YAFA-108/146: Implementing Ed25519-Embedding Cocks-Pinch Curves in Arkworks-rs

Rami Akeela
rami@dzk.org
DZK Labs

Weikeng Chen
weikeng@dzk.org
DZK Labs

Abstract

This note describes two pairing-friendly curves that embed ed25519, of different bit security levels. Our search is not novel; it follows the standard recipe of the Cocks-Pinch method. We implemented these two curves on `arkworks-rs`. This note is intended to document how the parameters are being generated and how to implement these curves in `arkworks-rs` 0.4.0, for further reference.

We name the two curves as YAFA-108 and YAFA-146:

- YAFA-108 targets at 108-bit security, which we parameterized to match the 103-bit security of BN254
- YAFA-146 targets at 146-bit security, which we parameterized to match the 132-bit security of BLS12-446 or 123-bit security of BLS12-381

We use these curves as an example to demonstrate two things:

- The "elastic" zero-knowledge proof, Gemini (EUROCRYPT '22), is more than being elastic, but it is ***more curve-agnostic and hardware-friendly***.
- The cost of nonnative field arithmetics can be drastic, and the needs of ***application-specific curves*** may be inherent. This result serves as evidence of the necessity of EIP-1962, and the insufficiency of EIP-2537.

The implementation and the associated Sage scripts can be found on our GitHub:

`https://github.com/DZK-Labs/ark-yafa`

Key Experiment Results[1]

(on a MacBook Pro, see Section 8 for the setup)

Proof systems	Curves			
	BN254	BLS12-381	YAFA-108	YAFA-146
Groth16 [Gro16]	52.3 s	59.6 s	⊘	⊘
Gemini [BCHO22]	579 s	874 s	3.03 s	3.37 s

Table A: **Proving time of one scalar multiplication on ed25519 in different proof systems and curves.** One can see that Groth16 does not work with YAFA, and verifying one scalar multiplication is already close to one minute. Gemini, which has a universal setup instead of a circuit-specific setup in Groth16, is much slower than Groth16. However, if we do the same computation over the YAFA curves, it is $172\times$ to $288\times$ faster compared with Gemini on the other curves, and $16\times$ to $20\times$ faster than Groth16 on the other curves. However, this comparison may be unfair to YAFA—a circuit-specific setup is a limitation because a setup ceremony for a large circuit is impractical. One may have to resort to another universal setup proof system like Marlin [CHMMVW20], and Marlin and Gemini exhibit similar performance.

Proof systems	Curves			
	BN254	BLS12-381	YAFA-108	YAFA-146
Groth16 [Gro16]	1987323	1987323	⊘	⊘
Gemini [BCHO22]	2747859	2747859	1520	1520

Table B: **Number of constraints in the R1CS constraint system when dealing with one scalar multiplication.** Both Groth16 and Gemini use R1CS, so we use the number of constraints to help understand where the cost comes from. One can see that when nonnative field arithmetic is used, there will be millions of constraints. When the computation can be done natively, as in the case of YAFA, the number of constraints is much lower. Note that the reason why the numbers stay the same between BN254 and BLS12-381, or between YAFA-108 and YAFA-146 is that their scalar field moduli have the same number of bits, and the nonnative field gadget will use the same strategy. Readers may wonder why Gemini uses much more constraints than Groth16, despite that they use the same nonnative gadget in `arkworks-rs`. This is because the gadget automatically detects if the proving cost depends on the number of constraints, or the weight of the R1CS matrices, and uses different optimization goals when generating the parameters for nonnative field simulation. Groth16 is the former, but Gemini is the latter, so more constraints are used to reduce the weight.

[1]We want to put a front notice that Groth16 in `arkworks-rs` takes time to synthesize the constraint system and inline the linear combinations. We may update our experiment results once we remove this overhead.

Ed25519-embedding curve	Batch size (large)		
	10	20	30
YAFA-108	22.0 s	36.9 s	56.1 s
– per scalar mult	2.20 s	1.85 s	1.87 s
YAFA-146	22.0 s	38.2 s	57.0 s
– per scalar mult	2.20 s	1.91 s	1.87 s

Table C: **The amortized proving time for one scalar multiplication on ed25519 with different sizes of the batch with YAFA curves in Gemini.** Recall that Gemini has two versions: time-efficient and space-efficient. We use the time-efficient version. In this case, Gemini proving time roughly grows linearly to the weight of the R1CS matrices. The percentage of the sublinear part will decline when we increase the batch size, so we can see that the amortized proving time is converging. One may be surprised that YAFA-146 is not slower than YAFA-108. This is actually reasonable and is by design. Both YAFA curves have moduli of about 574 bits, and the difference is in the embedding degree, which the prover does not need to worry about because the prover does not do any pairing. This is indeed an example of a trade-off between prover efficiency and verifier efficiency—if one chooses a reasonable but high embedding degree, we may be able to do with curves with smaller moduli, probably one less limb. This significantly helps the prover who does multiscalar multiplication, though the verifier's work may increase.

General-purpose curve	Batch size (small)		
	2	4	6
BN254	108 s	382 s	677 s
– per scalar mult	54 s	96 s	113 s
BLS12-381	123 s	450 s	686 s
– per scalar mult	62 s	112 s	114 s

Table D: **The amortized proving time for one scalar multiplication on ed25519 with different sizes of the batch with other curves in Groth16.** Due to nonnative field arithmetics, the overhead of proof generation is high. One can see that a large batch is not worthy because the per-scalar-multiplication cost is increasing. This is expected because Groth16 is quasilinear. We want to remind readers that Groth16 requires a circuit-specific setup. To create a circuit that verifies one hundred scalar multiplications, one needs a setup of size 2^{27}, which is challenging. The prover might have to instead choose a universal-setup proof system.

Contents

1 Introduction

Our motivation to find ed25519-embedding curves comes from the needs to build SNARK-Bridge[2] with chains that use ed25519[3] for signatures in the consensus. Today, the standard EVM only has precompiled contracts for BN254. We expect that soon, with EIP-2537, BLS12-381 will also have precompiled contracts in EVM. However, the more generic proposal, EIP-1962 (which plans to support most families of pairing-friendly curves), has hardly moved forward even though it was initially approved. In this note, we focus on the scalar multiplication part of the signature verification. The SHA-512 part should be handled by another proof system that is not discussed in this note. To efficiently verify such scalar multiplication, we have two types of solutions.

The first type, "the nonstop type," is to use a proof system on BN254 to directly verify the scalar multiplication. This can be expensive because the computation is nonnative and requires field simulation. This is similar to taking a nonstop flight—the price may be higher, but it is one-shot.

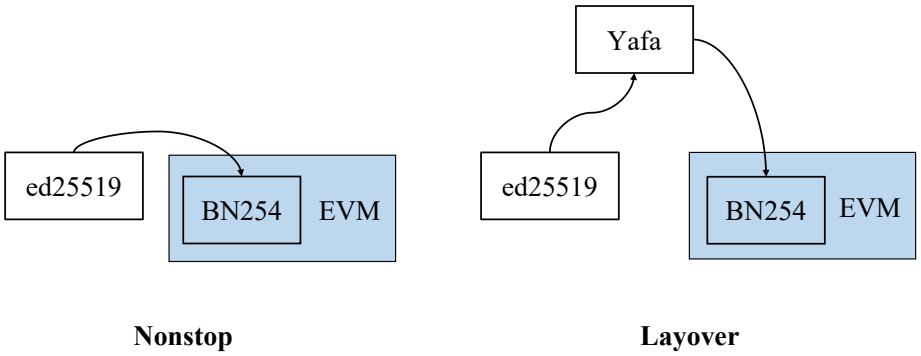


Figure 1: Two strategies of verifying ed25519 scalar multiplication (without EIP-1962).

The second type, "the layover type," uses more than one proof system. There are many possible combinations. For example, one can use Plonky2 [plo] to recursively compose proofs about SNARK-EVM and then compress the final proof using pairing-based proofs. This has an advantage over "the nonstop type" only if the amount of work to verify scalar multiplication of ed25519 in the nonnative field is significantly higher than having BN254 verify a succinct proof for this curve, which is also nonnative. This is similar to taking a layover flight—we may be able to reduce the price if we travel for a long distance and are willing to take a few more steps. The concern, however, is the high cost of verifying nonnative pairing, which can be seen as "the last mile."

In sum, both solutions suffer from the overhead of nonnative field arithmetics. For "the nonstop type" it is the nonnative scalar multiplication of ed25519. For "the layover type" it is the nonnative pairing. The challenge is that we are unlikely going to get the best of both worlds:

[2]We decide to use SNARK-Bridge instead of zk-Bridge because we feel that it is beneficial in the long run to adopt a less misleading name when we explain this to the community.

[3]We would like to remind readers that ed25519 is a different curve from curve25519, although they are "birationally equivalent". For this reason, we will use ed25519 instead of curve25519 in the rest of this note. See this GitHub issue for more information: `https://github.com/arkworks-rs/curves/issues/115`.

