Three most common signature schemes in blockchains are: Schnorr, ECDSA (Elliptic Curve Digital Signature Algorithm), and Boneh-Lynn-Shacham (BLS). We present a unified description of generating public and private keys, signing messages, and validating them. The aim of our formalism is to balance high-level abstraction and detailed formulaic levels in cryptographic signature primitives to help understanidng the differences between the signature schemes for non-expert readers.

Preliminaries

First, we explain the basic mathematical tools used in Schnorr, ECDSA and BLS signature schemes.

One-way functions with homomorphic property

Most signatures make use of a one-way function f()

, whose inverse is computationally challenging to compute. In other words, f()

is a function that can be efficiently computed; however, it is not feasible to compute x

from f(x)

. A fundamental assumption of these signatures that:

Assumption 1

. If f(x)

is given, then there is no polynomial time algorithm (neither deterministic nor probabilistic) to compute x

. This holds true even when numerous (linear) equations involving f()

are available.

For example, x

is a secret key, and the corresponding public key is generated as f(x)

. Ethereum, as many blockchains use elliptic curve cryptography (ECC), where f(x)

is a function that maps an integer in the range [1,q-1]

to a point on an elliptic curve over a finite field \mathbb F_p

where q

and p

are large primes (or prime powers). We denote points on the elliptic curve with capital letters, and integers in the range [1,q-1]

with lowercase letters. Note that this is only one possible definition of f(x)

; however, one can redefine f(x)

and come up with a new signature scheme in the future. Should this happen, our specification, being generic enough, loses none of its value.

The signatures make use of the homomorphic (a.k.a additive) property of f()

, e.g.:

f(x+y)=f(x) \oplus f(y) \enspace .

Note that, on the left side of the equation, scalars are added using the addition modulo q

in the argument of the one-way function, whereas, on the right side, points are added to each other using the operator defined on the elliptic curve.

We can define an operator \ominus

in an analogous fashion:

f(x-y)=f(x) \ominus f(y) \enspace ,

And with these in mind, we can also define an operator, denoted by \times

, that corresponds to multiplying by a positive integer constant:

f(c x)= f(\underbrace{x + \cdots + x}_{c \text{ times}}) = \underbrace{f(x) \oplus \cdots \oplus f(x)}_{c \text{ times}} =c \times f(x) \enspace .

There are efficient algorithms to compute operators \oplus

, \ominus

, and \times

using the secret keys as arguments, and for the security assumptions to hold, there should be no efficient algorithm to obtain x

from f(x)

.

For the operations \oplus

, \ominus

, and \times

, there are efficient algorithms tuned for modern CPUs, which take negligible time to calculate. On a commodity laptop \oplus

and \ominus

take only 0.0015

microseconds to execute on secp256k1

and \times

takes 0.5

microsec.

One-way functions implementations on elliptic curves

In actual blockchain implementations f(x)

is a function over an elliptic curve over a finite field \mathbb F_p

where p

is a prime. We define a finite set where each element is a point ({\scriptstyle\mathcal{X}},{\scriptstyle\mathcal{Y}})

on the plane \mathbb F^2_p

that satisfies the equation of an elliptic curve

{\scriptstyle\mathcal{Y}}^2 \equiv {\scriptstyle\mathcal{X}}^3+a{\scriptstyle\mathcal{X}}+b \pmod{p}

where 0 \leq {\scriptstyle\mathcal{Y}}< p

and 0 \leq {\scriptstyle\mathcal{X}}< p

are {\scriptstyle\mathcal{X}}

and {\scriptstyle\mathcal{Y}}

integers (There is also an additional element of the curve that is denoted by \infty

). For example, bitcoin and ethereum uses secp256k1

where the parameters are p = 2^{256} - 2^{32} - 977

, a = 0

and b=7

.

A space-efficient way of storing an elliptic curve point is to store only its {\scriptstyle\mathcal{X}}

coordinate, which is 256

bits for secp256k1

, and compute {\scriptstyle\mathcal{Y}}

using the eqution above. More precisely, one additional bit is reserved to represent the sign of {\scriptstyle\mathcal{Y}}

. When both coordinates are needed, {\scriptstyle\mathcal{Y}}^2

can be computed from the {\scriptstyle\mathcal{X}}

-coordinate using the equation of the elliptic curve, and then a modular square root is calculated in the field p

to obtain {\scriptstyle\mathcal{Y}}

coordinate. On a commodity laptop to compute {\scriptstyle\mathcal{Y}}

from {\scriptstyle\mathcal{X}}

takes 0.17

microsec on secp256k1

. The additional sign bit is used to store whether the point is on the curve above the horizontal axis ({\scriptstyle\mathcal{X}}, {\scriptstyle\mathcal{Y}})

or its reflection over the axis, e.g., ({\scriptstyle\mathcal{X}},-{\scriptstyle\mathcal{Y}})

. Note that {\scriptstyle\mathcal{X}}

can take only even numbers, thus overall an elliptic curve point can be store in 256 bits. We denote the {\scriptstyle\mathcal{X}}

coordinate of a point R

by |R|

.

