

BLS12-381 For The Rest Of Us

Everything I wish I'd known before I started fiddling with this thing.

The elliptic curve BLS12-381 (<https://electriccoin.co/blog/new-snark-curve/>) has become something of a celebrity in recent years. Many protocols are putting it to use for digital signatures and zero-knowledge proofs: Zcash, Ethereum 2.0, Skale, Algorand, Dfinity, Chia, and more.

Unfortunately, existing material about BLS12-381 is full of obscure incantations like "instantiating its sextic twist", and "optimal extension field towers". I'm here to fix that 😊^[1]

I won't be giving a general introduction to elliptic curves and their exciting group properties. There are already some great primers (<https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/>) out there and I will be assuming basic knowledge of these things. Equally, there's much in here that is not specific to BLS12-381, and applies to other curves.

Motivation

BLS12-381 is a pairing-friendly elliptic curve.

Pairing-based cryptography (https://en.wikipedia.org/wiki/Pairing-based_cryptography) has been developed over the last couple of decades, enabling useful new applications such as short digital signatures (<https://www.iacr.org/archive/asiacrypt2001/22480516.pdf>) that are efficiently aggregatable (<https://crypto.stanford.edu/~dabo/pubs/papers/agggreg.pdf>), identity-based cryptography (https://en.wikipedia.org/wiki/Boneh-Franklin_scheme), single-round multi-party key exchange (<http://cgi.di.uoa.gr/~aggelos/crypto/page4/assets/joux-tripartite.pdf>), and efficient polynomial commitment schemes such as KZG commitments (<https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html>).

Pairing-friendly elliptic curves are curves with both a favourable embedding degree (to be explained below!), and a large prime-order subgroup (also see below). These are rare. If you create an elliptic curve at random, it has a miniscule chance of being pairing-friendly. However, they can be constructed, and BLS curves are explicitly constructed to be pairing-friendly. Several other families of pairing-friendly curves are also known (<https://eprint.iacr.org/2006/372.pdf>).

Some good reading if you want to learn more about pairing-based cryptography:

- A short (but technical) [explainer](https://courses.csail.mit.edu/6.897/spring04/L25.pdf) (<https://courses.csail.mit.edu/6.897/spring04/L25.pdf>), and [another one](https://www.math.uwaterloo.ca/~ajmeneze/publications/pairings.pdf) (<https://www.math.uwaterloo.ca/~ajmeneze/publications/pairings.pdf>).
- Vitalik with a great general introduction to [elliptic curve pairings](https://vitalik.ca/general/2017/01/14/exploring_ecp.html) (https://vitalik.ca/general/2017/01/14/exploring_ecp.html).
- This [NIST report](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4730686/) (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4730686/>) is quite readable. I recommend Section 2 and the Appendix.
- Also good background is the [draft IETF standard](https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html) (<https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html>) for pairing-friendly curves.

If you really want to *understand* this stuff then [Pairings for Beginners](https://www.craigcostello.com.au/s/PairingsForBeginners.pdf) (<https://www.craigcostello.com.au/s/PairingsForBeginners.pdf>) is awesome. It turns out to be a lot less scary than it looks if you work through it carefully, studying the examples as you go. I really recommend this (I'm perpetually half way through it...).

About curve BLS12-381

History

Curve BLS12-381 was [designed](https://electriccoin.co/blog/new-snark-curve/) (<https://electriccoin.co/blog/new-snark-curve/>) by [Sean Bowe](https://twitter.com/ebfull) (<https://twitter.com/ebfull>) in early 2017 as the foundation for an upgrade to the Zcash protocol. It is both pairing-friendly (making it efficient for digital signatures) and effective for constructing zkSnarks.

A proliferation of "next-generation", scalable blockchain protocols has put a premium on generating short digital signatures that can be efficiently aggregated or easily thresholded. The properties of BLS12-381 frequently make it the curve of choice for these protocols.

Several cryptographic libraries—established ones such as Apache's [Milagro](https://milagro.apache.org/) (<https://milagro.apache.org/>), and emerging ones such as [Herumi](https://github.com/herumi/mcl) (<https://github.com/herumi/mcl>) and [Blst](https://github.com/supranational/blst) (<https://github.com/supranational/blst>)—support BLS12-381. And there are already moves to include BLS12-381 in IETF standards, such as [Pairing-Friendly Curves](https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2) (<https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2>), [Hashing to Elliptic Curves](https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05#section-8.9) (<https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05#section-8.9>), and [BLS signatures](https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00#section-4.2) (<https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-00#section-4.2>). This is good news for protocol interoperability!

Naming

BLS12-381 is part of a family of curves described by [Barreto, Lynn, and Scott](https://eprint.iacr.org/2002/088.pdf) (<https://eprint.iacr.org/2002/088.pdf>) (they are the B, L, and S in view here - a different BLS trio will appear later).

The 12 is the embedding degree of the curve: neither too low, nor too high. We'll discuss embedding degrees in a little while.

The 381 is the number of bits needed to represent coordinates on the curve: the field modulus, q . The coordinates of a point come from a finite field that has a prime order, and that prime number, q , is 381 bits wide. 381 is a fairly handy number as we can use 48 bytes per field element, with 3 bits left over for useful flags or arithmetic optimisations. This size of this number is guided both by security requirements and implementation efficiency.

Curve equation and parameters

The basic equation of the BLS12-381 curve is $y^2 = x^3 + 4$.

The key parameters for a BLS curve are set using a single parameter \mathfrak{x} (different from the x in the curve equation!) that can be selected to give the curve nice properties for implementation. BLS12-381 is derived from the $k \equiv 0 \pmod{6}$ case of Construction 6.6 in the taxonomy (<https://eprint.iacr.org/2006/372.pdf>).

Specific design goals for BLS12-381 are:

- \mathfrak{x} has "low hamming weight", meaning that it has very few bits set to 1. This is particularly important for the efficiency of the algorithm that calculates pairings (the Miller loop).
- The field modulus q mentioned above is prime and has 383 bits or fewer, which makes 64-bit or 32-bit arithmetic on it more efficient.
- The order r of the subgroups we use is prime and has 255 bits or fewer, which is good for the same reason as above.
- The security target is 128 bits - see below.
- To support zkSnark schemes, we want to have a large power of two root of unity in the field F_r . This means we want 2^n to be a factor of $r - 1$, for some biggish n . (Making \mathfrak{x} a multiple of $2^{\frac{n}{2}}$ will achieve this.) This property is key to being able to use fast Fourier transforms for interesting things like polynomial multiplication.

The value $\mathfrak{x} = -0\text{xd}201000000010000$ (hexadecimal, note that it is negative) gives the largest q and the lowest Hamming weight meeting these criteria. With this \mathfrak{x} value we have,

Parameter		Equation	Value (hexadecimal)
Field modulus	q	$\frac{1}{3}(x-1)^2(x^4-x^2+1)+x$	0x1a0111ea397fe69a4b1ba7b6434
Subgroup size	r	(x^4-x^2+1)	0x73eda753299d7d483339d8080

Reference (https://github.com/zkcrypto/pairing/tree/master/src/bls12_381) for much of this section. Lots of curve data is also in the IETF specification (<https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2>).

Field extensions

Field extensions are fundamental to elliptic curve pairings. The “12” in BLS12-381 is not only the embedding degree, it is also (relatedly) the degree of field extension that we will need to use.

The field F_q can be thought of as just the integers modulo q : $0, 1, \dots, q-1$. But what kind of beast is $F_{q^{12}}$, the twelfth extension of F_q ?

I totally failed to find any straightforward explainers of field extensions out there, so here’s my attempt after wrestling with this for a while.

Let’s construct an F_{q^2} , the quadratic extension of F_q . In F_{q^2} we will represent field elements as first-degree polynomials like $a_0 + a_1x$, which we can write more concisely as (a_0, a_1) if we wish.

Adding two elements is easy: $(a, b) + (c, d) = a + bx + c + dx = (a + c) + (b + d)x = (a + c, b + d)$. We just need to be sure to reduce $a + c$ and $b + d$ modulo q .

What about multiplying? $(a, b) \times (c, d) = (a + bx)(c + dx) = ac + (ad + bc)x + bdx^2 = ???$. Oops - what are we supposed to do with that x^2 coefficient?

We need a rule for reducing polynomials so that they have a degree less than two. In this example we’re going to take $x^2 + 1 = 0$ as our rule, but we could make other choices. There are only two rules about our rule^[2]:

1. it must be a degree k polynomial, where k is our extension degree, 2 in this case; and
2. it must be irreducible (https://en.wikipedia.org/wiki/Irreducible_polynomial) in the field we are extending. That means it must not be possible to factor it into two or more lower

degree polynomials.

Applying our rule, by substituting $x^2 = -1$, gives us the final result $(a, b) \times (c, d) = ac + (ad + bc)x + bdx^2 = (ac - bd) + (ad + bc)x = (ac - bd, ad + bc)$. This might look a little familiar from complex arithmetic: $(a + ib) \times (c + id) = (ac - bd) + (ad + bc)i$. This is not a coincidence! The complex numbers are a quadratic extension of the real numbers.

Complex numbers can't be extended any further because there are no irreducible polynomials over the complex numbers

(https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra). But for finite fields, if we can find an irreducible k -degree polynomial in our field F_q , and we often can, then we are able to extend the field to F_{q^k} , and represent the elements of the extended field as degree $k - 1$ polynomials, $a_0 + a_1x + \dots + a_{k-1}x^{k-1}$. We can represent this compactly as (a_0, \dots, a_{k-1}) , as long as we remember that there may be some very funky arithmetic going on.

Also worth noting is that modular reductions like this (our reduction rule) can be chosen so that they play nicely with the twisting operation.

In practice, large extension fields like $F_{q^{12}}$ are implemented as towers of smaller extensions. That's an implementation aspect, so I've put it in the more practical section below.

The curves

One of the initially non-obvious things about BLS12-381 is that we're really dealing with two curves, not one. Both curves share more-or-less the same curve equation, but are defined over different fields.

The simpler one is over the finite field F_q , which is just the integers mod q . So the curve has points only where the equation $y^2 = x^3 + 4$ has solutions with x and y both integers less than q . Such a point is $(0, 2)$, for example^[3]. We shall call this curve $E(F_q)$.

The other curve is defined over an extension (https://en.wikipedia.org/wiki/Field_extension) of F_q to F_{q^2} (think complex numbers). In this case, the curve equation is slightly modified to be $y^2 = x^3 + 4(1 + i)$ ^[4], and we'll call the curve $E'(F_{q^2})$ ^[5]. We'll explain where this comes from under Twists, below.

As an aside, the curve order of $E'(F_{q^2})$ is vastly bigger than that of $E(F_q)$: The curve equation has many more solutions when the domain is extended to the complex numbers. In fact, the order of E is close to q , and the order of E' is close to q^2 . This is no accident, but a result of the Hasse bound.

(https://en.wikipedia.org/wiki/Hasse%27s_theorem_on_elliptic_curves)

The Subgroups

In this section and the next I'll explain how BLS12-381 ended up having two curve equations rather than one.

A pairing is a bilinear map. This means that it takes as input two points, each from a group of the same order, r . r must be prime, and for security needs to be large. Also, for rather technical reasons, these two groups need to be distinct. Let's call them G_1 and G_2 .

Unfortunately, our simple curve $E(F_q)$ has only a single large subgroup of order r , so we can't define a pairing based solely on $E(F_q)$.

However (<https://web.archive.org/web/20131203082655/https://www.computing.dcu.ie/~mike/tate.html>), if we keep extending the field over which E is defined, it can be proved that we eventually find a curve that has more than one subgroup of order r (in fact, $r + 1$ of them). That is, for some k , $E(F_{q^k})^{[6]}$ contains other subgroups of order r that we can use. One of these subgroups contains only points having a trace of zero^[7], and we choose that subgroup to be G_2 .

This number k , the amount that we need to extend the base field by to find the new group, is called the *embedding degree* of the curve, which in our case is the "12" in BLS12-381. We'll discuss embedding degree more in a moment.

For completeness, note that each of G_1 and G_2 shares with its containing curve the "point at infinity". This is the identity element of the elliptic curve arithmetic group, often denoted \mathcal{O} . For any point P , $P + \mathcal{O} = \mathcal{O} + P = P$.

So, at this point, we have a group G_1 of order r in $E(F_q)$, and we have a distinct group G_2 of order r in $E(F_{q^{12}})$. Yay - we can do pairings!

Twists

But there's another challenge. As discussed earlier, doing arithmetic in $F_{q^{12}}$ is horribly complicated and inefficient. And curve operations need a lot of arithmetic. But it looks like that's what we are stuck with.

Or are we? Well, there's a twist in the tale...^[8]

A twist (<http://indigo.ie/~mscott/twists.pdf>) is something like a coordinate transformation. Rather wonderfully, this can be used to transform our $E(F_{q^{12}})$ curve into a curve defined over a lower degree field that still has an order r subgroup. Moreover, this subgroup has a simple mapping to and from our G_2 group^[9].

BLS12-381 uses a "sextic twist". This means that it reduces the degree of the extension field by a factor of six. So G_2 on the twisted curve can be defined over F_{q^2} instead of $F_{q^{12}}$, which is a huge saving in complexity.

I haven't seen this written down anywhere—but attempting to decode section 3 of [this](https://eprint.iacr.org/2005/133.pdf) (<https://eprint.iacr.org/2005/133.pdf>)—if we find a u such that $u^6 = (1 + i)^{-1}$, then we can define our twisting transformation as $(x, y) \rightarrow (x/u^2, y/u^3)$. This transforms our original curve $E : y^2 = x^3 + 4$ into the curve $E' : y^2 = x^3 + 4/u^6 = x^3 + 4(1 + i)$. E and E' look different, but are actually the same object presented with respect to coefficients in different base fields^[10].

When the twist is done correctly (<http://indigo.ie/~mscott/twists.pdf>), the resulting E' has a subgroup of order r that maps to our G_2 group and vice-versa. So, it turns out that we can work in E' over F_{q^2} for most purposes, and map G_2 back to $E(F_{q^{12}})$ only when required (i.e. while actually computing pairings).

So these are the two groups we will be using:

- $G_1 \subset E(F_q)$ where $E : y^2 = x^3 + 4$
- $G_2 \subset E'(F_{q^2})$ where $E' : y^2 = x^3 + 4(1 + i)$

And that's the story of why BLS12-381 looks like two curves, not one.

Note that coordinates of points in the G_1 group are pairs of integers, and coordinates of points in the G_2 group are pairs of complex integers, so G_2 points take twice the amount of storage, and are more expensive to work with. This leads to interesting implementation tradeoffs.

Pairings

So, what's this pairing thing all about?

As far as BLS12-381 is concerned, a pairing simply takes a point $P \in G_1 \subset E(F_q)$, and a point $Q \in G_2 \subset E'(F_{q^2})$ and outputs a point from a group $G_T \subset F_{q^{12}}$. That is, for a pairing e , $e : G_1 \times G_2 \rightarrow G_T$.

Pairings are usually denoted $e(P, Q)$ and have very special properties. I'm not going to go into all the details—we can pretty much treat them as a black box—but a great introduction is [Vitalik's article](https://vitalik.ca/general/2017/01/14/exploring_ecp.html) (https://vitalik.ca/general/2017/01/14/exploring_ecp.html), and for all the glorious details let me pitch again [Pairings for Beginners](https://www.craigcostello.com.au/s/PairingsForBeginners.pdf) (<https://www.craigcostello.com.au/s/PairingsForBeginners.pdf>).

What we are interested in is that:

- $e(P, Q + R) = e(P, Q) \cdot e(P, R)$, and
- $e(P + S, R) = e(P, R) \cdot e(S, R)$

From this, we can deduce that all of the following identities hold:

- $e([a]P, [b]Q) = e(P, [b]Q)^a = e(P, Q)^{ab} = e(P, [a]Q)^b = e([b]P, [a]Q)^{[11]}$.

This is just what we need when verifying [digital signatures](#).

Embedding degree

We've mentioned the embedding degree several times, and it is significant enough to appear in the name of the curve.

The embedding degree, k , is calculated as the smallest positive integer such that r divides $(q^k - 1)$. So, in the case of BLS12-381, r is a factor of $(q^{12} - 1)^{[12]}$, but not of any lower power.

It turns out that this number gives the smallest field extension F_{q^k} that satisfies the two equivalent conditions:

- F_{q^k} contains more than one subgroup of order r (used for constructing G_2 , see [above](#));
- F_{q^k} contains all the r th roots of unity (used for constructing G_T , see [below](#))

These are the conditions we need to satisfy for pairings to be possible.

The choice of an embedding degree is a balance between security and efficiency (as ever). On the [security](#) front, the embedding degree is also known as the security multiplier: a higher embedding degree makes the discrete logarithm problem harder to solve in G_T . However, a high embedding degree means we have to do field operations in high-degree extensions, such as $F_{q^{12}}$, which is clunky and inefficient. (This is true even when using [twists](#): the maximum available twist is degree six, so the best we can do is to reduce the field extension degree by six. And in any case pairing must be done in the large extension field.)

Embedding degrees of 12 or 24 seem to be a current sweet-spot for many applications. Once again, the embedding degree of BLS12-381 is 12 - it's in the name.

Security level

Security of cryptographic systems is measured in bits

(https://en.wikipedia.org/wiki/Security_level). Informally, I take n -bit security to mean something like, "would need about 2^n operations to break it".

For elliptic curve cryptography, security is all about making the discrete logarithm problem hard. That is, given a point g and a point g^k (in multiplicative group notation), finding k must be infeasible without prior knowledge. That is, it must take at least 2^n operations to achieve this, for $n > 100$ or so in today's terms.

For pairing-friendly curves, the discrete logarithm problem must be hard in each of the three of the groups we are using. Thus, to target n -bit security,

- The prime group order r must be at least $2n$ bits long as there are algorithms such as Pollard's rho algorithm (https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms) that have cost $O(\sqrt{r})$.
- Our extended field F_{q^k} must be large enough not to be vulnerable to methods like the number field sieve (https://en.wikipedia.org/wiki/General_number_field_sieve).

BLS12-381 was intended to offer around a 128 bit security level, based on these criteria, and this was supported by initial analyses. See Table 1.1 in the Taxonomy (<https://eprint.iacr.org/2006/372.pdf>), for example.

However, on closer examination it seems that "the finite extension field of size $3072 = 12 \times 256$ bits is not big enough" (quoting section 2 here (<https://eprint.iacr.org/2019/077.pdf>)), in view of the second criterion above.

According to a report by NCC Group (https://www.nccgroup.trust/globalassets/our-research/us/public-reports/2019/ncc_group_zcash2018_public_report_2019-01-30_v1.3.pdf), citing other sources, the actual security level is probably between 117 and 120 bits (see page 8). They regard this as a perfectly adequate level of security: "The value of reaching '128-bit' [being] mostly psychological". Sean Rowe has also commented on the security level in the light of the original design goals (<https://github.com/zcash/zcash/issues/4065#issuecomment-572202467>).

Cofactor

A subgroup's cofactor is the ratio of the size of the whole group to the size of the subgroup. Normal elliptic curve cryptography requires the cofactor to be very small, usually one (<https://crypto.stackexchange.com/questions/2881/why-would-anyone-use-an-elliptic-curve-with-a-cofactor-1>), in order to avoid small subgroup attacks on discrete logarithms. In pairing-based cryptography, however, this is not the case, and the cofactors of the G_1 and G_2 groups can be truly enormous.

It turns out that, with care, we can have large cofactors and still be secure. Namely, when the cofactors of G_1 , G_2 and G_T contain no prime factors smaller than r . Section 3.2 of [this paper](https://eprint.iacr.org/2015/247.pdf) (<https://eprint.iacr.org/2015/247.pdf>) discusses this in detail. This is *not* the case for BLS12-381, however, and the G_1 and G_2 cofactors both have several small factors. Thus, we have to be mindful of small subgroup attacks in our implementations.

For reference, the cofactors of G_1 and G_2 are as follows.

Group	Cofactor	Equation	Value (h)
G_1	h_1	$(x - 1)^2/3$	0x396c8
G_2	h_2	$(x^8 - 4x^7 + 5x^6 - 4x^4 + 6x^3 - 4x^2 - 4x + 13)/9$	0x5d543

What's the point of all this? Well, multiplying by the cofactor turns out to be a straightforward way to map any arbitrary point on the elliptic curve into the respective subgroup G_1 or G_2 ^[13]. This is important when doing "hash to curve" operations and the like: we first make a point on the curve, and then we map it into the appropriate group by multiplying by the cofactor, so-called cofactor clearing.

Roots of unity

Just a note on roots of unity, because they come up in two completely different and unrelated contexts, which can be confusing.

First, we said that to support zkSnark schemes with this curve, for some biggish n we want to have a 2^n th root of unity in the field F_r (not F_q , note). This is to facilitate efficient fast Fourier transforms for manipulating very large polynomials over the scalar field F_r . From the hexadecimal representation of $r - 1$, it's clearly a multiple of 2^{32} , so there is a 2^{32} th root of unity (2^{32} of them, in fact). Job done, 🍌

Second, and completely unrelated, the effect of the pairing is to map the two points from G_1 and G_2 onto an r th root of unity in $F_{q^{12}}$. These r th roots of unity actually form a subgroup in $F_{q^{12}}$ of order r ^[14], which is the group we call G_T .

Let's briefly revisit our discussion of extending the base field for E to $F_{q^{12}}$, which we did in order to find another subgroup of order r . It also turns out $F_{q^{12}}$ treated as a multiplicative group is the smallest field extension that contains the r th roots of unity in the field, the 12 coming from the embedding degree once again. This is why G_T is defined over $F_{q^{12}}$.

Using curve BLS12-381

This section is a miscellany of things relevant to using BLS12-381 in practice.

BLS digital signatures

Now it's time to introduce the other BLS: Boneh, Lynn and Shacham. (The L is the same L as in BLS12-381; the B and the S are different.)

BLS signatures were introduced back in 2001

(<https://www.iacr.org/archive/asiacrypt2001/22480516.pdf>), a little before the BLS curve family (<https://eprint.iacr.org/2002/088.pdf>) was published in 2002. But, pleasingly, they go hand-in-hand. (BLS signatures can use other curves; BLS curves have uses other than signatures. But it's nice when they come together.)

There's a pretty concise but lucid description of the BLS signature scheme in the draft IETF standard (<https://tools.ietf.org/html/draft-boneh-bls-signature-00>). See also the GitHub repo (<https://github.com/kwantam/draft-irtf-cfrg-bls-signature>).

Private and public keys

The private/secret key (to be used for signing) is just a randomly chosen number between 1 and $r - 1$ inclusive. We'll call it sk .

The corresponding public key (if we're using G_1 for public keys) is $pk = [sk]g_1$, where g_1 is the chosen generator of G_1 . That is, g_1 multiplied by sk , which is g_1 added to itself sk times.

The discrete logarithm problem means that it is unfeasible to recover sk given the public key pk .

Signing

To sign a message m we first need to map m onto a point in group G_2 (if we're using G_2 for signatures). See hashing to the curve, below, for a discussion on ways to do this. Anyway, let's assume we can do this, and call the resulting G_2 point $H(m)$.

We sign the message by calculating the signature $\sigma = [sk]H(m)$. That is, by multiplying the hash point by our secret key.

Verification

Given a message m , a signature σ , and a public key pk , we want to verify that it was signed with the sk corresponding to pk .

This is where pairing comes in. The signature is valid if, and only if,
 $e(g_1, \sigma) = e(pk, H(m))$.

We can confirm this using the properties of pairings:

$$e(pk, H(m)) = e([sk]g_1, H(m)) = e(g_1, H(m))^{(sk)} = e(g_1, [sk]H(m)) = e(g_1, \sigma).$$

Aggregation

A really neat property of BLS signatures is that they can be aggregated

(<https://eprint.iacr.org/2018/483.pdf>) (see also the original paper

(<https://crypto.stanford.edu/~dabo/pubs/papers/aggreg.pdf>)), so that we need only two pairings to verify a single message signed by n parties, or $n + 1$ pairings to verify n different messages signed by n parties, rather than $2n$ pairings you might naively expect to need. Pairings are expensive to compute, so this is very attractive.

It's possible to aggregate signatures over different messages, or signatures over the same message. In the case of Ethereum 2.0 we aggregate over the same message, so for brevity I'm just going to consider that.

To aggregate signatures we just have to add up the G_2 points they correspond to:

$$\sigma_{agg} = \sigma_1 + \sigma_2 + \dots + \sigma_n. \text{ We also aggregate the corresponding } G_1 \text{ public key points } pk_{agg} = pk_1 + pk_2 + \dots + pk_n.$$

Now the magic of pairings means that we can just verify that

$$e(g_1, \sigma_{agg}) = e(pk_{agg}, H(m)) \text{ to verify all the signatures together with just two pairings.}$$

Rogue key attacks

As noted in section 1.1 here (<https://eprint.iacr.org/2018/483.pdf>), when aggregating signatures over the same message, we need to take care of a possible "rogue public key attack".

Say your public key is pk_1 , and I have a secret key, sk_2 . But instead of publishing my true public key, I publish $pk'_2 = [sk_2]g_1 - pk_1$ (that is, my real public key plus the inverse of yours). I can sign a message $H(m)$ with my secret key to make

$\sigma = [sk_2]H(m)$. I then publish this claiming that it is an aggregate signature that both you and I have signed, and that the aggregate public key is $pk_{agg} = pk_1 + pk'_2$.

Now, when verifying, my claim checks out: it looks like you signed the message when you didn't: $e(g_1, \sigma) = e(g_1, [sk_2]H(m)) = e([sk_2]g_1, H(m)) = e(pk_1 + pk'_2, H(m))$.

One relatively simple defence against this—the one we are using in Ethereum 2.0—is to force validators to register a “proof of possession” of the private key corresponding to their claimed public key. You see, the attacker doesn't have the sk'_2 corresponding to pk'_2 . This can be done simply by getting the validator to sign its public key on registration: if the signature validates with that public key then all is well.

More complex schemes, not requiring proof of knowledge of the secret key (KOSK), are available (<https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html>).

Swapping G_1 and G_2

For the purposes of digital signatures, the G_1 and G_2 groups are interchangeable. We can choose our public keys to be members of G_1 and our signatures to be members of G_2 , or vice versa.

The trade-offs are execution speed and storage size. G_1 has small points and is fast; G_2 has large points and is slow. BLS12-381 was initially designed to implement Zcash, and for performance reasons they chose to use G_1 to represent signatures and G_2 to represent public keys.

With respect to Zcash, most other implementations are “reversed”. In Ethereum 2.0 we use G_1 for public keys: for one thing, aggregation of public keys happens much more often than aggregation of signatures; for another, public keys of validators need to be stored in state, so keeping the representation small is important. Signatures, then, are G_2 points.

Point compression

(Note that sometimes, the twisting operation is referred to as point compression - that's something completely different to what we're discussing here.)

For storing and transmitting elliptic curve points, it is common to drop the y -coordinate. This halves the amount of data. For BLS12-381, G_1 points are reduced from 96 bytes (2×381 bits-rounded-to-bytes) to 48 bytes, and G_2 points are reduced from 192 bytes to 96 bytes.

Any elliptic curve point can be regenerated from the x coordinate by using the relevant curve equation, E or E' . For any valid x coordinate on the curve, y is either zero or has two possible values that are the negative of each other: $y = \pm\sqrt{x^3 + 4}$ for G_1 , and analogously for G_2 .

Since field elements are 381 bits, and 48 bytes is 384 bits, we have a few bits to spare for flags. The most important is a flag to show which of the y values the point corresponds to (positive or negative). Another bit is used to signal whether this is the point at infinity (which has many possible representations). A third is simply to indicate whether this is a compressed or uncompressed representation, though context should handle this in practice.

For both G_1 and G_2 , about half of x values are not on the curve. In this case, the point is conventionally decoded to the point at infinity. But unless the infinity flag is set—in which case we would not have attempted to decode the point—this is an error condition.

The specific details of how the flag bits and x values are encoded is [here](https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#serialization)

(https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#serialization).

Subgroup membership checks

When dealing with any point with an unknown origin, whether it comes to us compressed or uncompressed, it's important that we check that it lies in the correct subgroup. The point decompression described above only results in a point on the curve; we don't know whether it lies in the appropriate G_1 or G_2 .

The main issue seems to be that both $E(F_q)$ and $E'(F_{q^2})$ contain small subgroups (you can see this by factoring the cofactors, eg. with [this](https://www.alpertron.com.ar/ECM.HTM) (<https://www.alpertron.com.ar/ECM.HTM>) tool). Inadvertantly working with points in these small subgroups could lead to vulnerabilities, as discussed in [this paper](https://eprint.iacr.org/2015/247.pdf) (<https://eprint.iacr.org/2015/247.pdf>).

Subgroup checks are easy in principle: simply multiply our point by r . For points in G_1 or G_2 this will result in the respective points at infinity; for points outside the groups, it won't.

Unfortunately, this is slow in practice, especially for G_2 , since r is so large. As an alternative, there are [new techniques](https://eprint.iacr.org/2019/814.pdf) (<https://eprint.iacr.org/2019/814.pdf>) making use of endomorphisms for performing faster subgroup checks.

Generators

G_1 and G_2 are cyclic groups of prime order, so any point (except the identity/point at infinity) is a generator. Thus, picking generators is just a matter of convention.

Generator points for G_1 and G_2 are specified in decimal [here](https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#generators)

(https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#generators) and the same points in hexadecimal [here](https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2)

(<https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2>).

These were chosen [as follows](#)

(https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#generators):

The generators of G_1 and G_2 are computed by finding the lexicographically smallest valid x-coordinate, and its lexicographically smallest y-coordinate and scaling it by the cofactor such that the result is not the point at infinity.

By my calculations, with h_1 and h_2 the respective group [cofactors](#), this makes the G_1 generator $g_1 = [h_1]p_1$, with p_1 as follows,

$p_1 = (0x04, 0x0a989badd40d6212b33cffc3f3763e9bc760f988c9926b26da9dd85e9284)$

and the G_2 generator $g_2 = [h_2]p_2$, with p_2 as follows,

$p_2 = ([0x02, 0x00], [0x013a59858b6809fca4d9a3b6539246a70051a3c88899964a42bc])$

(I think “lexicographically smallest” means treating all numbers in the base field as non-negative, and just taking the smaller one, prioritising real parts over imaginary parts.)

Final exponentiation

Calculation of a pairing has two parts: the Miller loop and the final exponentiation. Both are quite expensive, but there’s a nice hack you can do to reduce the impact of the final exponentiation.

Normally, we calculate two full pairings in order to perform signature verification, to check whether $e(g_1, \sigma) = e(pk, H(m))$.

If we denote as $e'(\cdot, \cdot)$ the pairing without the final exponentiation, then for, some x , we are checking whether $e'(g_1, \sigma)^x = e'(pk, H(m))^x$. (x happens to be $(q^k - 1)/r$, which is huge.)

We know how to multiply in group G_T , so we can reorganise this as a check whether $(e'(-g_1, \sigma)e'(pk, H(m)))^x = 1$. (We can negate any one of the points: the magic of pairings makes this equivalent to taking the inverse in G_T .)

So, to verify a signature, we do the two Miller loops, one with a negated input value, multiply the results and then do a single final exponentiation. If the result is unity in G_T then our pairings match. This ought to give a worthwhile speedup.

Hashing to the curve

To calculate a digital signature over a message, we first need to transform an arbitrary message (byte string) to a point on the G_2 curve (if we are using G_2 for signatures). There are many ways to do this, with varying degrees of efficiency and security.

Hash and check

The initial implementation in Eth2 was “hash-and-check”. This is very simple.

1. Hash your message to an integer modulo q
2. Check if there is a point on the curve with this x -coordinate (real part x , imaginary part 0). If not, add one and repeat^[15].
3. We have a point on the curve! Multiply by the G_2 cofactor to convert it into a point in G_2 .

About half the points that we try will result in a point on the curve, so this is not constant time—we don’t know how many iterations it will take to find one. In one sense it doesn’t matter: all the information is public, so we’re not leaking anything. However, it does open up a griefing attack. An attacker could pre-calculate messages that take a very long time to find a point (1 in one million messages will take 20 tries, for example) and slow us down considerably.

Simplified SWU map

We are now adopting a better approach which is described in [this paper](https://eprint.iacr.org/2019/403.pdf) (<https://eprint.iacr.org/2019/403.pdf>) and used in the new (draft) IETF standard for hashing to curves (<https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05>). As before (but a bit differently to ensure a uniform distribution of output points) we first create a field point by hashing the message mod q .

Now we use a special map (the SWU map) that is guaranteed to translate that field point into a valid point on an elliptic curve. For technical reasons, this is *not* the curve $E'(F_{q^2})$, but a curve isogenous (<https://www.johndcook.com/blog/2019/04/21/what-is-an-isogeny/>) to it (i.e.

having the same number of points). We then use another map (3-isogeny) to transfer this to a point on $E'(F_{q^2})$. Finally we use cofactor clearing to end up with a point in G_2 .

You can take a look at my implementation of this in Java

(<https://github.com/PegaSysEng/artemis/pull/898>), based on the reference code in Python

(https://github.com/algorand/bls_sigs_ref/tree/master/python-impl). The idea is for this approach to be generally adopted to enhance the interoperability of blockchains.

Cofactor clearing

We discussed multiplying by the cofactor as a way to make an arbitrary point on E or E' into a point in G_1 or G_2 respectively. This is useful when hashing to the curve, for example.

The G_2 co-factor is *huge*, so multiplying by it is slow. However, there are faster ways (<https://eprint.iacr.org/2017/419.pdf>) to map curve points into G_2 using an endomorphism (a map of a group to itself). This features in the new standard (<https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05>) (see section 7).

The endomorphism we want to use was subject to a patent

(<https://patents.google.com/patent/US7110538B2/en>), but this has now expired everywhere.

As a workaround to the patent, instead of multiplying by the G_2 cofactor, the standard suggests multiplying by an effective cofactor (see section 8.9.2

(<https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-05#section-8.9.2>) for the value) which gives the same result as the endomorphism. The effective cofactor is *even larger* than the G_2 cofactor, but the multiplication can be implemented using an addition chain (<https://github.com/PegaSysEng/teku/blob/55d04f87b422112312f79c1b4d662b3d58e3ca74/bls/src/main/java/tech/pegasys/teku/bls/impl/mikuli/hash2g2/Chains.java#L569>) as an optimisation.

The idea is that, now that the patent has expired, the endomorphism can be just dropped in as a replacement for the effective cofactor multiplication.

Extension towers

Recall our discussion of field extensions? In practice, rather than implementing a massive 12th-degree extension directly, it is more efficient to build it up from smaller extensions: a tower of extensions (<https://eprint.iacr.org/2009/556.pdf>).

For BLS12-381, the $F_{q^{12}}$ field is implemented as a quadratic (degree two) extension, on top of a cubic (degree three) extension, on top of a quadratic extension of F_q .

As long as the modular reduction polynomial (our reduction rule) is irreducible (can't be factored) in the field being extended at each stage, then this all works out fine.

Specifically

(https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md):

1. F_{q^2} is constructed as $F_q(u)/(u^2 - \beta)$ where $\beta = -1$.
2. F_{q^6} is constructed as $F_{q^2}(v)/(v^3 - \xi)$ where $\xi = u + 1$.
3. $F_{q^{12}}$ is constructed as $F_{q^6}(w)/(w^2 - \gamma)$ where $\gamma = v$

Interpreting these in terms of our previous explanation:

1. We write elements of the F_{q^2} field as first degree polynomials in u , with coefficients from F_q , and apply the reduction rule $u^2 + 1 = 0$, which is irreducible in F_q .
 - an element of F_{q^2} looks like $a_0 + a_1u$ where $a_j \in F_q$.
2. We write elements of the F_{q^6} field as second degree polynomials in v , with coefficients from the F_{q^2} field we just constructed, and apply the reduction rule $v^3 - (u + 1) = 0$, which is irreducible in F_{q^2} .
 - an element of F_{q^6} looks like $b_0 + b_1v + b_2v^2$ where $b_j \in F_{q^2}$.
3. We write elements of the $F_{q^{12}}$ field as first degree polynomials in w , with coefficients from the F_{q^6} field we just constructed, and apply the reduction rule $w^2 - v = 0$, which is irreducible in F_{q^6} .
 - an element of $F_{q^{12}}$ looks like $c_0 + c_1w$ where $c_j \in F_{q^6}$.

This towered extension can replace the direct extension as a basis for pairings, and when well-implemented can save a huge amount of arithmetic when multiplying $F_{q^{12}}$ points. See [Pairings for Beginners](https://www.craigcostello.com.au/s/PairingsForBeginners.pdf) (<https://www.craigcostello.com.au/s/PairingsForBeginners.pdf>) section 7.3 for a full discussion of the advantages.

Coordinate systems

Finding the inverse of a field element (i.e. division) is an expensive operation, so implementations of elliptic curve arithmetic try to avoid it as much as possible. It helps if we choose the right coordinate system for representing our points.

Affine coordinates

Affine coordinates are the traditional representation of points with just an (x, y) pair of coordinates, where x and y satisfy the curve equation. This is what we normally use when storing and transmitting points.

However, it is not always the most efficient form to use when actually working with points, and there are two other schemes I'm aware of that are used for BLS12-381.

The basic idea is to represent the coordinate using notional fractions, reducing the number of actual division operations needed. To do this, a third coordinate is introduced and we use (X, Y, Z) for the internal representation of a point. Like our familiar fractions, there are many representations of the same value, all corresponding to a single actual value ($\frac{1}{2}, \frac{3}{6}, \frac{197}{394}$ are all the same number).

The two systems I know of in use for BLS12-381 are Standard Projective coordinates and Jacobian coordinates.

Standard Projective coordinates

The Standard Projective coordinate

(https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Standard_Projective_Coordinates) point (X, Y, Z) represents the Affine coordinate point $(X/Z, Y/Z)$.

These are also called homogeneous projective coordinates because the curve equation takes on the homogeneous form $Y^2Z = X^3 + 4Z^3$. Points become straight lines through the origin in (X, Y, Z) space, with the Affine point being the intersection of the line with the plane $Z = 1$. Figure 2.10 in PfB

(<https://www.craigcostello.com.au/s/PairingsForBeginners.pdf>) gives a nice illustration.

Standard Projective coordinates are used by the Apache Milagro (<https://milagro.apache.org/>) BLS12-381 library under the hood.

Jacobian coordinates

A different kind of projective coordinates are Jacobian coordinates

(https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Jacobian_Coordinates). In this scheme, the Jacobian point (X, Y, Z) represents the Affine point $(X/Z^2, Y/Z^3)$. The curve equation becomes $Y^2 = X^3 + 4Z^6$.

The sample code (https://github.com/algorand/bls_sigs_ref/tree/master/python-impl) for the constant-time hash-to-curve uses Jacobian coordinates under the hood.

Note that, in both schemes, the easiest way to import the Affine point (x, y) is to map it to $(x, y, 1)$.

Resources and further reading

There are *lots* of references linked in the above, and I'm not going to repeat many here. I'll just pick out a few particularly useful or interesting things.

Useful reference material:

- The original (<https://electriccoin.co/blog/new-snark-curve/>) BLS12-381 announcement
- Concise (https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md) description of the parameters and serialisation
- Draft IETF standard (<https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html#rfc.section.4.2.2>)
- A tl;dr version of the above, BLS for Busy People (<https://gist.github.com/hermanjunge/3308fbd3627033fc8d8ca0dd50809844>)


In general, implementations of pairing libraries tend to be highly optimised and/or very generic (supporting many curves) which makes them quite hard to learn from. The Noble BLS12-381 (<https://github.com/paulmillr/noble-bls12-381>) library in JavaScript/TypeScript by Paul Miller is definitely among the easier to follow.

Finally, a couple of fun and interesting reads:

- This brand new whitepaper on Curve9769 (<https://github.com/pornin/curve9767/raw/master/doc/curve9767.pdf>) is not directly relevant to BLS12-381, but is a well-written and wonderful exploration of the joys and pains of designing and implementing an elliptic curve (not a pairing-friendly one in this case).
- Pairings are not dead, just resting (<https://ecc2017.cs.ru.nl/slides/ecc2017-aranha.pdf>). A great overview presentation. Some BLS12-381 things.

That's all, folks!

Many thanks to my esteemed colleagues Olivier Bégassat, Thomas Piellard, Herman Junge, Błażej Kolad, and Gus Gutoski for review, sanity check and comments. (I made some changes since you last saw it, guys - any screw-ups are all my own.)

 Ben Edgington (https://twitter.com/benjaminion_xyz), ben@benjaminion.xyz

PegaSys (<https://pegasys.tech/>), ConsenSys (<https://consensys.net/>)

1. I studied mathematics many years ago, but diligently shirked anything to do with pure maths, including group theory. I regret that now. Anyway, this will not be too technical, but I'm also not an expert, so might get some things wrong and will be a bit hand-wavy in general. In case it's not obvious, I am not a cryptographer. ↩
2. Our rule is "an extension field modular reduction" (terminology from [here](https://www.emsec.ruhr-uni-bochum.de/media/crypto/veroeffentlichungen/2015/03/26/crypto98rc9.pdf) (<https://www.emsec.ruhr-uni-bochum.de/media/crypto/veroeffentlichungen/2015/03/26/crypto98rc9.pdf>)). ↩
3. Another point on $E(F_q)$ is
(0x04,0x0a989badd40d6212b33cffc3f3763e9bc760f988c9926b26da9dd85e928483446346b8ed00e1de5d5ea93e354abe706c). On average about half of x values result in a point on the curve, and for most of those both (x, y) and $(x, -y)$ are on the curve (for some, $y = 0$). You soon get used to these ridiculously big numbers. ↩
4. Sometimes u is used rather than i here, with $u^2 + 1 = 0$. I'm using i . ↩
5. Here's a point on the E' curve: (1+i,
0x17faa6201231304f270b858dad9462089f2a5b83388e4b10773abc1eef6d193b9f
ce4e8ea2d9d28e3c3a315aa7de14ca + i *
0xcc12449be6ac4e7f367e7242250427c4fb4c39325d3164ad397c1837a90f0ea1a5
34757df374dd6569345eb41ed76e) ↩
6. Note that we lost the ' on E here - this is the original curve $y^2 = x^3 + 4$, but now defined over F_{q^k} . ↩
7. The "trace zero subgroup" qualifies as an obscure incantation. Basically, the trace of a point is $\sum_{i=0}^{k-1} (x^{q^i}, y^{q^i})$, where $k=12$ in our case. Understanding this involves stuff like the Frobenius endomorphism, and that rabbit hole goes deep. ↩
8. 😊 Please forgive me. ↩
9. ...because we previously selected the trace zero subgroup. [Pairings for Beginners](https://www.craigcostello.com.au/s/PairingsForBeginners.pdf) (<https://www.craigcostello.com.au/s/PairingsForBeginners.pdf>) dives into the details on this. ↩

10. Thank you to my reviewers for this insight ↵
11. $[a]P$ is multiplication of the point P by the scalar a . That is, adding P , a times. Traditionally, the group operation in G_1 and G_2 is represented additively, and in G_T multiplicatively. ↵
12. Numbers in this world are truly enormous. The number of times r divides $(q^{12} - 1)$ is 1299 digits long in decimal. This number is actually used in the final exponentiation when computing pairings. ↵
13. This is easy to see. The subgroup G has order r , and its cofactor is h , such that $hr = n$, the order of the whole elliptic curve group. Consider an arbitrary element P of the elliptic curve group. We have $\mathcal{O} = [n]P = [r]([h]P)$. Thus, $[h]P \in G$. While not specific to BLS12-381, here is an excellent article (<http://loup-vaillant.fr/tutorials/cofactor>) about cofactor clearing. ↵
14. This is a general property of roots of unity in multiplicative groups, not special to elliptic curves or pairings ↵
15. This should have been "increment the message and go back to one" on failure, which is more secure. ↵