- proving scalar multiplication of ed25519 *without* nonnative field arithmetics
- making this proof verifiable by EVM *without* nonnative field arithmetics

A seemingly working solution is to find a chain of ordinary curves in which a curve's scalar field is the previous curve's base field, i.e., starting from the origin $E_1[r = p_1, q = p_2]$ which is ed25519, finding a way to the destination $E_{n-1}[p_{n-1}, p_n]$ which is BN254. This is similar to flying a plane, in which we find the shortest path from the departure airport to the destination airport, which we illustrate as follows:

$$E_1[p_1, p_2] \to E_2[p_2, p_3] \to E_3[p_3, p_4] \to \cdots \to E_{n-1}[p_{n-1}, p_n]$$

This search, however, will fail. By Hasse's theorem, we know that:

$$|\#E_i - p_{i+1} - 1| < 2 \cdot \sqrt{p_{i+1}}$$

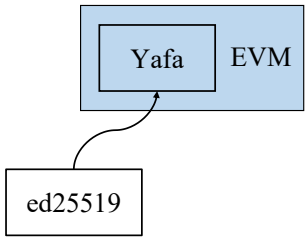
and p_i divides $\#E_i$. It suggests that going down this chain, it is easy to make the next prime larger by using a cofactor, but it is difficult to make the next prime much smaller as Hasse's theorem shows. To build such a chain, the number of intermediate curves that we need is at least:

$$\frac{p_1 - p_n}{2 \cdot \sqrt{p_1}} \approx 2^{126}$$

As a result, it appears that nonnative field arithmetic is unavoidable. This implies an inherent reason why verifying the scalar multiplication on another curve can often be slow.

Note that FRI protocols [BSBHR18], which actually do not work well here because ed25519 does not have good FFT space, are also not helping since it does not help us jump between fields.

A "new" hope: EIP-1962. To some extent, the problem comes from EVM, and we should ideally solve it in EVM. EIP-1962 once gave us some hope. This EIP plans to support common *families* of elliptic curves—an infinite number of curves. This EIP was tentatively accepted for the Berlin release, but it did not move forward.



Helicopter

Figure 2: A strategy of verifying ed25519 signatures (with EIP-1962).

The flexibility to pick an application-specific curve is beneficial for performance. For example, with EIP-1962, we can have a third type of the solution, which we call "the helicopter type," in

which the proof that we provide to the EVM does not necessarily be over BN254 or BLS12-381. It allows us to remove the second step in "the layover type," which avoids all the heavy computation.

Rest of the note. We have discussed our motivation. The rest of the paper is an engineer's note for how to generate the parameters of the YAFA curves and implement them in `arkworks-rs` and how we conduct the experiments. There are no technical insights in these sections, and readers are advised to just skim through them.

2 Examples

In this section we keep a list of adoption of the techniques that are described in this paper.

Verifiable time-lock encryption: zk-timelock [Yal]. Timofey Yaluhin, a researcher in ChainSafe, has implemented a design of verifiable time-lock encryption over the BLS12-381 curve, which is the native curve used in the `drand` threshold network [Dra], which is for distributed randomness generation. To be able to efficiently verify BLS12-381 signatures, Timofey creates the YT6-776 curve (where YT means "Yeti") which embeds BLS12-381. The implementation of the curve as well as the arithmetic circuits for verifiable time-lock encryption have been open-sourced.

Recursive verification for StarkNet. Mikerah Quintyne-Collins from HashCloak [Has] has been developing proof compression and recursive verification for StarkNet. Her team found a curve over the STARK field, called Lokum, and they plan to open-source soon.

3 Reference materials

We would like to share the list of resources that help this work.

Cocks-Pinch method. We use the `ecfactory` library [Ecf], which implements the Cocks-Pinch method, to generate the curves. The Cocks-Pinch method [CP01] is a way to find pairing-friendly curves with a prescribed order of a prime-order subgroup. The search is efficient, but it gives a curve whose base field modulus is twice the size of the scalar field modulus (also known as $\rho \approx 2$), for which we do not have a better solution today. There are other ways to construct pairing-friendly embedding curves, though it requires more effort [HG20].

Twist in the pairing. The YAFA-146 curve has an embedding degree of 12, and the pairing is defined with $\mathbb{G}_1$ over $\mathbb{F}_q$, $\mathbb{G}_2$ over $\mathbb{F}_{q^2}$, and $\mathbb{G}_T$ over $\mathbb{F}_{q^{12}}$. For this to happen, the curve needs to meet certain conditions. Michael Scott has a note [Sco] on twists for the pairing-friendly curves.

Field selection. This problem of verifying scalar multiplication in ed25519 is closely related to field selection. On the one hand, we want to choose a matching field to avoid nonnative computation. On the other hand, we need to ensure that the EVM can efficiently verify the final proof. There are also other considerations about the choice of the field, such as whether it has a handy FFT space. Gautam Botrel from Delendum Ventures has an article [Fie] that describes the general considerations of field selection today.

Miller loop. We implemented the Tate and ate pairings following the implementation of pairing over Cocks-Pinch curve and over BLS curve in `arkworks-rs` [Ark] as well as in Aurore Guillevic's Sage scripts [Zkc].

Final exponentiation. The last step of pairing is final exponentiation. For embedding degree of $k = 6$, we power the result of the Miller loop by $(q^6 - 1)/r$, and for $k = 12$ we do $(q^{12} - 1)/r$. In practice, we implement the final exponentiation by factoring the polynomial and use the Frobenius endomorphism for $x \to x^q, x^{2q}, ...$ and combine the result together.

Bit security. Informally speaking, for a pairing from $E(\mathbb{F}_q) \times E'(\mathbb{F}_{q^2}) \to \mathbb{F}_{q^k}$, the last part $\mathbb{F}_{q^k}$ is often the weakest link. In this note we estimate the bit security in a rough manner, in that we do not look into the concrete parameters one need to use in the number field sieve algorithm [AFKLO07; BD19]. One can analyze the bit security level more precisely by taking those parameters into consideration [BD19; GMT20; GS21], but it is beyond the scope of this note.

4 Bit security

We can obtain a rough estimate of the bit security of a pairing-friendly curve by estimating the cost of special number field sieve for $\mathbb{F}_{q^k}$. The cost can be estimated using the following formula.

$$\mathsf{cost}(\mathbb{F}_{q^k}) = 2^{\kappa} \cdot e^{\sqrt[3]{c/9 \cdot \ln(q^k) \cdot (\ln(\ln(q^k)))^2}}$$

with the constants $c = 32$ and $\kappa = -7$ based on the work by Aoki, Franke, Kleinjung, Lenstra and Osvik [AFKLO07]. Note that our estimation is rough as it ignores the inner structure of these curves, despite the fact that the parameters are special enough to be pairing-friendly. For a more precise analysis, readers can look at [BD19; GMT20; GS21].

Estimated bit security of existing curves in arkworks-rs for comparison. We now use this formula to estimate the bit security of the pairing-friendly curves implemented in `arkworks-rs`.

- BLS12-377, in which $\log q = 377$ and $k = 12$, has 122-bit security.
- BLS12-381, in which $\log q = 381$ and $k = 12$, has 123-bit security.
- BN254, in which $\log q = 254$ and $k = 12$, has 103-bit security.
- BW6-761, in which $\log q = 761$ and $k = 6$, has 123-bit security.
- CP6-782, in which $\log q = 782$ and $k = 6$, has 124-bit security.
- MNT4-298, in which $\log q = 298$ and $k = 4$, has only 67-bit security.
- MNT6-298, in which $\log q = 298$ and $k = 6$, has only 81-bit security.
- MNT4-753, in which $\log q = 753$ and $k = 4$, has 102-bit security.
- MNT6-753, in which $\log q = 753$ and $k = 6$, has 122-bit security.

Estimated bit security of YAFA-108 and YAFA-146. Both curves have base field moduli of 574 bits. YAFA-108 has an embedding degree of 6, and YAFA-146 has an embedding degree of 12. Using the formula above, we have:

- YAFA-108, in which $\log q = 574$ and $k = 6$, has 108-bit security (the result from the formula above is ≈ 108.53).
- YAFA-146, in which $\log q = 574$ and $k = 12$, have 147-bit security (the result from the formula above is ≈ 147.08), which would be sufficient for a targeted security of 146.

Note that using a higher embedding degree comes with a cost. Though it does not affect the prover as it only works with $\mathbb{G}_1$, the verifier will pay an extra cost to do a pairing.

5 Implementation tricks

By popular request, we want to share some implementation tricks that are essential for a successful attempt in finding a good Cocks-Pinch curve.

Increasing the base field modulus to reduce the embedding degree. To achieve 128-bit security, for the base field, there are two options: (1) increasing the embedding degree or (2) increasing the base field modulus. Both methods make $\mathbb{G}_T$ larger, which is the goal. And both methods can work together.

Generally, the original Cocks-Pinch method gives a curve whose base field modulus is about twice the scalar field modulus. Usually, this modulus is not sufficient for 128-bit security if we want to choose a small embedding degree such as 6. Increasing the embedding degree has a side effect that it increases the verifier overhead, but this is not the biggest issue.

The most challenging part is that only certain embedding degrees allows us to use *twists* [Sco]. The use of twists allows us to have $\mathbb{G}_2$ that is particularly smaller, for example, in BLS12-381, $\mathbb{G}_2$ is defined over $\mathbb{F}_{q^2}$. If we do not use the twist, it would need to be defined over $\mathbb{F}_{q^{12}}$.

It is ideal if the Cocks-Pinch method can give us a larger base field modulus. An implementation trick that we provide is to modify the Cocks-Pinch random search step, which starts at $i = 0$ and $j = 0$, into starting from $i = 2^{127}$ and $j = 0$. The use of a larger i lets the Cocks-Pinch search from larger primes first. This method can be found in our example script.

`https://github.com/DZK-Labs/ark-yafa/blob/main/CP-Curve-Script.ipynb`

Avoiding difficult prime factorization of $(q-1)$ when searching for a primitive root. Another issue that arises in practice is that, in many use cases, we need to find a generator (i.e., a primitive root) of the field $\mathbb{F}_q$. There are very few ways to find a generator *with only a negligible possibility of failure* without doing the integer factorization. Moreover, some methods that can probabilistically test a primitive root given a partial factorization would not give a good-looking generator—we generally prefer generators that are small integers, such as 2, 3, 7, 11, or so.