In such a plane, any line will intersect the elliptic curve in at most three points. Based on the geometric properties of the curve, a binary operator can be defined on the points of the curve that is usually referred to as addition, denoted by \oplus

.

Let ({\scriptstyle\mathcal{X}}_1,{\scriptstyle\mathcal{Y}}_1)

and ({\scriptstyle\mathcal{X}}_2,{\scriptstyle\mathcal{Y}}_2)

be two points on the elliptic curve. A third point on the line is defined by these two points intersecting the elliptic curve in the point denoted by ({\scriptstyle\mathcal{X}}_3,-{\scriptstyle\mathcal{Y}}_3)

. Here the negative of {\scriptstyle\mathcal{Y}}_3

is understood modulo p

. When accidentally the two points happen to be the same, then the line is defined as the tangent of the curve passing through this point, and in this case, the operation is usually called point doubling.

We denote this operation generating from two points on the curve a new point as follows: ({\scriptstyle\mathcal{X}}_1, {\scriptstyle\mathcal{Y}}_1) \oplus ({\scriptstyle\mathcal{X}}_2,{\scriptstyle\mathcal{Y}}_2)=({\scriptstyle\mathcal{X}}_3, {\scriptstyle\mathcal{Y}}_3) \enspace .

The operation \oplus

shares many properties of the usual addition, with \infty

behaving like 0. For any point G

on the elliptic curve, we can repeatedly apply the above addition till we reach point \infty

. Let q

be a positive integer such that G,2G=G\oplus G,3G,\dots,qG

are all distinct points on the elliptic curve. Then the last point qG

is the point at infinity \infty

.

As something of a miracle, there is a point G

in secp256k1

such that the number of elements q

is a prime and almost as large as the prime p

. This results in very space-efficient storage of public keys because each 256-bit-long private key corresponds to a 256-bit-long public key.

For the first 256 powers of 2, the algorithm known as exponentiating by squaring can be used to compute G,2G,4G,8G,16G,\dots

and 2^{256}G

. Then we can use the bit representation of x

and add the terms corresponding to the e_2

bits. Based on these considerations, an efficient algorithm can be devised to compute f(x)=xG

where x=0,\dots,q-1

. On the other hand, computing the inverse function is computationally challenging because the discrete logarithm in secp256k1

is believed to be hard to solve. The name logarithm is because multiplication is also a commutative field operator, in our terminology, it would be discrete division. There is no proof that the decision version of the problem is NP-hard. In fact, it is in NP \cap

co-NP.

Weil pairing

The BLS signature is based on a function called Weil pairing, which is a bilinear map. In this case, one has to deal with two (multiplicative) cyclic groups of prime order q

. Both are defined over elliptic curves, and the generator point of the first is denoted by G_1

, and the generator point of the second is by G_2

. In addition there are two one-way functions f_1(x)=xG_1

and f_2(x)=xG_2

.

The bilinear map is denoted by e(x,y)

. It enables one to check if

z=x+y

without the knowledge of x

, y

, and z

, by testing whether

e(x G_1,y G_2)=e(G_1,z G_2)

holds. The computation of the Weil pairing requires complex arithmetics and it is computationally intensive. Computing the Weil pairing takes 1.3

milliseconds on average utilising the BLS12-381

curve and executing on a commodity laptop.

Furthermore, there is a function Shallue-Woestijne-Ulas (SWU) that hashes a message m

to a point H

on the elliptic curve corresponding to G_1

. This algorithm is rather involved and is designed to run in constant time. On a commodity laptop SWU took 0.42

milliseconds on the BLS12-381

curve.

Digital signatures

We employ the terms public

keys and private/secret

key pairs to ensure the unforgeability of digital signatures. For each public key there exists a secret key, such that with the secret key one can generate a signature (Primitive Sign

) and the corresponding public key is sufficient to verify that indeed the corresponding key (Primitive VerifySig

) was used during the signature generation. The primitives can be thought of as cryptographic algorithms executed by a single entity. The VerifySig

returns 1

if the signature is valid and 0

otherwise. The signature should conceal the secret key such that when a message is signed, no substantial information on it is revealed.

Schnorr signatures

A Schnorr signature takes advantage of the basic principle of algebra that states that a linear equation ax+b

for a\neq 0

has a single root. In other words, there is only a single x

such that ax+b=0

for a\neq 0

. The theorem holds in F_q

where q

is a prime or prime power.

Let s \equiv r+ h \cdot x \pmod{q}

be a congruence modulo the prime q

. If we fix the value for any three of the variables 1\leq s,r,h,x < q

, then there shall be only one possible value for the fourth variable that satisfies the congruence (the fourth value (root) may be zero). In other words,

s\equiv r+ h \cdot x \pmod q

is only satisfied by integers s,r,h,x

from 1,\dots,q-1

that satisfy the following equation as well:

f(s) = f(r) \oplus \enspace h \times f(x) \enspace ,

where in case of ECC, f(s)

, f(r)

and f(x)

are points on the elliptic curve, while h

is an integer.

Schnorr Primitive Sign

(m,x)=\sigma

The public key is X=f(x)

.

Pick a random number r

, called the nonce. R =f(r)

Concatenate the message m

with R

and X

and compute its hash, i.e. h=H(m|R |X)

Compute s\equiv r+h \cdot x \pmod{q}

Return \sigma:=(R ,s)

Schnorr Primitive VerifySig

(\sigma,m,X)={0,1}

The signature is (R ,s)=\sigma

.

Compute S=f(s)

from s

.

Concatenate the message m

with R

and X

and compute its hash h=H(m|R |X)

Return S \stackrel{?}{=}R \oplus \enspace h \times X

Schnorr Primitive Sign

shows how a valid signature is generated for a message under the secret key x

. The algorithm uses the hash of the message, where the hash is calculated using a function H()

that maps an array of characters of arbitrary length to an integer [1,q-1]

(in implementations 0

is treated in a slightly different way). H()

is invoked on a string that is a concatenation of the message, f(r)

and f(x)

to make the signature ephemeral.

The signature is composed of two parts. The random nonce concealed by the one-way function R =f(r)

and s

. Note that s

does not reveal the secret key x

as long as the random nonce r

is not presented. Computing r

from R =f(r)

is as hard as computing x

from f(x)

.

Schnorr Primitive VerifySig

shows how the verification of signatures is implemented. The high-level idea is to use the homomorphic property of the one-way function f()

. Thus instead of verifying s\equiv r+ h \cdot x \pmod q

, the algorithm verifies whether the equality f(s) = f(r) \oplus \enspace h \times f(x)

holds. In practical blockchain applications, the terms in the equation are all published onto the chain.

ECDSA signatures

The Schnorr signature algorithm was patented by its inventor, Claus Schnorr, which limited its use in open standards and open-source software. ECDSA was seen as a direct adaptation of the Digital Signature Algorithm (DSA) to elliptic curves. This made it somewhat easier to integrate into systems that were already using DSA, allowing for a smoother transition and compatibility with existing cryptographic infrastructure. This early standardization and integration into various protocols cemented its position in the industry. Schnorr patent expired in 2008, but by then, ECDSA had already become widely adopted.

In DSA (Digital Signature Algorithm) one calculates the nonce and the public key in a finite field (not necessarily prime) and the signature is calculated in a cyclic subgroup using a suitable operator (usually taking \pmod{q}

). ECDSA achives the same by converting a curve point R

to an integer r

. For a slighly more detailed description of this operation, readers should refer to. A public key is a point stored using 256 bits, and a secret key is an integer in the range of [1,q-1]

, where q \lesssim 2^{256}

, which is stored using 256 bits. Observe that a public key can be converted into a secret key by ignoring the left-most bit and taking \pmod{q}

. Note that q\sim2^{256}

, thus this operator is almost a one-to-one mapping between the private and public keys. We denote this operator by |R |

, and we assume that its output is an integer in [1, q-1]

.

In comparison with Schnorr signatures, ECDSA signatures rely on slightly different congruences. They are r \equiv h\cdots + {\scriptstyle{R}} \cdot s \cdot x \pmod q

f(r) = f(h \cdot s) \oplus \enspace ({\scriptstyle{R}} \cdot s) \times f(x) \enspace ,

where {\scriptstyle{R}}= |f(r)|

is the above described cryptographic one-way conversion from r

to {\scriptstyle{R}}

.

ECDSA Primitive Sign

(m,x)=\sigma

Compute the hash of the message, i.e. h=H(m)

Pick a random number r

, called the nonce. {\scriptstyle{R}} = |f(r)|

Compute s \equiv (h + {\scriptstyle{R}} \cdot x)r^{-1} \pmod{q}

Return \sigma:=(R ,s)

ECDSA Primitive VerifySig

(\sigma,m,X)={0,1}

Extract signature (R ,s)=\sigma

, h=H(m)

, {\scriptstyle{R}}=|R |

Return s \times R \stackrel{?}{=}f(h) \oplus \enspace {\scriptstyle{R}} \times X

ECDSA Prm. Sign

show the implementation of the required primitives to calculate and verify signatures. They proceed in a very similar way to the primitives we introduced in the previous sections, with a slight difference in applying our one-way cryptographic conversion function. For the sake of simplicity, we present a modified ECDSA signature scheme, in which the implementation of ECDSA Prm. Sign

returns ({\scriptstyle{R}},s)

instead of (R ,s)

. We can reconstruct R

from {\scriptstyle{R}}

. Note that, in prime fields, the multiplicative inverse is unique, and it is usually computed using the Extended Euclidean Algorithm. Using it we can compute r^{-1}

such that 1 \equiv r^{-1} \cdot r \pmod{q}

.

