# Fast* Implementations of Finite Fields and Elliptic Curves in Lean 4

Matej Penciak

Lurk Lab

March 23, 2023

Slides available at:

TODO: INCLUDE LINK

# Outline

1. Motivation: what we're building at Lurk Lab
2. A bit about zero knowledge cryptography
3. Optimized FF arithmetic
   - Add chains for fast exponentiation
   - Batched inversion
   - Montgomery form
4. Optimized Elliptic Curve arithmetic
   - Point representation
   - Efficient endomorphisms
   - Multi-scalar multiplication
5. Brief demo!
6. Lessons learned and future outlook

# Motivation

**Who am I?** Software engineer and aspiring cryptographer at Lurk Lab (Formerly Yatima).

**What is Lurk Lab?** We're building the Lurk programming language, and an ecosystem of tools around it. Examples: The Yatima compiler and content addresser, a Lean typechecker, and the cryptography to generate "Lurk proofs"

**What is Lurk?** Lurk is a turing complete programming language in the style of Lisp which generates zero knowledge proofs of execution.
If I run a Lurk program `f`, I can prove things like:

- I have run the computation of `f` applied to `u` and gotten `v`
- I know an input `u` to `f` such that `f(u) = v` without revealing `u`
- I can commit ahead of time to a program `f` and run it on any inputs $\{u\}$ you provide with proofs of execution.

# A little about zero knowledge proofs

Modern perspective on zero knowledge proofs: they are built out of two components

- An interactive proof scheme (the "information-theoretic" component)
- A commitment scheme (the "cryptographic" component)

An IP scheme is a protocol between two parties `P` (prover) and `V` (verifier) where (the potentially untrustworthy) `P` can provide a convincing argument to `V` that they know some fact/have performed some computation.

# An example IP

P and V agree to two graphs $G_1$ and $G_2$, and P wants to prove to V that they know an isomorphism $\phi : G_1 \to G_2$ without revealing the isomorphism.

1. P chooses a random permutation $\sigma$ of $G_1$, and sends $H = \sigma(G_1)$ to V
2. V chooses randomly between $\{1, 2\}$ and sends the choice $i$ to P
3. P responds with an isomorphism $\phi_i : G_i \to H$
4. V checks that in fact $\phi_i(G_i) \cong H$
5. Repeat until V is sufficiently convinced

This protocol is **complete**, **knowledge sound**, and **zero knowledge**
It is not: **non-interactive**, or **succinct**
The goal is to build **Z**ero **K**nowledge **S**uccinct **Ar**guments of **K**nowledge (ZK SNARKs) in Lean.

# The basic building blocks for cryptography

Add some cryptography to the IPs:
Work over finite fields, and encode our computations in so-called "arithmetic circuits".
The level of security necessary is measured in the number of "bits" of security. To prevent a dishonest prover from forging a fraudulent proof, a good estimate is $\approx 128$ bits of security.
**Example 1**: The fastest (publicly) known factorization/discrete logarithm algorithm is the general number field sieve method with complexity

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)(logn)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}\right)$$

To achieve $\approx 128$ bits of security we need to work over finite fields of size $\approx 2^{2048}$

# Elliptic curve cryptography

**Example 2**: Let $E$ be an elliptic curve with a cyclic subgroup of points of size $N$.
The best (publicly) known discrete logarithm algorithm (given $P = a \cdot G$, find $a \in \mathbb{N}$) is the Pollard-$\rho$ method which has complexity $O(\sqrt{N})$.
To achieve $\approx 128$ bits of security we only need a curve with $\approx 2^{256}$ number of points.
Thanks to Hasse's bounds, it's "easy" to find elliptic curves with enough points, if $E$ is an elliptic curve defined over a finite field with $q$ elements, then the number of points $N$ is bounded by:

$$|N - (q + 1)| \leq 2\sqrt{q}$$

When we discuss curves, we will see some magical curves with extra nice properties