Integer factorization is not impossible. Today, we can first remove the small prime factors from the integer, for example, using Dario Alejandro Alpern's tool [Alp]. For example, $p = 2 \cdot 3^2 \cdot 5 \cdot 3424200193 \cdot B$ where B is a large composite number that does not have small prime factors.

The next step is to factor B. Sometimes, B can be about 650 bits, so it is going to be difficult. The best approach to our knowledge is to use Cado-NFS library [Cad] with a cluster of machines.[4] This is going to take days to complete.

We suggest a different approach. That is, we give up curves whose base field modulus is *not trivial* to factorize. The approach is very simple and consists of two steps.

- **Step 1: finding many curves.** Run the Cocks-Pinch algorithm multiple times. Since the Cocks-Pinch algorithm is randomized, we should be able to get many curves with the prescribed scalar field modulus. We suggest to sample $\ln(2 \cdot r)$ curves. This is a heuristic, not a tight bound. It is based on the prime number theorem about the density of primes.
- **Step 2: factoring with a bounded time.** Run the factorization algorithm and give up after a period of time if the factorization is not complete. Try this until we find a curve with q that can be factorized very quickly. Generally, the bound can be just 15 seconds.

The reason that it will work is because the number B appears above, if we assume that the use of Cocks-Pinch method does not distort the distribution of B too much, is just a number that comes randomly. In many situations, B is a composite number, so we need to factorize it further. But we should have a probability similar to the density of primes to meet B that is exactly a prime number.

One can see that there is a heuristic about *smoothness*, and we do not have a proof. However, it appears to work. We implemented this revised algorithm, and the code can be found here:

`https://github.com/DZK-Labs/ark-yafa/blob/main/CP-Curve-Script.ipynb`

6 Yafa-108

We want to find a pairing-friendly curve with bit security similar to that of BN254, so it can be used for performance comparison. We use the Cocks-Pinch method [FST10], which will generate a curve of more than 512 bits in our case. We can choose the embedding degree among $k = 2, 3, 4, 6, 12$. Our goal is to achieve a high level of security without paying too much cost.

An interesting observation in `arkworks-rs` is that field operations are implemented through *limbs* each of 64 bits. The performance of field operations (excluding powering by another field element) depends on the number of limbs. As a result, a base field of 530 bits will end up having almost the same performance as that of 570 bits, while the latter offers higher security.

We aim to find a secure curve that we can obtain within an embedding degree $k = 6$ and with only nine limbs (because $576/64 = 9$). To do so, we make a small tweak to the implementation of the Cocks-Pinch algorithm, so it can find a base field with a large modulus.

[4] Be sure to override the client thread numbers in the configuration, the default of which is 2, which would render a 128-core machine not useful.

We end up with the following curve, which we call YAFA-108.

$$
\begin{aligned}
q &= 34780837939964648916783391909097838466714157209549513956741661668420080527396705590597622124144427710754148825442779617551149172671304488041519255243374122027992030525080041\\
t &= 37299242855567268678993569817230422880633791847334844156946271893873030437719595280618 3\\
r &= 57896044618658097711785492504343953926634992332820282019728792003956564819949\\
k &= 6\\
D &= -3
\end{aligned}
$$

6.1 Fr and Fq

Scalar field $\mathbb{F}_r$. Prime r is the modulus of the ed25519, i.e., $2^{255} - 19$.

$$
r = 57896044618658097711785492504343953926634992332820282019728792003956564819949
$$

We pick 2 as the generator. The following code implements $\mathbb{F}_r$ using `MontConfig`.

```
use ark_ff::fields::{Fp256, MontBackend, MontConfig};

#[derive(MontConfig)]
#[modulus = "57896044...64819949"]
#[generator = "2"]
pub struct FrConfig;
pub type Fr = Fp256<MontBackend<FrConfig, 4>>;
```

Base field $\mathbb{F}_q$. Prime q is the modulus of the curve's coordinates.

$$
q = 34780837939964648916783391909097838466714157209549513956741661668420080527396705590597622124144427710754148825442779617551149172671304488041519255243374122027992030525080041
$$

We pick 11 as the generator. The following code implements $\mathbb{F}_q$.

```
use ark_ff::fields::{Fp576, MontBackend, MontConfig};

#[derive(MontConfig)]
#[modulus = "34780837...25080041"]
#[generator = "11"]
pub struct FqConfig;
pub type Fq = Fp576<MontBackend<FqConfig, 9>>;
```

We choose the two prime field structs `Fp576` (for $\mathbb{F}_q$) and `Fp256` (for $\mathbb{F}_r$) according to the number of bits of these prime moduli. The "`Mont`" prefix here refers to Montgomery representations, which is the backend for computation on the prime fields (and their extensions).

The config struct is populated by the macro `MontConfig`, so that developers do not need to compute the constants themselves and fill them in like in `arkworks-rs` 0.3.0.

6.2 Fq3 in the tower

We want to extend the field $\mathbb{F}_q$ to $\mathbb{F}_{q^3}$ using an irreducible polynomial $u^3 - N$ with some number $N \in \mathbb{F}_q$. Now, we can represent an element in $\mathbb{F}_{q^3}$ as follows:

$$Au^2 + Bu + C \in \mathbb{F}_{q^3}$$

When we are selecting N, we additionally require that x is a quadratic nonresidue in $\mathbb{F}_{q^3}$, which allows us to use x as the twist when we extend $\mathbb{F}_{q^3}$ to $\mathbb{F}_{q^6}$.

Polynomial and field extension. We use the following polynomial for the extension from $\mathbb{F}_q$ to $\mathbb{F}_{q^3}$ by trying different positive numbers from small to large and checking if it is an irreducible polynomial and if x is a quadratic nonresidue in the resultant extension field.

$$u^3 - 11$$

This is implemented by setting the const `NONRESIDUE` as N here.

```
impl Fp3Config for Fq3Config {
    ...
    const NONRESIDUE: Fq = MontFp!("11");
    ...
}
```

Two-arity and trace. Note that trace t is an odd number in the following equation.

$$q^3 - 1 = 2^s \cdot t$$

We have $s = 4$ and:

$$t = 525932653293435374174200488624990236916665940801818813706342014752099374033447210135549060622620717011071671693272538358293105245237479379326746652814911542461087226787768095425155099716832764949927245962724602069737397253512374869324312203186722944214958991228724079225647841579660102562437692163239478356776501667281470943697448305241626580128732062563470240610556826316286272447330951620340284427063866504890086849298968340090267929152471814199277236027625825270463884370012365472318301185268449247586621564731361 5$$

In the config `Fq3Config`, we need to provide the two-arity constant $s = 4$ as well as $(t-1)/2$, which would be as follows:

$$(t-1)/2 = 262966326646717687087100244312495118458332970400909406853171007376049687016723605067774530311310358505535835846636269179146552622618739689663373326407455771230543613393884047712577549858416382474963622981362301034868698626756187434662156101593361472107479495614362039612823920789830051281218846081619739178388250833640735471848724152620813290064366031281735120305278413158143136223665475810170142213531933252445043424649484170045133964576235907099638618013812912635231942185006182736159150592634224623793310782365680 7$$

This is implemented by setting the constants `TWO_ADICITY` and `TRACE_MINUS_ONE_DIV_TWO`.

```
impl Fp3Config for Fq3Config {
    ...
    const TWO_ADICITY: u32 = 3;

    const TRACE_MINUS_ONE_DIV_TWO: &'static [u64] = &[
        0x9df37fa6adfa3f67,
        ...,
        0xb640003d666c6,
    ];
    ...
}
```

Quadratic nonresidue to power *t*. Another const that needs to be provided is for the Tonelli-Shanks algorithm [Ton91; Sha73], which is a quadratic nonresidue in $\mathbb{F}_{q^3}$ to power t. For convenience, we choose u to be the quadratic nonresidue here, and we have:

$$u^t = 203522109311071054040905766067655209617733217918478524368910696672576620577682951501008509894874847390380668843408367149188977183297426787737086746714636999647521717572067 93$$

This is implemented by providing this field element in $\mathbb{F}_{q^3}$. Note that we can use `MontFp` macro to avoid manual translation to the Montgomery representation.

```
impl Fp3Config for Fq3Config {
    ...
    const QUADRATIC_NONRESIDUE_TO_T: Fq3 = Fq3::new(
        MontFp!("20352210...57206793"),
        Fq::ZERO,
        Fq::ZERO,
    );
```

```
    ...
}
```

Frobenius endomorphism coefficients. We heavily use, especially for pairing, the Frobenius endomorphism, which gives an efficient mapping from an element $e \in \mathbb{F}_{q^3}$ to e^q (and e^{q^2} as well), by multiplying each modulo-q part of e with the corresponding Frobenius endomorphism coefficients (for $\mathbb{F}_{q^3}$, others will be slightly different). More concretely, let us consider:

$$e = c_2 u^2 + c_1 u + c_0$$

We use the Frobenius endomorphism coefficients as follows with $N = 11$.

$$\begin{aligned}
f_1^{(1)} &= 1 \\
f_2^{(1)} &= 1 \\
f_1^{(q)} &= N^{(q-1)/3} \\
&= 26889017910506495156320058129568772566653933834480888250851357865717474104627594400663718411564763129536634204743601058030506421687843844062271438195269332524424351101183828 \\
f_2^{(q)} &= N^{2*(q-1)/3} \\
&= 78918200294581537604633337795290659000602233750686257058903038027026064227691111899339037125796645812175146206991785595206427509834606439792478170481047895035676794238962 12 \\
f_1^{(q^2)} &= N^{(q^2-1)/3} = f_2^{(q)} \\
f_2^{(q^2)} &= N^{2*(q^2-1)/3} = f_1^{(q)}
\end{aligned}$$

