has to be validated. Table A.1 presents the validation procedure of the input.

After having looked at the basic construction we can give a deeper look to the verification procedure for an m-of-n multi-signature transaction. But first, let's try to understand why they are so appealing.
The primary benefit is to greatly increase the difficulty of stealing the coins. With a 2-of-2 address, you can keep the two keys on separate machines, and then theft will require compromising both; it can also be used to protect against accidental loss: with a 2-of-3 address, not only does theft require obtaining 2 different keys, but you can still use the coins if you forget any single key; it can also be used for more advanced scenarios such as an address shared by multiple people, where a majority of vote is required to use the funds. This setting would be typically used for corporate accounts, with different layers for security, and by escrow services. For practical reasons, multi-signature transactions are fundamental for a proper custody of the digital assets.

In the m-of-n scenario, the couple of scripts would look something like this:

```
scriptPubKey: m <pubKey1> <pubKey2> ... <pubKeyn> n OP_CHECKMULTISIG
scriptSig : 0 <sig1> <sig2> ... <sigm>
```

The scriptSig starts with a zero due to a bug in the OP_CHECKMULTISIG operator, that requires an additional dummy element to work.
The verification procedure in this case is exemplified in Table A.2.

Stack	Script	Description
Empty.	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	The scriptSig and the scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	The signature and the public key are added to the stack.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	The operator OP_DUP takes the last element on the stack and puts a duplicate of it on top.
<sig> <pubKey> <pubHash>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	The OP_HASH160 takes the public key as input and applies to it in sequence SHA-256 and RIPEMD160.
<sig> <pubKey> <pubHash> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	The address <pubKeyHash> is pushed on top of the stack.
<sig> <pubKey>	OP_CHECKSIG	OP_EQUALVERIFY checks that the last two elements on top of the stack are equal. If not, the validation fails.
True.	Empty.	OP_CHECKSIG checks that the signature really comes from the public key present in the unlocking script. The validation process succeeds if at the end on the stack there is a boolean value indicating true.

Table A.1: Verification procedure for a P2PKH transaction.

Stack	Script	Description
Empty.	0 <sig1> <sig2> ... <sigm> m <pubKey1> <pubKey2> ... <pubKeyn> n OP_CHECKMULTISIG	The scriptSig and the scriptPubKey are combined.
0 <sig1> <sig2> ... <sigm> m <pubKey1> <pubKey2> ... <pubKeyn> n	OP_CHECKMULTISIG	All the constants are added to the stack.
True.	Empty.	OP_CHECKMULTISIG checks whether the *m* signatures come from some of the *n* public keys.

Table A.2: Verification procedure for an input locked by an m-of-n policy.

As the reader may have detected another small flaw of the multi-signature construction adopted in Bitcoin is that the signatures have to be ordered as the public keys in the locking script. Since verification is made by subsequent attempts (first signature checked against first public key: if valid pass to the second signature, otherwise check the first signature against the second public key), a proper ordering traduces in a faster verification.

In the Bitcoin network, the bigger a transaction is in bytes, the more costly it is in terms of transaction fees. With the previous construction for the multi-signature account, the burden had to be carried by the payer, even if it was the receiver who asked for a multi-signature locking script. This is the main reason that led to the creation of Pay-to-Script-Hash (P2SH) transactions, that now allows many possibilities with various complex redeem paths. In a P2SH transaction the locking script is always of the following form:

```
scriptPubKey: OP_HASH160 <20 bytes hash of the redeem script> OP_EQUAL
```

Such a construction enables huge possibilities: in summary, it is not anymore a locking script of the type "pay to a person able to provide ownership of the private key corresponding to this address", but more generally of the kind "pay to a person validating a script whose hash is this". For a better understanding we will exemplifies the behaviour of P2SH using the previous multi-signature transaction. We have:

```
Redeem script: m <pubKey1> <pubKey2> ... <pubKeyn> n OP_CHECKMULTISIG
```

```
Locking script: OP_HASH160 <20 bytes hash of the redeem script> OP_EQUAL
Unlocking script: 0 <sig1> <sig2> ... <sigm>
```

Now the burden of the greater redeem script is on the receiver of the transaction, since the output is locked with the locking script, that is much lighter than the redeem script. When the receiver will finally try to spend this transaction, he will need to supply the serialization of the unlocking and of the redeem scripts: first the redeem script will be checked against the locking script to ensure that the hash matches, if this is the case then the redeem script is validated against the unlocking script.
Once again we would like to point out that, even if P2SH was conceived in order to make multi-signature transactions more aligned with the economic incentives, they allow a huge amount of redemption script combined with Bitcoin scripting language.

Bibliography

[1] Sec 1: Elliptic curve cryptography. http://www.secg.org/sec1-v2.pdf, 2009.

[2] Sec 2: Recommended elliptic curve domain parameters. http://www.secg.org/sec2-v2.pdf, 2010.

[3] Antonoupolos, A. M. Mastering bitcoin: Programming the open blockchain. Oreilly & Associates Inc; 2nd edition, 2017.

[4] Boneh, D., and Shoup, V. A graduate course in applied cryptography. https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf, 2017.

[5] Brown, D. R. L. Generica group, collision resistance, and ecdsa. https://eprint.iacr.org/2002/026.ps, 2002.

[6] Corbellini, A. Elliptic curve cryptography: a gentle introduction. http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[7] Eike Kiltz, D. M., and Pan, J. Schnorr signatures in the multi-user setting. https://pdfs.semanticscholar.org/0e43/be818bd4664e667154533fd2badb7be2e3b5.pdf.

[8] Eric Lombrozo, P. W. Bip144: Segregated witness (peer services). https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki.

[9] Eric Lombrozo, Johnson Lau, P. W. Bip141: Segregated witness (consensus layer). https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[10] Fry, S. Bip148: Mandatory activation of segwit deployment. https://github.com/bitcoin/bips/blob/master/bip-0148.mediawiki.

[11] Johnson Lau, P. W. Bip143: Transaction signature verification for version 0 witness program. https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki.

[12] Washington, L. C. Elliptic curves: Number theory and cryptography. Chapman and Hall/CRC; 2nd edition, 2008.

[13] Wuille, P. Schnorr's bip. https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki.