Implementing Digital Signatures in Rust

Simon Brown

Follow

FCAT Blockchain Incubator

--

Listen

Share

Introduction

I'm attempting to learn the Rust programming language, and for me, the only way to learn something, is by trying to solve a problem. So I've decided to try to implement my own digital signature library for educational purposes, and hopefully share a little knowledge along the way.

Elliptic curve cryptography was invented independently by Neal Koblitz and Victor Miller in 1987 and 1985, stemming from research that Lenstra had being doing to try to crack RSA. It wasn't until the mid 2000's that it started to see significant adoption. I'm not going to go into the background material on elliptic curve cryptography, for that there are a number of well-known resources that are well worth reading [1] [2]. However, though there are some great introductions to the theory of ECC available online, the resources that explain the actual implementation side of things can be a bit hard to digest at first.

So instead of going over the theory, I'm going to focus on writing an implementation in Rust, but will try to explain as much as is needed to help you understand what the code is doing, and how to actually implement some of theory you've read about. As I mentioned, I am learning this stuff, so if you spot anything incorrect or something I can do in a better way, do me a solid and leave a comment telling me about it.

Public Keys

The first thing we need to do is to create a public key, and to do that, we need to choose a curve. There's plenty to choose from, but I'm going to stick with the tried and tested secp256k1. For a list of other curves that you could use, see SafeCurves [3].

Creating a public key involves multiplying our "generator point" by our private key. Our private key is a 256 bit integer, and the "generator point" is a point that is specified in the curve parameters for our chosen curve. In order to multiply any point G

by a scalar value n

, we basically add it to itself repeatedly, n

times. So how do we add a point to itself? We use the following formula, where p

and q

are points, and p = q

and p + q = r

:

Note that this formula is for adding a point to itself, for adding a point to another point, the lambda value is instead calculated using the equation of line, i.e.:

Point Doubling

Ok, not so bad, looks fairly straightforward, except there's catch. In fact there's two:

Catch number one: if we take the naive approach and multiply our generator point G

by our private key n

by adding P

to itself n

times, we'll be here forever (literally). This is because n

is a very large number. This is why we need a workaround, which I'll delve into shortly.

Catch number two: we are performing all operations inside a finite field, therefore operations like addition, subtraction and multiplication behave as expected, but division is problematic. So instead of dividing by n

, we multiply by the multiplicative inverse of n modulo p

, with p

being the prime order of our field (as specified in the curve parameters for our chosen curve).

The common way to do this is by using the Extended Euclidian Algorithm. Fortunately this is quite well documented, in fact, there's an entire website dedicated to it[4].

Let's try to code this up using recursion:

If you read through the code above you'll see it follows the algorithm fairly closely. There are a couple of things to note with regards to how I've written it in Rust. For one, I've used the `Option`

enum for the parameters t1 and t2, and I've used the`unwrap_or`

method to set a default value in case none is specified. This is as close as you'll get to default method parameters in Rust, in this case, we would pass a `None`

value for t1 and t2 on the initial call to this method.

The other oddity you might notice is that I'm making a call to a function called`modulo`

. This is something I had to implement myself, the reason being that in Rust, the %

operator performs the remainder operation, not the modulus operation, which can have unexpected results. Look at this example to see what I mean:

-21 modulus 4 => 3

-21 remainder 4 => -1

println!("{}", -21 % 4); // prints "-1"

The only other thing that probably requires explanation is that I'm using the`BigInt`

library, and I've imported the function `zero`

and `one`

directly, which is why you see `zero()`

instead of just `0`

.

So that's it, go ahead and test this against the calculator on the Extended Euclidian Algorithm website, and hopefully it should work.

Right, now that we have our Extended Euclidian Algorithm coded up, let's delve into how we can multiply a point by a scalar value in an efficient way. We'll begin by creating a function for adding a point to itself, otherwise known as "point-doubling".

First create a Point struct:

Notice we're implementing `Clone`

trait for this struct, so that we can easily clone an entire instance should we need to, you'll why this is useful later.

Once we have our Point struct, we can create our point double function, which will take a point and add it to itself to itself. If you read this code carefully, you'll see that we're directly implementing the point addition formula that we included earlier. Again, we're using our custom modulo function for safety, and we're replacing the division operation by using the modular multiplicative inverse instead.

Double and Add

In theory we should be able to create our public key at this stage, by simply doubling the generate point (i.e. adding it to itself), and repeating that process n

number of times, (with n

being our private key). As I mentioned earlier, this is where theory and implementation diverge. To do this practically, we will use the “double-and-add

” algorithm. This involves converting our scalar value (i.e. our private key) into its binary representation, i.e.:

Once, we have the binary expansion of our scalar, we can use it in our double-and-add algorithm. We start with the point at infinity, (i.e. our identity point in our finite field), as our starting point. We then iterate through each bit of our scalar private key, right-to-left, i.e. from least-significant to most-significant bit, and double our generator point on each iteration. For every bit of d

that is set, we add the value that we have in our current iteration, to our starting point.

Let’s try to visualize what we’re doing with an example. Let’s use n = 1356:

Scalar value in decimal: 1356

Binary representation: 10101001100

1024 + 256 + 64 + 8 + 4 = 1356

This approach gives us a much more efficient way to perform multiplication. In this example we performed 10 doublings and 5 additions. Much better than 1356 doublings.

Now in order to use this approach we need to add a new method to our code. This method performs addition of a point to another point, in the scenario that the two points being added are different (remember our previous formula was only for adding a point to itself). The formula for point addition for two different points is very similar:

Let’s add this new point addition formula to our code so that we can then implement the double-and-add algorithm. We’ll add a new function to our Point struct, that’s very similar to our double function, except we’ll just plug in the formula from above.

The main thing that you’ll notice from the code above, is that there are four exceptions that we check for before we perform the point addition formula. Without going into too much detail, (Silverman has a great explanation if you wish to find out more [5]), we can mention these edge cases as being:

1. The two points we are adding are both the same, in which case we use the point doubling formula, as we covered previously (usually this means the point is a tangent to the curve).
2. Both points are additive inverses of each other, in other words, the x

coordinate is the same in each point, but the y

is different, (this looks like a vertical line when graphed). In this case we can only return the point at infinity.

1. If either point is the “point at infinity”, or the identity element within our group, in which case adding it to the other point makes no difference, e.g. 2 + 0 = 2

.

Now that we have both our `double` and `add` methods, we can go ahead and implement our double-and-add algorithm:

Fairly straightforward at this point. Putting it all together, we now have everything we need to create our public key. The code below uses the `Point`

struct we just created, and we’re feeding in the parameters of the secp256k1 curve, as I mentioned at the start of the post. I’m using an anonymous function called `bigint`

to just make the code a little more readable.

If you go ahead and run this code, you should get a public key that looks like this:

03d2bc5446c3ef8b8d2be6e62d33be132050cb307f4e872df8383fb803b646305f

If you want to, you can test it against the values obtained using a public tool such as Greg Walker’s public key generator [6].

Wait a minute, what’s that `compress_point`

function we called at the end there? Basically we are using a form of compression called “SEC point compression” [6]. Ok, bear with me here…

By looking at a graph of an elliptic curve, it’s easy to see that every x

value has two y

values, one positive, one negative. Now, recall that we’re working inside a finite field to the order some prime p

, so there aren’t really negative

numbers, i.e. there’s no (x, y)

and (x, -y)

, instead what we have is actually (x, y)

, and (x, p - y)

, make sense? Ok, so if p

is a prime number, then it follows that it must be an odd number, which means that if the y

coordinate of our point is even, then p - y

must be odd, because odd minus even is odd. Inversely, if y

is odd, then p - y

is even. It all boils down to the fact that the type of elliptic curve we’re using can have at most two y

coordinates for every point, positive and “negative”. This means that we can extrapolate the y

coordinate given only the x

coordinate, using the equation of the curve, which is: $y^2 = x^3 + ax + b$

, so as to know which one of the other two points to use.

The only information we need then, is our x

value, and a single byte to represent whether our y

value is even or odd, (0x03

for odd, 0x02

for even). Writing the code for the compress_point

function is very straightforward. Note that we’re using the format module with a formatting trait of x

to convert to lowercase hexadecimal.

Let’s see how we would code this function then:

Signature algorithm (ECDSA)

So now that we have our public key, that we’ve generated from our private key, we can actually do all sorts of things with it. We can use it in an ECDH key-exchange to generate an ephemeral symmetric key for encryption, or we can use it to create a digital signature. I’m going to use it to create a digital signature using the ECDSA algorithm, as described by the ANSI X9.62 standard [7] . The signature is generated as follows:

Where m

is the message to sign, H

is a hash function (sha256), n is our private key, and r

$_x$ is the x

coordinate of our public key. The signing algorithm itself is surprisingly simple:

1. Choose some random integer k: $k \in \mathbb{F}_p$
2. Calculate the point $(x, y) = k \cdot G$
3. Calculate r

$= x$

1. Calculate $s = ((m + r \cdot n) \div k) \bmod p$
2. The signature is the tuple (r, s)

Where k

is a random nonce within our finite field, G

is our curve's generator point, m

is the message to sign (or a hash of the message more precisely), and n

is our private key.

You'll notice that we're making a call to a function called`get_entropy`

if none was provided. Imagine this function being some random number generation that provides us a secure source of randomness. This is actually non-trivial so I'll re-visit this later in the post. The most important thing to remember is that it has to be truly random, (or practically indistinguishable from random [8]), and must NEVER BE RE-USED! [9].

The `get_message_hash`

function can in theory be anything, but the standard hash function for ECDSA in most cases is sha256. Also, I've made a call to a `get_curve`

function, which just returns all the curve parameters I used earlier, to save typing them out again.

With that covered, we can now proceed to creating our verification algorithm, which is also surprisingly simple as well:

Verification algorithm (ECDSA)

1. Verify that r

and s

are elements of our finite field.

1. Calculate $w = s^{-1} \bmod p$
2. Calculate u

$_1 = z \cdot w \bmod p$

1. Calculate u

$_2 = r \cdot w \bmod p$

1. Calculate the point $(x, y) = u$

$_1G + u$

$_2Q$

1. Verify the signature by asserting that $r = x \bmod p$

Where r

and s

are our signature, p

is the order of our finite field, G

is our curve's generator point, Q

is our public key and z

is our message is the hash of the message that we are signing.

We now have a working implementation of the ECDSA signature scheme. As with the generation of our public key, you can go ahead and test your generated signature using a tool such as this one:

https://www.javacardos.com/tools/ecdsa-sign-verify .

As it stands, there is nothing wrong with the implementation so far, but in the world of blockchain, every millisecond counts, especially when doing something like competing to mine a block, or syncing from a genesis block, whereby there are thousands of signatures to verify. In these cases, every millisecond you save adds up.

Let's look at some ways that we can speed up our calculations a little bit, and save some time with our signature verification.

Window method

The simplest way to reduce the computational cost of creating or verifying signatures, is to reduce the number of additions we perform during point multiplication. One such way to do this is to use what's known as the "windowed method", which reduces the number of times we need to invoke the "add" operation. According to Wikipedia [10], the algorithm for the windowed method is as follows:

The key to understanding this algorithm, and why it works, is to understand that it works from a set of pre-computed points. These pre-computed points can be cached and re-used for both public key and signature generation. Let's take an example:

Let's look at a normal double-and-add for the scalar 2329.

2329 in binary is 100100011001, so going from right to left:

As you can see there are 12 doubles and five additions. Now I'm going to try to convert this to a window method so that there are less additions. Let's use a window size of 4:

This means only three additions, a reduction of 40%, excellent. Remember, a 40% reduction is quite a lot when dealing with hundreds of operations.

The key to understanding this is that the 9, 16, and 2304 values in the above example are all contained in our pre-computed set of values. According to Wikipedia [10]: "one selects a window size w

and computes all 2

w values of dP

for d = 0, 1, 2, …, (2^w) - 1"

.

If we use a window size of 4, with P

being our generator point, then the pre-computed values would be: { P, $2P$, $3P$, $4P$, $5P$, . . . ,$13P$,$14P$,$15P$ }

(since w = 4 and 2

w -1 = 16 - 1 = 15

).

Let's do a walkthrough with another scalar value, let's say we use the number 3245:

First we need to start with a value of m

. To obtain m

, we take our scalar value, (the one we're going to multiply the generator point by to get our public key), and convert it to binary, count the number of bits, and divide by our window width, in this case 4.

3245 = 1100 1010 1101 = 12 bits. 12 / 4 = 3, therefore m

= 3, and there will be 3 iterations in our loop. Then we calculate an array of window values, which should be of length m

:

- Take right most 4 bits of n

, which is 1101, and add it to d

so that d

looks like [1101].

- Take the next right most 4 bits of n

, which is 1010, and add it to d

so that d

looks like [1101, 1010].

- Take the next right most 4 bits of n

, which is 0001, and add it to d

so that d

looks like [1101, 1010, 1100]. Convert to decimal and d

is now [13,10,12].

Iteration 1:

- i = 3
- double Q

(point at infinity) 4 times (since we're using a window width of 4).

- because we're starting with the identity element, the first iteration yields: 0 + 0 = 0.
- If d

$_i$> 0

, get the point from the pre-computed points at index d

$_i$, and in this iteration i = 3,

therefore d

$_i$= d

$_3$ (actually d

$_2$ since our array is zero-indexed), which is 12

.

- Look up the point at index 12

in our pre-computes, and add it to Q.

- Q

is now12P.

Iteration 2:

- i = 2
- double Q

for w

iterations (i.e. 4 times), so we have $2^4 \cdot Q = 2^4 \cdot 12P$

- in this iteration i = 2,

therefore d

$_i$= d

$_2$ (d

$_1$ since our array is zero-indexed), which is 10.

- Look up the point at index 10 in our pre-computes, and add it to Q.
- Q

is now ($2^4 \cdot 12P$) + 10P = 192P + 10P = 202P.

Iteration 3:

- i = 1
- double Q

4 times, so we have $2^4 \cdot Q = 2^4 \cdot 202P = 3232P$.

- in this iteration i = 1,

therefore d

$_i$= d

$_2$ (d

$_0$ since array is zero-indexed), which is 13.

- Look up the point at index 13 in our pre-computes, and add it to Q.
- Q

is now 3232P + 13P = 3245P.

As this algorithm is not too complicated, it's not that difficult to code in Rust, but I did run into a couple of oddities that perhaps require some explanation. The code for the above algorithm is below:

First, you'll notice that this function accepts a parameter p

as an argument, these are our pre-computes, I'll cover how to generate those shortly.

You'll notice that in order to derive the windowed representation of our scalar value, I'm repeatedly performing a bitwise AND operation on the scalar with a 'mask' (which is basically four 1s), followed by a bitwise right-shift. Ok, so this is fairly obvious I think, but in case it isn't, I thought I would point it out.

Notice as well that we need each 'chunk' in the windowed representation of our scalar to be cast to u16

, which makes sense as we only need unsigned integers due to the fact that we are in a finite field, and we are using a window width of 4, (i.e. 24 = 16). You'll notice as well that we're using a usize

type to index the array of chunks, if we don't do that we get a compiler error: "slice indices are of type usize or ranges of usize"

. This is one of the more idiomatic sides of Rust: array slices must be indexed using a usize

, which is basically an unsigned integer type that has the same size as a memory pointer. It is used to point to a location in memory or to refer to the size of an object in memory. On a 32 bit system, usize

is 32 bits, on a 64 bit system, yep, you've guessed it: its 64 bits.

In case you're wondering how the pre-computes are calculated, that part of it is very simple to implement, and probably almost not worth including, but here it is for reference. We call this function and pass in our generator point and window width. We repeatedly add the generator point to itself (note: that doesn't mean we repeatedly double it, it's a linear series, not quadratic). We keep going until we have 2

^{w} points, where w

is our window width, like so:

Even faster, wNAF:

The windowed method is a subtle but efficient improvement over our basic double-and-add algorithm, but it can be improved further by representing our scalar value in non-adjacent-form, and applying a window to it. Let me explain…

When we look at our basic double-and-add algorithm, we can see that every time we have a 1 in the number we are multiplying by, then we invoke an addition operation, which can be costly. For a scalar value with n

number of bits, then the cost is roughly n

doubles + n/2

additions. However, if we reduce the hamming weight

of the scalar value before multiplication then we will naturally reduce the number of additions. In fact, when using a window width of w

, the number of additions becomes approximately 1 / w+1

which is a significant improvement.

To illustrate what this looks like, I'm going to steal an example from "A Mathematical Analysis of Elliptic Curve Point Multiplication" by Kodali [11]:

Let's take the integer 1234567. It's binary representation would be: 100101101011010000111. In this representation, there are 11 instances where the bit is not zero, which results in 20 doubles and 10 additions. Now let's use the w-width NAF representation: 9 0 0 0 0 13 0 0 0 0 13 0 0 0 0 0 0 7. Here there are only 4 instances where the bit is not zero. The fact that is non-zero representation is not a 1 can be strange looking, but remember that we are using precomputes, so when we see a 13, we are just referencing 13P in our pre-computes. Using this representation, there are only 4 additions, compared to eleven.

In order to pull this off, we need to break it into three parts:

- Calculate the our pre-computes
- Determine the w-width NAF of our scalar value
- Multiply our point by the w-width NAF

Determining the w-NAF:

Note: the notation $a \leftarrow b \bmod c$

means the a

is an integer satisfying $a \equiv b \pmod c$.

How this is actually calculated is not in scope for this post. For more information on how to actually do this, here are some handy resources on Khan academy, including solving equivalence relations [12], and modular congruences [13].

Implementing this algorithm in rust is fairly straightforward:

As well as calculating the w-NAF, we also need to calculate our pre-computes, which is similar to how we calculated the pre-computes for the windowed method:

Which looks a bit like this when coded up in Rust:

Once you've calculated the actual w-NAF, we need an algorithm for multiplying a point by it. If it isn't entirely clear how or why this works, there are a good number of papers that describe the logic behind how it much better than I can [14] [15] [16] .

In this algorithm, n

is our private key, and P

would be our generator point, Q

is initialized to the point at infinity. So now that we've seen the algorithm, how do we code it in Rust? Glad you asked…

This doesn't really require any explanation at all does it? It follows the algorithm very closely. The only thing to note is where the algorithm "subtracts" one of the pre-computes from Q

. In fact what we're doing is just negating the point by changing the sign of the y

coordinate.

There are other algorithms that are frequently used as well, but they are all sort of similar. The other one to look out for is the Montgomery ladder, which is frequently used in EdDSA signatures such ed25519 or ed448, which uses an Edwards curve instead of sec256k1 or similar.

Jacobi Points

If you examine the double and add methods that we use for multiplication, you can see that the most complex part of each method is the recursive extended-Euclidian-algorithm that is used to compute the modular multiplicative inverse that we need for modular division. When dealing with 256 bit numbers, there's a lot of recursion, and it becomes a very expensive operation. The good news is that there's a trick we can use (isn't there always) that eliminates the need for that computation. It involves essentially swapping the division operation for a dozen more multiplication operations, and then to do one division at the end. How this actually works is beyond the scope of this post, but it involves using "Jacobian" coordinates, which are coordinates in a projective plane. Let me explain…

The points we've been using up until now are called "affine points" and have two coordinates, (x, y)

, whereas points in projective space have three coordinates (x, y, z)

. Converting from Affine to Jacobian involves choosing any z

, such that $(xz^2, yz^3, z) \leftarrow (x, y) : z \in \mathbb{N}$. So long as z

is not zero, then any value will do, but by convention z

is normally chosen as 1, as this just makes calculations easier. Converting from Jacobian back to affine is simply the inverse, i.e.:

As an example, if we take the affine coordinates (x, y)

, and choose z

with a value of say 2, then the Jacobian coordinates would be (4x, 8x, 2)

, with z = 3

it would (9x, 27y, 3)

, etc. The version with z=1

is the easiest to create, so usually you would use this one, leaving x

and y

unchanged.

Note that Jacobian coordinates are similar to, but calculated slightly differently to, what are referred to as "projective coordinates" [17] .

In order to implement this, both the double and the add methods will use a different algorithm. There are a number of variations on the algorithm that we can use, but the one that I opted for is the following from 2009:

You can find this algorithm, and many others, at Hyperelliptic website'sExplicit Formulas Database [18] .

Side Channel Attacks

The ECDSA library that we've developed so far is fine, it will work, and will work fairly efficiently, but it has one major weakness: it's vulnerable to side channel attacks. Side channel attacks use indirect analysis of all information available to the attacker, such as the power consumed by the processor while executing a signature for example. There are numerous side channel attacks, but perhaps the most commonly referred to is a timing attack. A timing attack is a form of CPA, or chosen-plaintext-attack, which measures the amount of time taken to compute a signature of a number of known plaintexts. Since the hash of the message used in the signature algorithm is already known, an attacker can analyse the time taken to compute the signature and work backwards to figure out the private key. Even we can't figure out the private key exactly, we can narrow it down a range that allows for a feasible, (if not expensive), brute-force attack. If you want to know more about how these sort of attacks work against an ECC system, there are some well-known papers that go more detail [20] [21].

The most straightforward way to mitigate against timing attacks is to implement constant-time multiplication for generating our public key. Let's look at our first implementation of the double-and-add algorithm. You'll notice that while we double our identity point on every iteration, we only use the addition operation when the current bit of our private key is odd. Intuitively, we can reason that if n

is a larger number, it will likely have more bits that are positive, therefore involving more addition operations, and thus taking longer to compute our public key. If n

is a smaller number, it is likely to have less positive bits and will take a shorter amount of time to compute the public key.

In order to make this a safe-multiplication algorithm, we need to make it constant time, and the easiest we to do that is to perform a “fake” addition operation on every iteration, sort of like this:

Now there's a double and an addition operation on every iteration, which will result in a constant time multiplication operation. This will give us reasonable protection against side channel attacks including SPA and timing attacks, but will also make our algorithm less efficient, which is an unfortunate trade-off. For this reason it’s recommended to have two multiplication methods, a ‘normal’ method and a ‘safe’ method as above. Remember, we don’t need ‘safe’ multiplication when carrying out signature verification, as all required properties in the verification process are already known, so there’s basically nothing to steal.

Future Work

- Explain how to secure entropy deterministically using IETF’sRFC-6979
- Demonstrate how to implement DER encoding using the Distinguished Encoding Rules.
- Show how to recover a public key from a signature.

Conclusion

There are many more topics that I would love to cover here, in fact, we haven’t even really gotten to the fun parts yet. I would love to cover Schnorr signatures for instance. Schnorr signatures are very simple to implement. We basically take everything we’ve implemented here, but change the actual signing and verification algorithms. What makes Schnorr signatures cool, is that they are additively homomorphic. That means that if I have signature s, and I can verify it using the public key a

, and I have another signature s’

, which I can verify using the public key a’

, then I add the signatures s + s’

together to get b

, and then I add the two respective public keys together, (s + s’)

, I get an aggregate public key f

, which can be used to verify the aggregate signature b

. Now you can do all sorts of cool stuff with these aggregate signatures, including things likeTaproot (when combined with MAST), and Mimblewimble (when combined with Pedersen commitments). If I get the chance I’ll try to cover these in a future post, but don’t wait around for that, go check them out if you’re not familiar with them, they are very interesting subjects!

I hope you’ve found at least parts of this post interesting or useful. If you have any comments or suggestions, or have spotted any mistakes, I’d really appreciate any feedback. I plan on writing some more crypto-related engineering posts in the future, so stayed tuned.

Works Cited

[1]. Available: https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

[2] M. Hughes, “How Elliptic Curve Cryptography Works,” All About Circuits, 26 June 2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/elliptic-curve-cryptography-in-embedded-systems/.