We have the Frobenius endomorphism:

$$\begin{aligned}
e^q &= c_2 \cdot f_2^{(q)} \cdot u^2 + c_1 \cdot f_1^{(q)} \cdot u + c_0 \\
e^{q^2} &= c_2 \cdot f_2^{(q^2)} \cdot u^2 + c_1 \cdot f_1^{(q^2)} \cdot u + c_0
\end{aligned}$$

This is implemented as follows:

```
impl Fp3Config for Fq3Config {
    ...
    const FROBENIUS_COEFF_FP3_C1: &'static [Fq] = &[
        Fq::ONE,
        MontFp!("26889017...01183828"),
        MontFp!("78918200...23896212"),
    ];

    const FROBENIUS_COEFF_FP3_C2: &'static [Fq] = &[
        Fq::ONE,
        Self::FROBENIUS_COEFF_FP3_C1[2],
```

```
            Self::FROBENIUS_COEFF_FP3_C1[1],
        ];
        ...
}
```

6.3 Fq6 in the tower

For the degree-six Cocks-Pinch curve, the next step is to construct the field extension $\mathbb{F}_{q^6}$. We extend $\mathbb{F}_{q^3}$ using a degree-two irreducible polynomial (with coefficients in $\mathbb{F}_{q^3}$, not $\mathbb{F}_q$), as follows, with $N' \in \mathbb{F}_{q^3}$.

$$v^2 - N'$$

With this, we can represent an element $e \in \mathbb{F}_{q^6}$ as:

$$e = (d_2 \cdot u^2 + d_1 \cdot u + d_0) \cdot v + (c_2 \cdot u^2 + c_1 \cdot u + c_0)$$

Polynomial and field extension. Note that in the previous step, we have intentionally choose N' such that x is a quadratic nonresidue, and therefore we pick $N' = u$.

$$v^2 - u$$

We can implement this extension by specifying the nonresidue, here u, in the config of $\mathbb{F}_{q^6}$.

```
impl Fp6Config for Fq6Config {
    ...
    const NONRESIDUE: Fq3 = Fq3::new(Fq::ZERO, Fq::ONE, Fq::ZERO);
    ...
}
```

Frobenius endomorphism coefficients. Similar to the case of $\mathbb{F}_{q^3}$, we have the Frobenius endomorphism. The mapping $e \in \mathbb{F}_{q^6}$ to e^q (and, e^{q^2} to e^{q^5}) is as follows. We first write out e.

$$e = (d_2 \cdot u^2 + d_1 \cdot u + d_0) \cdot v + (c_2 \cdot u^2 + c_1 \cdot u + c_0)$$

for e^{q^k} where $k \in \{1, 2, 3, 4, 5\}$. The Frobenius endomorphism will first be applied to each of the $\mathbb{F}_{q^3}$ coefficients, and then it multiplies the degree-one or degree-two terms (i.e., with u or u^2) with the coefficients. Let $k' = k \mod 3$.

$$e^{q^k} = (d_2 \cdot f_2^{(q^{k'})} \cdot u^2 + d_1 \cdot f_1^{(q^{k'})} \cdot u + d_0) \cdot g_1^{(q^k)} \cdot v + (c_2 \cdot f_2^{(q^{k'})} \cdot u^2 + c_1 \cdot f_1^{(q^{k'})} \cdot u + c_0)$$

where the new Frobenius endomorphism coefficients $g_1^{(1)}$, $g_1^{(q)}$, $g_1^{(q^2)}$, $g_1^{(q^3)}$, $g_1^{(q^4)}$, and $g_1^{(q^5)}$ are as

follows.

$$
\begin{aligned}
g_1^{(1)} &= 1 \\
g_1^{(q)} &= N'^{(q-1)/2} = g_1^{(q^2)} + 1 \\
g_1^{(q^2)} &= N'^{(q^2-1)/2} = f_1^{(q)} \\
g_1^{(q^3)} &= N'^{(q^3-1)/2} = -1 \\
g_1^{(q^4)} &= N'^{(q^4-1)/2} = f_1^{(q^2)} \\
g_1^{(q^5)} &= N'^{(q^5-1)/2} = g_1^{(q^4)} + 1
\end{aligned}
$$

6.4 G1

First of all, we use the complex multiplication (CM) method to find a short Weierstrass curve. Note that since $D = -3$, so there is no "ax" term.

$$E(\mathbb{F}_q) : y^2 = x^3 + 5645376$$

Curve parameters. We can implement the curve in `arkworks-rs` as follows.

```
impl SWCurveConfig for Parameters {
    ...
    const COEFF_A: Fq = Fq::ZERO;

    const COEFF_B: Fq = MontFp!("5645376");
    ...
}
```

Cofactor. The number of points on this curve can be computed directly from q and t:

$$\#E(\mathbb{F}_q) = q + 1 - t$$

We can compute the cofactor h as follows:

$$
\begin{aligned}
h &= \frac{\#E(\mathbb{F}_q)}{r} \\
&= 60074635787391708399320583251931717746633472649617463898912353838980429345872418178650083051659 1
\end{aligned}
$$

We then compute the inverse of the cofactor modulo r:

$$
\begin{aligned}
h^{-1} \mod r = {} & 29525709254955688747989059845541707767137581572857490857189440336502437830996
\end{aligned}
$$

This is implemented as follows.

```
impl CurveConfig for Parameters {
    type BaseField = Fq;
    type ScalarField = Fr;

    const COFACTOR: &'static [u64] = &[
        0xed29225ad506d56f,
        ...,
        0x4800000815ea74e0
    ];

    const COFACTOR_INV: Fr =
        MontFp!("29525709...37830996");
}
```

Prime-order subgroup generator. The last step is to find a generator of the prime-order subgroup (with order r). To do so, we first find a point on the curve, and then we clear the cofactor to obtain a point in that prime-order subgroup.

We start with $x = 1$ and, fortunately, we can solve the corresponding y via modular square-root:

$$\left(1,\ \begin{matrix} 225180609013982455818835603878917576139104286769556658479161968 7 \\ 372201424747844952708255342828857479046478251177789121867581508 1 \\ 955082014169388538809844641785743769008334 04 \end{matrix}\right)$$

Clearing its cofactor, we obtain the point (x_0, y_0) as the generator of this subgroup, where:

$$\begin{aligned} x_0 = {} & 342248826894878568978954794187452399856293663506096444824264574795199602203651277865229988423487237803095853640448382072984385947208207686691937758849409428204406414030638 78 \\ y_0 = {} & 531252248722162526044089347491731336050196735203224076668566468302402949866896014787473405742067724694482254210724441982798591548014930288574006468717544900780067911325970 \end{aligned}$$

This is implemented as follows.

```
impl SWCurveConfig for Parameters {
    ...
    const GENERATOR: G1Affine =
        G1Affine::new_unchecked(G1_GENERATOR_X, G1_GENERATOR_Y);
}

pub const G1_GENERATOR_X: Fq = MontFp!("34224882...03063878");
pub const G1_GENERATOR_Y: Fq = MontFp!("53125224...11325970");
```

6.5 G2

We now work on the tricky part. Note that the complex multiplication algorithm does not give us a curve in $\mathbb{F}_{q^3}$. We also cannot randomly pick a curve, as we want this curve to have a prime-order

subgroup $\mathbb{G}_2$ of order r. This is done by using the twist $u \in \mathbb{F}_{q^3}$. We have specifically made sure that it is a quadratic nonresidue in $\mathbb{F}_{q^3}$. Given the curve where $\mathbb{G}_1$ resides:

$$E : y^2 = x^3 + b$$

This curve has a quadratic twist E' in $\mathbb{F}_{q^3}$:

$$E' : y^2 = x^3 + u^3 \cdot b$$

which would have the same number of points. The reason that this new curve has the same number of points is that, if $(x, y) \in E(\mathbb{F}_{q^3})$, then the following (x', y') is on the twist $E'(\mathbb{F}_{q^3})$.

$$x' = u \cdot x$$
$$y' = u\sqrt{u} \cdot y$$

Curve parameters. As discussed above, for the twist $E' : y^2 = x^3 + b'$ and $b' = u^3 \cdot b$:

$$b' = \texttt{62099136}$$

This is implemented as follows:

```
impl SWCurveConfig for Parameters {
    const COEFF_A: Fq3 = Fq3::new(Fq::ZERO, Fq::ZERO,  Fq::ZERO);
    const COEFF_B: Fq3 = Fq3::new(MontFp!("62099136"), Fq::ZERO, Fq::ZERO);
    ...
}
```

Cofactor. We run the point counting algorithm to obtain the number of points:

$$\#E'(\mathbb{F}_{q^3}) = \texttt{420746122634748299339360390899992189533332752641455050965073611801679499226757768108439248498096573608857337354618030686634484196189983503461397322251929233968869781430214476340124079773466211959941796770179681655789917802809899895459449762549378355371967193112709155484805208360841279974471883330334495563453338991543827777138434663636604680411357118442020806117785175500498887413823475858165388843866627905506432739986806671113207912916003889604437124562504540228615695839848258974415002557622479154918613740049589 00$$

We then compute the cofactor h' as follows:

$$
\begin{aligned}
h' &= \frac{\#E'(\mathbb{F}_{q^3})}{r} \\
&= 420746122634748299339360390899992189533332752641455050965073611872672688679522484460586206900989894086168790130307569176290498577986219964362954936816064810151619350167295543707587565299392416364421986785733281438298168766280647612861346012789796421723037816904205191704179354833022069520640678460834095328614366187359199056485287757573541495685890299988388746469464695785011172642357280924265223801538252609958842083428696991687019986883557238600204639686543939240927556859455764705768433213286823523610 0
\end{aligned}
$$