# Big numbers in Lean

Most computers don't natively support arithmetic for numbers larger than $2^{64} - 1$, stored in the Lean datatype `UInt64`.
Lean 4's `Nat` and `Int` types are built on multiprecision arithmetic libraries written in C(++), and linked in via the `@[extern "..."]` attribute

```
@[extern "lean_nat_add"]
protected def Nat.add : (@& Nat) → (@& Nat) → Nat
    | a, Nat.zero   => a
    | a, Nat.succ b => Nat.succ (Nat.add a b)
```

which calls the C function defined in `lean.h`

```
static inline lean_obj_res
    lean_nat_add(b_lean_obj_arg a1, b_lean_obj_arg a2) {
      . . .
}
```

# Multiprecision arithmetic libraries

Lean uses the **G**NU **M**ulti**P**recision library on Linux based systems. Mac and Windows get a home-brewed version written in the Lean 4 C++ runtime code.
Both implementations represent `Nats` as arrays of `UInt64`s, and define arithmetic in base $2^{64}$. If $n$ is the number of "limbs" (`uint64_ts` in C) then the efficiency of some of the most commonly employed algorithms are:

1. Schoolbook multiplication: $O(n^2)$
2. Karatsuba multiplication: $O(n^{1.58})$
3. Strassen FFT algorithm: $O(n \log n \log \log n)$

Because our goal is to work in a fixed-precision finite field, we tried replacing the natural choice of `Nat` with a fixed precision arithmetic based off Lean's `ByteArray` type.
This turned out to be $\approx 100\times$ slower than `Nat`, so we quickly dropped that plan.

# A comment on benchmarking and testing

Testing and benchmarking is very important for non-formalized systems. For testing we have built a testing framework called LSpec which is expressive and easy to use.

```
#lspec check "add_comm"
    (forall n m : Nat, n + m = m + n)
#lspec check "not true"
    (forall n m : Nat, n^2 + m^2 >= n^3 + m)
```

which returns

```
? add_comm
x not true
====================
Found problems!
n := 2
m := 0
issue: 8 <= 4 does not hold
(1 shrinks)
```

# A comment on benchmarking and testing

For benchmarking we have a rudimentary suite of tools for running benchmarks:

```
import YatimaStdLib.Benchmark
import YatimaStdLib.AddChain

open Benchmark Exp

def f1 : Nat → Nat := (37 ^ .)
def f2 : Nat → Nat := fastExp 37
def addChainBench : Comparison f1 f2 where
    inputs := #[10000000, 15000000,  ...]

def main (args : List String) : IO UInt32 :=
    benchMain args addChainBench.benchmark
```

# Benchmarking continued...

This can be run as `Benchmarks-AddChain --num 50 --log out.bench` and times the benchmarks and writes to the file:

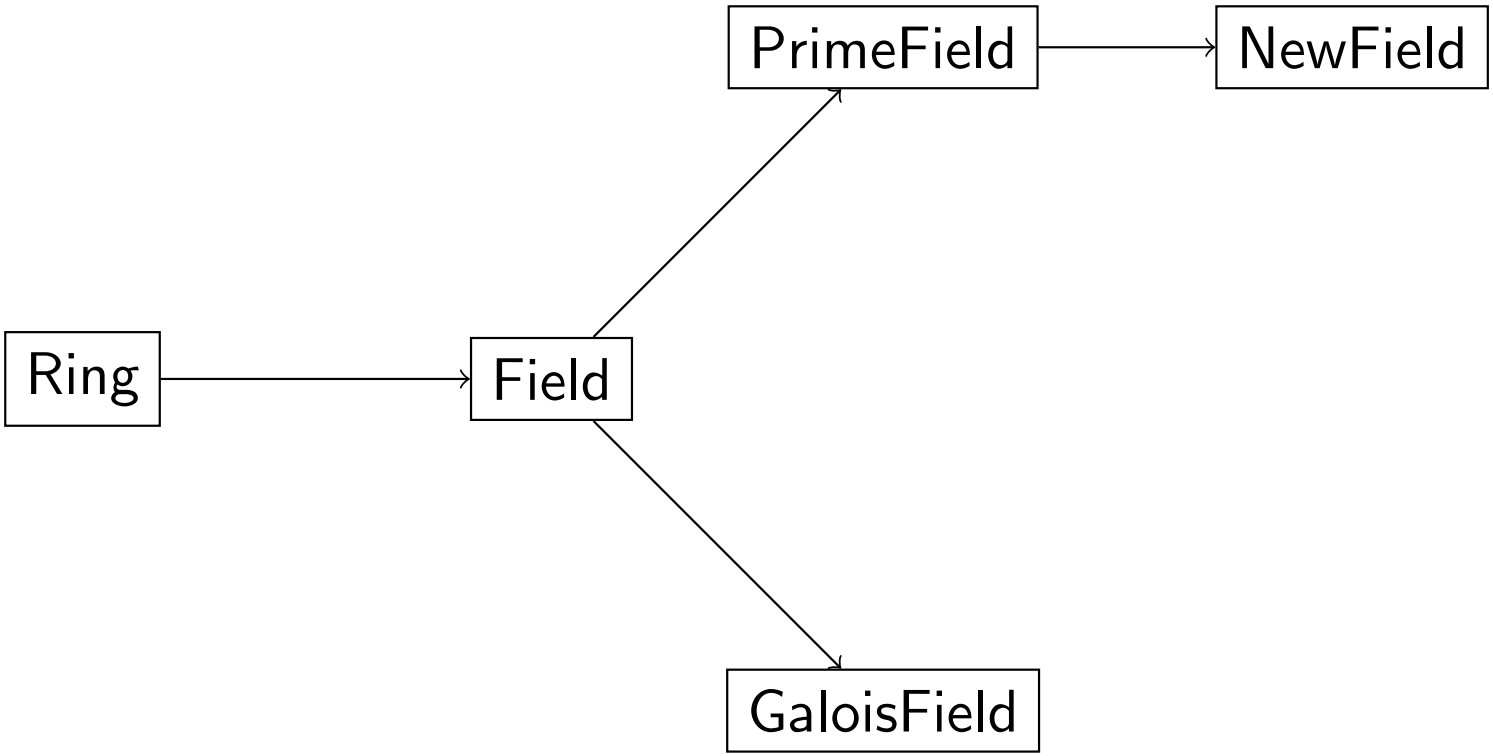
```
10000000: f:213993503 vs g:236587657
15000000: f:315046597 vs g:335218176
20000000: f:457342988 vs g:537591730
25000000: f:598272104 vs g:681122849
30000000: f:670924537 vs g:743308727
35000000: f:786821581 vs g:940891049
40000000: f:950924114 vs g:1103080520
45000000: f:1108323608 vs g:1213150412
50000000: f:1242277288 vs g:1288141664
```

# Back to fields

The arithmetic typeclasses we have (so far):

- `Ring`: Basic arithmetic: $+$, $*$, and so on
- `Field`: Ring together with inversion
- `PrimeField`: Arithmetic, plus pre-computations and methods to improve efficiency
- `NewField`: Automatically generated instance of a `PrimeField` with special methods related to Montgomery form
- `GaloisField`: Fields obtained as extension fields of a base field

# Field typeclasses




(thanks ChatGPT)

# Example classes

```
class Ring (R : Type _) extends Add R, Mul R, Sub R,
        HPow R Nat R, BEq R, Coe Nat R where
    zero : R
    one : R

class PrimeField (K : Type _) extends Field K where
    char : Nat -- characteristic
    sqrt : K → Option K -- Efficient `sqrt` implementation
    content : Nat -- char.log2
    twoAdicity : Nat x Nat -- `(s, r)` where `p-1 = 2^s * r'
    legAC : Array ChainStep -- Pre-computed addchains to
    frobAC : Array ChainStep -- calculate ^p and ^(p-1)/2
    fromNat : Nat → K -- to and from `Nat`' methods
    natRepr : K → Nat
    batchedExp : K → Array Nat → Array K
    batchedInv : Array K → Array K
```

# AddChains, batched arithmetic, and Montgomery form

**AddChains**: Exponentiation is a commonly used operation, but can be very slow.
Naive implementation: $a^N = a \cdot a^{N-1}$ is $O(N)$ (too slow)
Square and multiply method requires $\approx \frac{3}{2} \log N$ operations ($\log N$ squarings plus the average Hamming weight of $N$'s binary expansion)
(WARNING: susceptible to side channel attacks)
For common exponents ($p$ and $\frac{p-1}{2}$): pre-compute a minimal length "addition chain": $[n_1, n_2, n_3, ..., n_r = N]$ where each $n_i = n_k + n_l$ for $k, l < i$.
**Example**: Calculating the Frobenius modulo the prime
`0x4000000000000000000000000000000224698fc094cf91b992d30ed00000001`
381 field ops using square and multiply, and 337 field ops using a minimal addition chain.