BLS signatures

Schnorr and ECDSA both use a random nonce, and the signature consists of two fields, each approximately 256 bits in length. The use of a random nonce can lead to significant vulnerabilities if not managed properly. It is essential that the nonce be truly random and unpredictable. While this might seem obvious, several reported successful attacks have been where attackers were able to infer the nonce, typically due to a software bug.

Boneh-Lynn-Shacham (BLS) signatures are often favored for their signature aggregation capabilities. In BLS, there are two homomorphic one-way functions, f_1()

and f_2()

, which map integers to points on two separate elliptic curves, forming two distinct cyclic groups. These mappings adhere to the homomorphic properties defined earlier. We define the group operation for the first cyclic group associated with f_1()

by \oplus

, and G_1=f_1(1)

is its generator. Similarly, for the second group associated with f_2()

, the group operation is defined by \oplus

and G_2=f_2(1)

serves as its generator. To be short, we call them group G_1

and G_2

, respectively. For example, in BLS BLS12-381

, a point associated with f_1()

is 48

bytes (=384

bit) that will be a public key, and a point associated with f_2()

is 768

bits (a signature). A private key is a 32

byte (=256

bit) integer (i.e. q\sim2^{256}

). For curve BLS12-381

, the G_1

and G_2

groups are interchangeable. For instance, in Zcash G_1

was used for signatures due to its efficiency. In contrast, Ethereum 2.0 uses G_1

for public keys to benefit from smaller storage size and frequent aggregation, with signatures being G_2

points. It involves trade-offs in speed and storage: while G_1

points are smaller and faster, G_2

points are larger and slower.

Furthermore, there is an operator e(x,y)

, as defined earlier. Finally, there is a function Shallue-Woestijne-Ulas (SWU) that hashes a message m

to a point H

on the elliptic curve corresponding to f_2()

. This algorithm is designed to run in constant time and is rather involved. On a commodity laptop SWU took 0.42

milliseconds on the BLS12-381

curve.

BLS Primitive Sign

(m,x)=\sigma

h=H(m)

, H = SWU(h)

, S = x H

Return \sigma:=(S)

The public key is X=f_1(x)

and the secret key is x

. BLS Prm. Sign

and VerifySig

outline the implementation of calculating and veryfing the signatures. The signature itself is a point S

on the cyclic group associated with f_2()

. To compute the signature, the hash of message m

is first calculated, then mapped onto the curve corresponding to f_2()

with the SWU algorithm, which is the dominant factor in the signature computation process

The signature is verified as follows. The hash of message m

is calculated and mapped with the SWU algorithm to obtain point H

. Then, we use the bilinear mapping function to verify the signature because the following equation must hold

e(H,X) = e(H,x G_2) = e(x H , G_2) = e(S , G_2) \enspace .

BLS Primitive VerifySig

(\sigma,m,X)={0,1}

(S)=\sigma

, h=H(m)

, H = SWU(h)

Return e (S , G_2) \stackrel{?}{=}e(H,X)

Performance Evaluation

The table below shows our measurement of ECDSA, Schnorr, BLS schemes for elliptic curves secp256k1

(used in blockchains Bitcoin and Ethereum 1.0) and BLS12-381

curves (used in blockchains Ethereum 2.0 and Chia), respectively. For BLS, we used Chia’s implementation, and for ECDSA and Schnorr, the bitcoins. These implementations are both in c++

and well optimized. Runtimes are computed on a commodity laptop (with 2,5 GHz Quad-Core Intel Core i7 CPU) and averaged at over 10000

messages and signatures. Security level n

means an attacker would have to perform approximately 2^{n}

operations to break it. ECDSA and Schnorr, compared to BLS, was 40\times

faster in signature verification, 8\times

faster in signature generation, and 17\times

faster in public and secret key generation.

Signature scheme

Elliptic curve

Security level

secret key size (bits)

public key size (bits)

signature size (bits)

Keygen (microsec)

Sign (microsec)

Verify (microsec)

ECDSA

secp256k1

128

256

256

512

30

41

51

Schnorr

secp256k1

128

256

256

512

30

31

54

BLS

BLS12-381

128

256

384

768

503

265

2050

Conclusion

This work surveys Schnorr, ECDSA, and BLS cryptographic signatures for non-expert readers. We aim to find a good balance in the high-level abstraction and detailed formulas, when defining the generation and validation of public and private keys and signatures.