We also compute the inverse of the cofactor modulo r:

$$
h^{-1} \mod r = 225827219586991204597908152714567808108588298592617184894787454805789726301 74
$$

This is implemented as follows.

```
impl CurveConfig for Parameters {
    type BaseField = Fq3;
    type ScalarField = Fr;

    const COFACTOR: &'static [u64] = &[
        0x55b65ba67d349304,
        ...
        0x16c80007accd8dc
    ];
    const COFACTOR_INV: Fr = MontFp!("22582721...72630174");
}
```

Prime-order subgroup generator. We start with $x = u + 1$ and after a few attempts, we find that when $x = u + 3$, we can solve the corresponding y.

$$
\left(
\begin{array}{l}
u + 3, \\
\\
\\
\\
\end{array}
\begin{array}{l}
+ \left(\begin{array}{l} 16399745525914760875122822753296350670146148646493981347801019658497465183226048954715496573023028790116281383881950421850106365423872926445380862056496154724756285267266888 \end{array} \right) * u^2 \\
+ \left(\begin{array}{l} 14287254449057613428239123269232765925619703824469704202901877151310093048222249908017251200009970845728118751765949951105910650164572175833612033146936098637881156817742266 \end{array} \right) * u \\
+ \left(\begin{array}{l} 67581327726597132966749957858127838644323731843665910730728403630277160345581250943954549492116413341182297640653467835630039837549443336827320517413471064766205641484865 0 \end{array} \right)
\end{array}
\right)
$$

Clearing its cofactor, we obtain the following point as the generator of the subgroup:

$$x_0 = c_2 \cdot u^2 + c_1 \cdot u + c_0$$
$$y_0 = d_2 \cdot u^2 + d_1 \cdot u + d_0$$

with the parameters as follows.

$$\begin{aligned}
c_2 &= 0 \\
c_1 &= 0 \\
c_0 &= 2539269890488470913299903776025487665772518766768269681318682797086578747189164076404921241729192168201150546594885700406229682279647082230073412377752088833465897799856859 \\
d_2 &= 0 \\
d_1 &= 0 \\
d_0 &= 4396708447983789823202089491402954890042030420320791355295989970672982059020950821852365967834639548445057566914035829163770719788716368901156926304377953920559312066532663
\end{aligned}$$

This is implemented as follows:

```
impl SWCurveConfig for Parameters {
    ...
    const GENERATOR: G2Affine =
        G2Affine::new_unchecked(G2_GENERATOR_X, G2_GENERATOR_Y);
}

const G2_GENERATOR_X: Fq3 =
    Fq3::new(G2_GENERATOR_X_C0, G2_GENERATOR_X_C1, G2_GENERATOR_X_C2);
const G2_GENERATOR_Y: Fq3 =
    Fq3::new(G2_GENERATOR_Y_C0, G2_GENERATOR_Y_C1, G2_GENERATOR_Y_C2);

pub const G2_GENERATOR_X_C0: Fq = MontFp!("25392698...98568595");
pub const G2_GENERATOR_X_C1: Fq = Fq::ZERO;
pub const G2_GENERATOR_X_C2: Fq = Fq::ZERO;
pub const G2_GENERATOR_Y_C0: Fq = MontFp!("43967084...66532663");
pub const G2_GENERATOR_Y_C1: Fq = Fq::ZERO;
pub const G2_GENERATOR_Y_C2: Fq = Fq::ZERO;
```

6.6 Ate pairing

Recall the ate pairing has the following formula:

$$\mathsf{ate}(P, Q) = f_{t-1,Q}(P)^{(q^6-1)/r}$$

where t is the trace. Note that we can rewrite it as follows with A, B derived from q.

$$q^6 - 1 = (q^3 - 1) \cdot (q + 1) \cdot r \cdot (Aq + B)$$

The reason we rewrite it in this way is to use the Frobenius endomorphism, which is an efficient way to map e to e^{q^k} for different k. We can find A and B with some calculation.

Ate pairing loop count. For the ate pairing, we simply use $t - 1$ as the ate pairing loop count.

$$\begin{aligned} t - 1 = \ & 37299242855567268678993569817230422880633791847 3 \\ & 34844156946271893873030437719595280618 2 \end{aligned}$$

The constants related to the Miller loop portion of the ate pairing are implemented as follows:

```
pub const TWIST: Fq3 = Fq3::new(Fq::ZERO, Fq::ONE, Fq::ZERO);
pub const ATE_IS_LOOP_COUNT_NEG: bool = false;
pub const ATE_LOOP_COUNT: [u64; 5] = [
    0x55792745f9e71926,
    0xe5e55257bec5baaf,
    0x2678d0333aa42ad6,
    0xc7e345c9559a9a4a,
    0xc000000a,
];
```

Final exponentiation last chuck coefficients. We compute A and B as follows.

$$\begin{aligned} A = \ & 19533729308556160109794825562832707092 1 \\ & 34991522349430861538344636908041560785 8 \\ & 66966036323918363729219562215110665533 9 \\ & 50746637708295543215802259030565596621 8 \\ & 23690292271103113 \\ B = \ & 60074635787391708399320583251931717746 6 \\ & 33472649617463898912353838980429345872 4 \\ & 181786507272967556 \end{aligned}$$

This is implemented as follows.

```
pub const FINAL_EXPONENT_LAST_CHUNK_W0_IS_NEG: bool = false;
pub const FINAL_EXPONENT_LAST_CHUNK_ABS_OF_W0: BigInt<9> = BigInt::new([
    0x7d919d468ad5bc89,
    ...
    0x1437eba2e55524f8,
]);
pub const FINAL_EXPONENT_LAST_CHUNK_W1: BigInt<9> = BigInt::new([
    0xed29225c5506d584,
    ...
    0x0,
]);
```

7 Yafa-146

We now want to find a pairing-friendly curve that achieves 128-bit security. To avoid having a very large q, we choose to increase the embedding degree to $k = 12$. Then, leveraging the observation that the performance depends on the number of limbs rather than exactly the number of bits, we choose to find q that is not far away from, but lower than 2^{575}, which allows us to obtain a desirable bit security without performance overhead.

$$
\begin{aligned}
q = {} & 347808467935976491159023607593301147659538310377 \\
& 620365965388902199388348386346598101672702796413 \\
& 236015098861083339046516024649685450016420130751 \\
& 020435104587515335438974282 01 \\
t = {} & 372992476029187189170462968044878524314803726725 \\
& 266721356058824603476064987295668589899 \\
r = {} & 578960446186580977117854925043439539266 \\
& 349923328202820197287920039565648199 49 \\
k = {} & 12 \\
D = {} & -3
\end{aligned}
$$

7.1 Fr and Fq

Scalar field $\mathbb{F}_r$. Prime r is the modulus of the ed25519, i.e., $2^{255} - 19$.

$$
\begin{aligned}
r = {} & 578960446186580977117854925043439539266 \\
& 349923328202820197287920039565648199 49
\end{aligned}
$$

We pick 2 as the generator. The following code implements $\mathbb{F}_r$ where "57896044...64819949", which is given as a parameter to `MontConfig`, is r above.

```
use ark_ff::fields::{Fp256, MontBackend, MontConfig};

#[derive(MontConfig)]
#[modulus = "57896044...64819949"]
#[generator = "2"]
pub struct FrConfig;
pub type Fr = Fp256<MontBackend<FrConfig, 4>>;
```

Base field $\mathbb{F}_q$. Prime q is the modulus of the Yafa curve's coordinates.

$$
\begin{aligned}
q = {} & 347808467935976491159023607593301147659538310377 \\
& 620365965388902199388348386346598101672702796413 \\
& 236015098861083339046516024649685450016420130751 \\
& 020435104587515335438974282 01
\end{aligned}
$$

We pick 59 as the generator. The following code implements $\mathbb{F}_q$.

```
use ark_ff::fields::{Fp576, MontBackend, MontConfig};

#[derive(MontConfig)]
#[modulus = "34780846...97428201"]
#[generator = "59"]
pub struct FqConfig;
pub type Fq = Fp576<MontBackend<FqConfig, 9>>;
```

7.2 Fq2 in the tower

We want to extend the field $\mathbb{F}_q$ to $\mathbb{F}_{q^2}$ using an irreducible polynomial in $\mathbb{F}_q$ that looks like $u^2 - N$ with some number $N \in \mathbb{F}_q$. Now, we can represent an element in $\mathbb{F}_{q^2}$ as follows:

$$Au + B \in \mathbb{F}_{q^2}$$

Polynomial and field extension. We use the following polynomial for the extension from $\mathbb{F}_q$ to $\mathbb{F}_{q^2}$ by trying different positive numbers from small to large and checking if it is an irreducible polynomial and if x is a quadratic nonresidue in the resultant field extension.

$$u^2 - 17$$

This is implemented by setting the const `NONRESIDUE` as N here.

```
impl Fp2Config for Fq2Config {
    ...
    const NONRESIDUE: Fq = MontFp!("17");
    ...
}
```

Frobenius endomorphism coefficients. We use the Frobenius endomorphism coefficients as follows with $N = 17$.

$$f^{(1)} = 1$$
$$f^{(q)} = -1$$

We have the Frobenius endomorphism:

$$e^q = c_1 \cdot f_1^{(q)} \cdot u + c_0$$
$$e^{q^2} = c_1 \cdot f_1^{(q^2)} \cdot u + c_0$$