# Batched inversion

**Batched inversion**:
Inversion is another costly operation (inversion in $\mathbb{Z}/p\mathbb{Z}$ is $O(\log p)$ using the extended Euclidean algorithm)
To invert $[n_1, n_2, n_3, \ldots, n_r]$, calculate $N_j = \prod_i^j n_i$ for $j = 1, \ldots, r$ caching the results. Invert $N_r$. Recover all of the $n_i^{-1}$ by multiplying with the cached results.
**Example:** $[2, 3, 5]$.

1. Calculate $N_1 = 2$, $N_2 = 6$, $N_3 = 30$.
2. Invert $N_3$ : $N_3^{-1} = 30^{-1}$, let $t = N_3^{-1}$
3. Multiply $t \cdot N_2 = 30^{-1} \cdot 6 = 5^{-1}$, and $t = t \cdot n_3 = 6^{-1}$
4. Proceed as before: $t \cdot N_1 = 6^{-1} \cdot 2 = 3^{-1}$, and $t = t \cdot n_2 = 2^{-1}$

This method replaces $r$ field inversions with 1 field inversion and $O(3r)$ field multiplications.

# Montgomery form

**Problem**: How should we calculate $n \cdot m$ in $\mathbb{F}_p$ when we use `Nat` representatives?
Naive algorithm: Calculate $a \cdot b \mod p$, but this requires division by $p$ for every multiplication.
**Intuition behind the solution**: Replace reduction modulo $p$ with reduction modulo $R$ for some other $R$.
Let $R > p$ be a number co-prime to $p$ for which division and reduction modulo $R$ is efficient.
In practice: Choose $R$ to be a power of 2 greater than $p$.
Montgomery reduction is an efficient algorithm that calculates $nR^{-1} \mod p$ using only arithmetic modulo $R$.
Represent $n$ in "Montgomery form" $[n]_w = n \cdot R \mod p$.
Addition: $[n]_w + [m]_w = (n + m) \cdot R \mod p = [n + m]_w$
Multiplication: $[n]_w \cdot [m]_w = (n \cdot m)R^2 \mod p = [n \cdot m]_w R \mod p$

# NewField implementation

We implement a `macro_rule` that expands the following declaration

```
new_field TestField with
    prime: 2011
    generator: 7
    root_of_unity: 2010 -- optional
```

into a type `TestField` with pre-computations and arithmetic, instances, and methods implemented.
Not advised to be used yet: Missing some key GMP methods to make Montgomery reduction more efficient (`mpz_tdiv_r_2exp`).
Part of Montgomery reduction of $a \in \mathbb{N}$ involves calculating $a \mod R$.

# Elliptic Curves

Short term goals have us working with curves defined over fields of large characteristic, so representing a curve is simple:

```
/--
Curves with Weierstrass form satisfying the equation
`y² = x³ + a x + b` for a prime field `F` such that
`char K > 3`
-/
structure Curve (F : Type _) [Field F] where
    a : F
    b : F
```

# Arithmetic on curves

Arithmetic is more interesting:

```
class CurveGroup {F : outParam $ Type _} [Field F] (K : Type _
    zero : K
    inv : K → K
    add : K → K → K
    double : K → K
```

Separating the curve from its arithmetic is important to allow for different coordinate systems on the same curve.
Projective coordinates avoid field inversions!
Affine coordinates are more compact on disk!
More exotic coordinate systems have other benefits.