[3] D. J. Bernstein and T. Lange, “Introduction,” 1 December 2014. [Online]. Available:https://safecurves.cr.yp.to/.

[4] D. d. M. Keizer, “Multiplicative inverse,” 2018. [Online]. Available: https://extendedeuclideanalgorithm.com/multiplicative_inverse.php.

[5] J. H. Silverman, “An Introduction to the Theory of Elliptic Curves,” 7 June 2006. [Online]. Available: https://www.math.brown.edu/~jhs/Presentations/WyomingEllipticCurve.pdf.

[6] G. Walker, “Learn me a bitcoin,” 10 March 2018. [Online]. Available:https://learnmeabitcoin.com/technical/public-key.

[7] Standards for Efficient Cryptography Group, “Standards for Efficient Cryptography Group,” 21 May 2009. [Online]. Available: https://www.secg.org/sec1-v2.pdf.

[8] D. Johnson and A. Menezes, “The Elliptic Curve Digital Signature Algorithm (ECDSA),” 23 August 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.8014.

[9] W. contributors, “Semantic Security,” Wikipedia, The Free Encyclopedia., 17 April 2020. [Online]. Available: https://en.wikipedia.org/wiki/Semantic_security.

[10] B. Buchanan, “Recovering The Private Key in a ECDSA Signature Using A Single Random Nonce,” Medium.com, 6 August 2020. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/cracking-ecdsa-with-a-leak-of-the-random-nonce-d72c67f201cd.

[11] W. contributors, “Elliptic curve point multiplication,” Wikipedia, The Free Encyclopedia., 28 January 2021. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Windowed_method.

[12] R. Kodali, “A Mathematical Analysis of Elliptic Curve Point Multiplication,” Communications in Computer and Information Science, vol. 467, 2014.

[13] Khan Academy, “Equivalence relations,” Khan Academy, [Online]. Available: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/equivalence-relations.

[14] Khan Acamdemy, “Congruence modulo,” Khan Acamdemy, [Online]. Available: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/congruence-modulo.

[15] D. Hankerson, J. López Hernandez and A. Menezes, “Software Implementation of Elliptic Curve Cryptography over Binary Fields,” in Cryptographic Hardware and Embedded Systems — CHES 2000, Worcester, ME, USA, 2000.

[16] J. A. Solinas, “Efficient Arithmetic on Koblitz Curves,” Designs, Codes and Cryptography, vol. 19, no. 2, p. 195–249, 2000.

[17] M. Almousa, A. Sokhon, M. Sh, M. Daoud and H. Almimi, “A Survey on Single Scalar Point Multiplication Algorithms for Elliptic Curves over Prime Fields,” IOSR Journal of computer engineering, vol. 18, no. 2, pp. 278–661, 2016.

[18] Nayuki, “Elliptic curve point addition in projective coordinates,” 14 May 2018. [Online]. Available: https://www.nayuki.io/page/elliptic-curve-point-addition-in-projective-coordinates.

[19] T. Lange and D. J. Bernstein, “Explicit Formulas Database,” [Online]. Available:https://hyperelliptic.org/EFD/.

[20] B. B. Billy Bob Brumle and N. Tuveri, “Remote timing attacks are still practical,” in Proceedings of ESORICS 2011, volume 6879 of LNCS, Leuven, 2011.

[21] W. Wunan, C. Hao and C. Jun, “The Attack Case of ECDSA on Blockchain Based on Improved Simple Power Analysis,” in Artificial Intelligence and Security, Springer, Cham, 2019, pp. 120–132.

[22] V. S. Miller, “Use of Elliptic Curves in Cryptography,” in Advances in Cryptology — CRYPTO ’85 Proceedings. CRYPTO 1985, Berlin, 1985.

[23] N. Koblitz, “Elliptic curve cryptosystems,” Mathematics of Computation, American Mathematical Society, pp. 203–209 , 1987.

[24] N. I. o. S. a. Technology, “FIPS 186–5 (Draft) — Digital Signature Standard (DSS),” October 2019. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/186/5/draft.