This is implemented as follows.

```
impl Fp2Config for Fq2Config {
    ...
    const FROBENIUS_COEFF_FP2_C1: &'static [Fq] = &[
        Fq::ONE, MontFp!("-1"),
    ];
    ...
}
```

7.3 Fq6 in the tower

The next step is to construct the field extension $\mathbb{F}_{q^6}$. We choose to extend $\mathbb{F}_{q^2}$ using a degree-three irreducible polynomial (with coefficients in $\mathbb{F}_{q^2}$, not $\mathbb{F}_q$), as follows, with $N' \in \mathbb{F}_{q^2}$.

$$v^3 - N'$$

With this, we can represent an element $e \in \mathbb{F}_{q^6}$ as:

$$e = (h_1 \cdot u + h_0) \cdot v^2 + (d_1 \cdot u + d_0) \cdot v + (c_1 \cdot u + c_0)$$

Polynomial and field extension. We pick $N' = 7 \cdot u$.

$$v^3 - 7 \cdot u$$

We can implement this extension by specifying the nonresidue in the config of $\mathbb{F}_{q^6}$.

```
impl Fp6Config for Fq6Config {
    ...
    const NONRESIDUE: Fq2 = Fq2::new(Fq::ZERO, MontFp!("7"));
    ...
}
```

Frobenius endomorphism coefficients. Similar to the case of $\mathbb{F}_{q^3}$, we have the Frobenius endomorphism. The mapping $e \in \mathbb{F}_{q^6}$ to e^q (and, e^{q^2} to e^{q^5}) is as follows. We first write out e.

$$e = (h_1 \cdot u + h_0) \cdot v^2 + (d_1 \cdot u + d_0) \cdot v + (c_1 \cdot u + c_0)$$

for e^{q^k} where $k \in \{1, 2, 3, 4, 5\}$. The Frobenius endomorphism will first be applied to each of the $\mathbb{F}_{q^2}$ coefficients, and then it multiplies the degree-one term (i.e., with u) with the coefficients. Let $k' = k \mod 2$.

$$e^{q^k} = (h_1 \cdot f_1^{(q^{k'})} \cdot u + h_0) \cdot g_2^{(q^k)} \cdot v^2 + (d_1 \cdot f_1^{(q^{k'})} \cdot u + d_0) \cdot g_1^{(q^k)} \cdot v + (c_1 \cdot f_1^{(q^{k'})} \cdot u + c_0)$$

where the new Frobenius endomorphism coefficients $g_1^{(1)}$ to $g_1^{(q^5)}$ and $g_2^{(1)}$ to $g_2^{(q^5)}$ are as follows.

$$\begin{aligned}
g_1^{(1)} &= 1 \\
g_1^{(q)} &= N'^{(q-1)/3} = g_1^{(q^2)} + 1 \\
g_1^{(q^2)} &= N'^{(q^2-1)/3} = f_1^{(q)} \\
g_1^{(q^3)} &= N'^{(q^3-1)/3} = -1 \\
g_1^{(q^4)} &= N'^{(q^4-1)/3} = f_1^{(q^2)} \\
g_1^{(q^5)} &= N'^{(q^5-1)/3} = g_1^{(q^4)} + 1 \\
g_2^{(1)} &= 1
\end{aligned}$$

$$
\begin{aligned}
g_2^{(q)} &= N'^{2(q-1)/3} = g_1^{(q^2)} \\
g_2^{(q^2)} &= N'^{2(q^2-1)/3} = g_1^{(q^4)} \\
g_2^{(q^3)} &= N'^{2(q^3-1)/3} = 1 \\
g_2^{(q^4)} &= N'^{2(q^4-1)/3} = g_1^{(q^2)} \\
g_2^{(q^5)} &= N'^{2(q^5-1)/3} = g_1^{(q^4)}
\end{aligned}
$$

This is implemented as follows.

```
impl Fp6Config for Fq6Config {
    ...
    const FROBENIUS_COEFF_FP6_C1: &'static [Fp2<Self::Fp2Config>] = &[
        Fq2::new(
            Fq::ONE,
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413060"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413059"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("-1"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015141"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015142"),
            Fq::ZERO,
        ),

    ];
    const FROBENIUS_COEFF_FP6_C2: &'static [Fp2<Self::Fp2Config>] = &[
        Fq2::new(
            Fq::ONE,
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413059"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015141"),
            Fq::ZERO,
```

```
        ),
        Fq2::new(
            Fq::ONE,Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413059"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015141"),
            Fq::ZERO,
        ),
    ];
}
```

7.4 Fq12 in the tower

We choose $D = -3$ because it provides us a sextic twist [Sco] from $\mathbb{F}_{q^2}$ immediately to $\mathbb{F}_{q^{12}}$. This is important for pairing. There are two ways to instantiate $\mathbb{F}_{q^{12}}$. One is to extend $\mathbb{F}_{q^6}$ with $w^2 - v$, and the other one is to extend $\mathbb{F}_{q^2}$ with $w^6 - 7 \cdot u$. We implement this part as follows.

```
impl Fp12Config for Fq12Config {
    ...
    const NONRESIDUE: Fq6 = Fq6::new(Fq2::ZERO, Fq2::ONE, Fq2::ZERO);
    ...
}
```

Frobenius endomorphism coefficients. The mapping $e \in \mathbb{F}_{q^{12}}$ to e^q (and, e^{q^2} to $e^{q^{11}}$) can be described as follows. We first write out e.

$$\begin{aligned} e = &((h_1' \cdot u + h_0') \cdot v^2 + (d_1' \cdot u + d_0') \cdot v + (c_1' \cdot u + c_0')) \cdot w \\ &+ (h_1 \cdot u + h_0) \cdot v^2 + (d_1 \cdot u + d_0) \cdot v + (c_1 \cdot u + c_0) \end{aligned}$$

for e^{q^k} where $k \in \{1, 2, ..., 11\}$. The Frobenius endomorphism will work as follows. Let $k' = k \mod 6$ and $\widetilde{k} = k \mod 2$.

$$\begin{aligned} e = &((h_1' \cdot f_1^{(q^{\widetilde{k}})} \cdot u + h_0') \cdot g_2^{(q^{k'})} \cdot v^2 + (d_1' \cdot f_1^{(q^{\widetilde{k}})} \cdot u + d_0') \cdot g_1^{(q^{k'})} \cdot v + (c_1' \cdot f_1^{(q^{\widetilde{k}})} \cdot u + c_0')) \\ &\cdot m_1^{(q^k)} \cdot w \\ &+ (h_1 \cdot f_1^{(q^{\widetilde{k}})} \cdot u + h_0) \cdot g_2^{(q^{k'})} \cdot v^2 + (d_1 \cdot f_1^{(q^{\widetilde{k}})} \cdot u + d_0) \cdot g_1^{(q^{k'})} \cdot v + (c_1 \cdot f_1^{(q^{\widetilde{k}})} \cdot u + c_0) \end{aligned}$$

where the new Frobenius endomorphism coefficients $k_1^{(1)}$ to $k_1^{(q^11)}$ are as follows.

$$\begin{aligned} m_1^{(1)} &= 1 \\ m_1^{(q)} &= N'^{(q-1)/6} \\ &= 9757343029987591069013430723332555716764176417776465 \end{aligned}$$

$$
\begin{aligned}
& 69787288414444945311593497431512102321318168567 2 \\
& 69856400200172867906148712026502435622397151728 1 \\
& 62451226139810885168012943 30 \\
m_1^{(q^2)} &= N'^{(q^2-1)/6} \\
&= 22684684505229882416948285270218997440839683235 3 \\
& 27132785256469607245693680604029123404282437563 4 \\
& 74551944295516999841064542480240558845597162904 7 \\
& 26440521080541363789749413060 \\
m_1^{(q^3)} &= N'^{(q^3-1)/6} \\
&= 10907262302462287557760590083099326333305744449 3 \\
& 31424419812495591525729643523851429020508443027 6 \\
& 05463784203812151719095264580234048899993127396 1 \\
& 34080477967224420449328059879 \\
m_1^{(q^4)} &= N'^{(q^4-1)/6} \\
&= 22684684505229882416948285270218997440839683235 3 \\
& 27132785256469607245693680604029123404282437563 4 \\
& 74551944295516999841064542480240558845597162904 7 \\
& 26440521080541363789749413059 \\
m_1^{(q^5)} &= N'^{(q^5-1)/6} \\
&= 11499192724746964887471593597667706165415666846 7 \\
& 44456909710510969945702937807002187883766261708 7 \\
& 84781441837948649284803933775838053377534122233 1 \\
& 78353553532433319325267655 49 \\
m_1^{(q^6)} &= N'^{(q^6-1)/6} = -1 \\
m_1^{(q^7)} &= N'^{(q^7-1)/6} \\
&= 25023503763610058046888930035997559049189653273 1 \\
& 05057867697445725407675488891508599935138462784 5 \\
& 96615869866091047114036731262318301439402297902 2 \\
& 85798387844770445027096133871 \\
m_1^{(q^8)} &= N'^{(q^8-1)/6} \\
&= 12096162288367766698954075489111117325114147802 4 \\
& 34903811282420612693141158030630686762987842077 8 \\
& 49049565590591334063587059984727986156044850170 3 \\
& 75602989378210169754148015141
\end{aligned}
$$

$$
\begin{aligned}
m_1^{(q^9)} &= N'^{(q^9-1)/6} \\
&= 238735844911353615581417706762307884326480865884 \\
&\quad 306121767263946284131051951108083811467618366137 \\
&\quad 181377256822961821855563378847344961016488856789 \\
&\quad 67963032491527113094569368322 \\
m_1^{(q^{10})} &= N'^{(q^{10}-1)/6} \\
&= 120961622883677666989540754891111173251141478024 \\
&\quad 349038112824206126931411580306306867629878420778 \\
&\quad 490495655905913340635870599847279861560448501703 \\
&\quad 756029893782101697541480151 42 \\
m_1^{(q^{11})} &= N'^{(q^{11}-1)/6} \\
&= 336309275211229526271552013995633441494122643530 \\
&\quad 875909055678391229442645448539595913788936534704 \\
&\quad 451233657023134689761712090873847396638886008517 \\
&\quad 842081551055082016113706626 52
\end{aligned}
$$

This is implemented as follows.

```
impl Fp12Config for Fq12Config {
    ...
    const FROBENIUS_COEFF_FP12_C1: &'static [Fq2] = &[
        Fq2::new(Fq::ONE, Fq::ZERO),
        Fq2::new(
            MontFp!("97573430...01294330"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413060"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("10907262...28059879"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("22684684...49413059"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("11499192...26765549"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("-1"),
            Fq::ZERO,
```

```
        ),
        Fq2::new(
            MontFp!("25023503...96133871"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015141"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("23873584...69368322"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("12096162...48015142"),
            Fq::ZERO,
        ),
        Fq2::new(
            MontFp!("33630927...70662652"),
            Fq::ZERO,
        ),
    ];
}
```

7.5 G1

The complex multiplication (CM) method gives us the following curve where $a = 0$.

$$E(\mathbb{F}_q) : y^2 = x^3 + b$$

where $b = 33355508094144$.

Curve parameters. We can implement the curve in `arkworks-rs` as follows.

```
impl SWCurveConfig for Parameters {
    ...
    const COEFF_A: Fq = Fq::ZERO;
    const COEFF_B: Fq = MontFp!("33355508094144");
    ...
}
```

Cofactor. The number of points on this curve can be computed directly from q and t:

$$\#E(\mathbb{F}_q) = q + 1 - t$$

We compute the cofactor h as follows:

$$\begin{aligned} h &= \frac{\#E(\mathbb{F}_q)}{r} \\ &= 60074651079685091332544189152021479255239942764476097817603033432226815818441770856742416948434 7 \end{aligned}$$

We also compute the inverse of the cofactor modulo r:

$$h^{-1} \mod r = 5473669270250647926635954563013553500873468338965408984060107115428611828955 0$$

This is implemented as follows.

```
impl CurveConfig for Parameters {
    type BaseField = Fq;
    type ScalarField = Fr;

    const COFACTOR: &'static [u64] = &[
        0xba0cbd29dbed203b,
        ...,
        0x4800013b93e07009
    ];

    const COFACTOR_INV: Fr =
        MontFp!("54736692...18289550");
}
```

Prime-order subgroup generator. The last step of the implementation is to find a generator of the prime-order subgroup (with order r). To do so, we first find a point on the curve, and we clear the cofactor to obtain a point in that prime-order subgroup.
We start with $x = 4$ and, fortunately, we can solve the corresponding y via modular square-root:

$$\left(4,\ \begin{matrix} 9980063903480773747485374411020714173816276815606417861225 4 \\ 3965455439074571900572727405595500929462457895602986516057 5 \\ 9677175794683384875828784333423593103526902047173508 54 \end{matrix}\right)$$

Clearing its cofactor, we obtain the point (x_0, y_0) as the generator of this subgroup, where:

$$x_0 = 2104147906033405999491791635256199559968149668337818433849504414855433486592228504327546552874196430698038347500145540461060067528597360729591129549104643355056266594885010 9$$

$$y_0 = 1841897213774345215194080213544536275781413636928742996668 2408792399987603108067514115419789663370270540458880262504824821590977011959474826305566676273107411347744242094 42$$

This is implemented as follows.

```
impl SWCurveConfig for Parameters {
    ...
    const GENERATOR: G1Affine =
        G1Affine::new_unchecked(G1_GENERATOR_X, G1_GENERATOR_Y);
}

pub const G1_GENERATOR_X: Fq = MontFp!("21041479...48850109");
pub const G1_GENERATOR_Y: Fq = MontFp!("18418972...24209442");
```

7.6 G2

Different from Yafa-108, here we consider a divisive twist, using the quadratic nonresidue $7 \cdot u$.

$$E : y^2 = x^3 + b$$

This curve has a quadratic twist E' in $\mathbb{F}_{q^2}$:

$$E' : y^2 = x^3 + b/(7 \cdot u)$$

which would have the same number of points.

Curve parameters. The parameters for the twist $E' : y^2 = x^3 + b'$ where $b' = b/(7 \cdot u)$ are:

$$\begin{aligned} b' = \ & 13444697079878082851525282310329288060788875863336585575132 \\ & 68025308560002165709538880415489801261248461726689901982868 \\ & 88547343575888241641395080226386679084249628266789058 10 \cdot u \end{aligned}$$

This is implemented as follows:

```
impl SWCurveConfig for Parameters {
    const COEFF_A: Fq2 = Fq2::new(Fq::ZERO, Fq::ZERO);
    const COEFF_B: Fq2 = Fq2::new(Fq::ZERO, MontFp!("13444697...78905810"));
    ...
}
```

Cofactor. We run the point counting algorithm to obtain the number of points:

$$\begin{aligned} \#E'(\mathbb{F}_{q^2}) = \ & 12097073036797118695217010035002906074157574967770394847 1 \\ & 55480035440017069659712151314241697637611826882570078699 3 \\ & 36838344331626954875886166837539349405006589346957661356 3 \\ & 60935844425384587681888858933225339160876943690994618127 5 \\ & 41578518425952456020557460102175623471011376326893828987 1 \\ & 54182469982855678805687102544586690184564244379889694769 5 \\ & 2804 \end{aligned}$$

We compute the cofactor h' as follows:

$$\begin{aligned} h' &= \frac{\#E'(\mathbb{F}_{q^2})}{r} \\ &= 20894472353813627584293482548772171994435736321892493652 1 \\ & 56536764483249695448423605439551802991931976343692104147 3 \\ & 78449067499016714870159490307584066363261732930457034869 1 \\ & 06625232959381521895831602203960656224597442267949372368 2 \\ & 42654610238587983559434866653274386897396 \end{aligned}$$

We compute the inverse of the cofactor modulo r:

$$h^{-1} \mod r = \begin{array}{l} 52342251624758656962424635542600246432390708264947618736639330869844643473789 \end{array}$$

This is implemented as follows.

```
impl CurveConfig for Parameters {
    type BaseField = Fq2;
    type ScalarField = Fr;

    const COFACTOR: &'static [u64] = &[
        0x97eb95cb492a99f4
        ...
        0xa200058c197e205
    ];
    const COFACTOR_INV: Fr = MontFp!("52342251...43473789");
}
```

Prime-order subgroup generator. We start with $x = u + 1$ and we find that when $x = u + 2$, we can solve the corresponding y.

$$\left(u + 2, \begin{array}{l} \left(\begin{array}{l} 20496978775801279902615402437137383229601465884 \\ 73270665304981553838136031430176272679003883510 \\ 45700703177566295797875957462797032226111459130 \\ 6190709376617511248954288698 7 \end{array} \right) * u \\ + \left(\begin{array}{l} 29719394895172508108585191209165461584444602080 \\ 27964196739819847427441708043532634528199894702 \\ 29728475440851214105793689714831797463402970539 \\ 25407932021570845701203976101 \end{array} \right) \end{array} \right)$$

Clearing its cofactor, we obtain the following point as the generator of the subgroup:

$$x_0 = c_1 \cdot u + c_0$$
$$y_0 = d_1 \cdot u + d_0$$

with the parameters as follows.

$$
\begin{aligned}
c_1 = {} & 650966855759905342444096668027249778491139980324 \\
& 049753763524734907310739708621631645672763056181 \\
& 680906869079696151064953623466928504377995969683 \\
& 44525663615422358892360458 35 \\
c_0 = {} & 276877800888102642891595917708168330711695521882 \\
& 153924606117580050716422113944769779227015367634 \\
& 170510326250994267548021751651803755130267995138 \\
& 29826051305654731318363059026 \\
d_1 = {} & 571527274820342672130635162888797885508208950312 \\
& 775860491897599778790338317661362807521821317769 \\
& 992408593831310540033309510466456343599759688185 \\
& 754915040952696413757435 0316 \\
d_0 = {} & 119253991112167965184771381516912355685685107114 \\
& 940606503400176200676449391489397367837567649698 \\
& 058122232666762736423692806603088960585248269986 \\
& 851627586463559129915433 36123
\end{aligned}
$$

This is implemented as follows:

```
impl SWCurveConfig for Parameters {
    ...
    const GENERATOR: G2Affine =
        G2Affine::new_unchecked(G2_GENERATOR_X, G2_GENERATOR_Y);
}

const G2_GENERATOR_X: Fq2 =
    Fq2::new(G2_GENERATOR_X_C0, G2_GENERATOR_X_C1);
const G2_GENERATOR_Y: Fq2 =
    Fq2::new(G2_GENERATOR_Y_C0, G2_GENERATOR_Y_C1);

pub const G2_GENERATOR_X_C0: Fq = MontFp!("27687780...63059026");
pub const G2_GENERATOR_X_C1: Fq = MontFp!("65096685...36045835");
pub const G2_GENERATOR_Y_C0: Fq = MontFp!("11925399...43336123");
pub const G2_GENERATOR_Y_C1: Fq = MontFp!("57152727...74350316");
```

7.7 Tate pairing

Since $r = 2^{255} - 19$, here we actually want to use Tate pairing, which has the following formula:

$$\mathsf{Tate}(P, Q) = f_{r,P}(Q)^{(q^{12}-1)/r}$$

Note that we can rewrite it as follows with A, B, C, D derived from q.

$$q^{12} - 1 = (q^6 - 1) \cdot (q^2 + 1) \cdot r \cdot (Aq^3 + Bq^2 + Cq + D)$$

The reason we rewrite it in this way is to use the Frobenius endomorphism, which is an efficient way to map e to e^{q^k} for different k. We can find A and B with some calculations.

Tate pairing loop count. For the Tate pairing, we use r as the loop count, as follows:

```
pub const TWIST: Fq2 = Fq2::new(Fq::ZERO, Fq::ONE);
pub const TATE_LOOP_COUNT: [u64; 4] = [
    0xffffffffffffed,
    0xffffffffffffff,
    0xffffffffffffff,
    0x7fffffffffffff
];
```

Final exponentiation last chuck coefficients. We compute A to D as follows.

$$\begin{aligned}
A = {} & 60074651079685091332544189152021479255239942764476097817603033432226815818441770856743061193613 2\\
B = {} & 18787556099692390437676225874885260726423443207869755270416397806622396448189003069574413752571064036218150696957580977806429906239508024997419791567936987076281415850010664\\
C = {} & 19533734280962826151300710596734343137033625352945232921346926017502773512159370287059500101658952606652219497406040410875183317615347849812200980879667269084124303920125 72\\
D = {} & 27816679719392039590916985961594891329592754889347470862124370788722235197910030055580499179007884620636974433034081916025003956486380300643703133308392747535275498881809581
\end{aligned}$$

This is implemented as follows.

```
pub const FINAL_EXPONENT_LAST_CHUNK_W0: BigInt<9> = BigInt::new([
    0xbc41361f419dfcad,
    ...,
    0x1ccaaec847ce3756,
]);
pub const FINAL_EXPONENT_LAST_CHUNK_W1: BigInt<9> = BigInt::new([
    0x4a8171212b68af1c,
    ...,
    0x1437ebf93e40e187,
]);
```

```
pub const FINAL_EXPONENT_LAST_CHUNK_W2: BigInt<9> = BigInt::new([
    0x3e418200ad402828,
    ...,
    0x1372344f3c10e50b,
]);
pub const FINAL_EXPONENT_LAST_CHUNK_W3: BigInt<9> = BigInt::new([
    0xba0cbd2b5bed2384,
    ...,
    0x0,
]);
```

8 Evaluation

In this section we provide additional detail about the experiments. The results are shown previously in Table A, Table B, Table C, and Table D.

Setup. For convenience we run the experiments on a MacBook Pro with Intel Core i7 Quad-Core at 2.3 GHz and a memory of 32 GB. We use the parallel feature of the `arkworks-rs` library.

Baselines. We recognize that BN254 and BLS12-381 both have sufficient FFT space and can be most efficiently implemented in Groth16 [Gro16], while only a few proof systems work with YAFA-108 and YAFA-146, and we choose Gemini [BCHO22] for them.

We previously mentioned BLS12-446, which has 131-bit security, and should be a more fair choice to compare with YAFA-146, which offers 146-bit security, over BLS12-381. We choose not to compare with BLS12-446 for the following two reasons.

- BLS12-381 is a popular curve, with many (audited) implementations, and its universal setup parameters are widely available. It offers 122-bit security, which is not far from 128-bit security.
- Although definitions of $\mathbb{G}_1$ for the BLS12-446 curve is clear from [Gui20], the quadratic non-residue for the extension from $\mathbb{F}_q$ to $\mathbb{F}_{q^2}$ has not been standardized or been included in the official implementation. We believe that we are not the right person to pick this parameter, as it might contribute to conflicting implementations in the future.

Limitations. Our experiments are for a qualitative analysis of the benefit of application-specific curves, and it is imprecise. In addition, our experiment is not a comprehensive study of this overhead, for the following reasons.

First, this experiment result only reflects the situation on a consumer-grade laptop. The streaming nature of Gemini is hardware-friendly and suitable for ASICs, in which case the experiment results will be more favorable to Gemini. Several companies in the industry have observed that the bottlenecks in software implementations do not necessarily translate to hardware implementations. Rahul Maganti from Jump Crypto [Jum] has summarized a list of different considerations for proof acceleration between a software implementation and an FPGA prover.

Second, this experiment result does not reflect the situation with application-specific proof systems, such as TurboPlonk and Halo2. Customized gates can reduce the overhead of nonnative field arithmetics, such as binary checking, and have been used in production. We want to point out that

such optimization also benefits the native one—customized gates can help reduce the cost of native scalar multiplication over twisted Edwards curves (which is the case of ed25519). More work is needed to assess how customized gates change the results.

Finally, the problem of verifying a large number of scalar multiplications is an example of verifying repeated computation. Protocols that work with structured circuits [XZZPS19; ZXZS20] or with succinct reduction [BSCGGRS19] may have an advantage here.

Acknowledgments

The authors want to thank Simon Masson and Pratyush Mishra for their advice and guideline. We thank Timofey Yaluhin for pointing out a few mistakes in Section 4 regarding the cost estimation. We thank Mikerah Quintyne-Collins from HashCloak for pointing out another mistake in Section 4. We thank Kobi Gurkan, Zhenfei Zhang, and Zihan Zheng for information regarding the situation with EIP-1962 and EIP-2537. We also thank our client Greater Heat from which we learned about the state-of-the-art hardware acceleration for zero-knowledge proofs in professional SNARK-mining services. We thank Aurore Guillevic whose Sage scripts and research papers have been crucial in making this work possible. We thank Alessandro Chiesa, Peter Manohar, Madars Virza, and Howard Wu for their `ecfactory` toolkit.

References

[AFKLO07] Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen K. Lenstra, and Dag Arne Osvik. "A kilobit special number field sieve factorization". In: *ASIACRYPT '07*. 2007.

[Alp] *Dario Alejandro Alpern's integer factorization calculator*. URL: `https://www.alpertron.com.ar/ECM.HTM`.

[Ark] *ark-ec::models*. URL: `https://github.com/arkworks-rs/algebra/tree/master/ec/src/models`.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. "Gemini: Elastic SNARKs for diverse environments". In: *EUROCRYPT '22*. 2022.

[BD19] Razvan Barbulescu and Sylvain Duquesne. "Updating key size estimations for pairings". In: *JoC '19*. 2019.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Scalable, transparent, and post-quantum secure computational integrity". In: *IACR ePrint' 18*. 2018.

[BSCGGRS19] Eli Ben-Sasson, Alessandro Chiesa, Lior Goldberg, Tom Gur, Michael Riabzev, and Nicholas Spooner. "Linear-size constant-query IOPs for delegating computation". In: *TCC '19*. 2019.

[CHMMVW20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. "Marlin: Preprocessing zkSNARKs with universal and updatable SRS". In: *EUROCRYPT '20*. 2020.

[CP01] Clifford Christopher Cocks and Richard G. E. Pinch. "Identity-based cryptosystems based on the Weil pairing". In: *Unpublished manuscript*. 2001.

[Cad] *Cado-NFS: An implementation of the number field sieve algorithm*. URL: `https://gitlab.inria.fr/cado-nfs/cado-nfs`.

[Dra] *Drand: Distributed randomness beacon*. URL: `https://drand.love/`.

[Ecf] *ecfactory: A SageMath library for constructing elliptic curves*. URL: `https://github.com/scipr-lab/ecfactory/`.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. "A taxonomy of pairing-friendly elliptic curves". In: *JoC '10*. 2010.

[Fie] *Field selection for recursive SNARKs*. URL: `https://medium.com/delendum/field-selection-for-recursive-snarks-726ad56c3a3c`.

[GMT20] Aurore Guillevic, Simon Masson, and Emmanuel Thomé. "Cocks–Pinch curves of embedding degrees five to eight and optimal ate pairing computation". In: *Designs, Codes and Cryptography '20*. 2020.

[GS21] Aurore Guillevic and Shashank Singh. "On the alpha value of polynomials in the tower number field sieve algorithm". In: *Mathematical Cryptology '21*. 2021.

[Gro16] Jens Groth. "On the size of pairing-based non-interactive arguments". In: *EUROCRYPT '16*. 2016.

[Gui20] Aurore Guillevic. "A short-list of pairing-friendly curves resistant to special TNFS at the 128-bit security level". In: *PKC '20*. 2020.

[HG20] Youssef El Housni and Aurore Guillevic. "Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition". In: *CANS '20*. 2020.

[Has] *Hashcloak Inc*. URL: `https://hashcloak.com/`.

[Jum] *Rahul Maganti*. URL: `https://mobile.twitter.com/rahulmaganti_/status/1560365591717896192`.

[Sco] Michael Scott. *A note on twists for pairing friendly curves*. URL: `http://indigo.ie/~mscott/twists.pdf`.

[Sha73] Daniel Shanks. "Five number theoretic algorithms". In: *Proceedings of the Second Manitoba Conference on Numerical Mathematics*. 1973, pp. 51–70.

[Ton91] Alberto Tonelli. "Bemerkung über die Auflösung quadratischer Congruenzen". In: *Nachrichten von der Königlichen Gesellschaft der Wissenschaften und der Georg-Augusts-Universität zu Göttingen* (1891), pp. 344–346.

[XZZPS19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *CRYPTO '19*. 2019.

[Yal] Timofey Yaluhin. *zk-timelock*. URL: https://github.com/timothy/zk-timelock.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. "Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof". In: *S&P '20*. 2020.

[Zkc] *Families of SNARK-friendly 2-chains of elliptic curves*. URL: https://gitlab.inria.fr/zk-curves/snark-2-chains.

[plo] https://blog.polygon.technology/introducing plonky2/. "Introducing Plonky2". In.