# Coordinate breakdown

| Doubling | | Addition | |
|---|---|---|---|
| Operation | Costs | Operation | Costs |
| $2\mathcal{P}$ | $7\mathrm{M}+5\mathrm{S}$ | $\mathcal{J}^m+\mathcal{J}^m$ | $13\mathrm{M}+6\mathrm{S}$ |
| $2\mathcal{J}^c$ | $5\mathrm{M}+6\mathrm{S}$ | $\mathcal{J}^m+\mathcal{J}^c=\mathcal{J}^m$ | $12\mathrm{M}+5\mathrm{S}$ |
| $2\mathcal{J}$ | $4\mathrm{M}+6\mathrm{S}$ | $\mathcal{J}+\mathcal{J}^c=\mathcal{J}^m$ | $12\mathrm{M}+5\mathrm{S}$ |
| $2\mathcal{J}^m=\mathcal{J}^c$ | $4\mathrm{M}+5\mathrm{S}$ | $\mathcal{J}+\mathcal{J}$ | $12\mathrm{M}+4\mathrm{S}$ |
| $2\mathcal{J}^m$ | $4\mathrm{M}+4\mathrm{S}$ | $\mathcal{P}+\mathcal{P}$ | $12\mathrm{M}+2\mathrm{S}$ |
| $2\mathcal{A}=\mathcal{J}^c$ | $3\mathrm{M}+5\mathrm{S}$ | $\mathcal{J}^c+\mathcal{J}^c=\mathcal{J}^m$ | $11\mathrm{M}+4\mathrm{S}$ |
| $2\mathcal{J}^m=\mathcal{J}$ | $3\mathrm{M}+4\mathrm{S}$ | $\mathcal{J}^c+\mathcal{J}^c$ | $11\mathrm{M}+3\mathrm{S}$ |
| $2\mathcal{A}=\mathcal{J}^m$ | $3\mathrm{M}+4\mathrm{S}$ | $\mathcal{J}^c+\mathcal{J}=\mathcal{J}$ | $11\mathrm{M}+3\mathrm{S}$ |
| $2\mathcal{A}=\mathcal{J}$ | $2\mathrm{M}+4\mathrm{S}$ | $\mathcal{J}^c+\mathcal{J}^c=\mathcal{J}$ | $10\mathrm{M}+2\mathrm{S}$ |
| — | — | $\mathcal{J}+\mathcal{A}=\mathcal{J}^m$ | $9\mathrm{M}+5\mathrm{S}$ |
| — | — | $\mathcal{J}^m+\mathcal{A}=\mathcal{J}^m$ | $9\mathrm{M}+5\mathrm{S}$ |
| — | — | $\mathcal{J}^c+\mathcal{A}=\mathcal{J}^m$ | $8\mathrm{M}+4\mathrm{S}$ |
| — | — | $\mathcal{J}^c+\mathcal{A}=\mathcal{J}^c$ | $8\mathrm{M}+3\mathrm{S}$ |
| — | — | $\mathcal{J}+\mathcal{A}=\mathcal{J}$ | $8\mathrm{M}+3\mathrm{S}$ |
| — | — | $\mathcal{J}^m+\mathcal{A}=\mathcal{J}$ | $8\mathrm{M}+3\mathrm{S}$ |
| — | — | $\mathcal{A}+\mathcal{A}=\mathcal{J}^m$ | $5\mathrm{M}+4\mathrm{S}$ |
| — | — | $\mathcal{A}+\mathcal{A}=\mathcal{J}^c$ | $5\mathrm{M}+3\mathrm{S}$ |
| $2\mathcal{A}$ | $\mathrm{I}+2\mathrm{M}+2\mathrm{S}$ | $\mathcal{A}+\mathcal{A}$ | $\mathrm{I}+2\mathrm{M}+\mathrm{S}$ |

# Naiive vs. Actual implementation

```
let mut Y₃ := b₃ * t₂
Y₃ := X₃ + Y₃
X₃ := t₁ - Y₃
Y₃ := t₁ + Y₃
Y₃ := X₃ * Y₃
X₃ := t₃ * X₃
Z₃ := b₃ * Z₃
t₂ := a * t₂
t₃ := t₀ - t₂
t₃ := a * t₃
t₃ := t₃ + Z₃
Z₃ := t₀ + t₀
t₀ := Z₃ + t₀
t₀ := t₀ + t₂
t₀ := t₀ * t₃
Y₃ := Y₃ + t₀
t₂ := Y * Z
t₂ := t₂ + t₂
t₀ := t₂ * t₃
X₃ := X₃ - t₀
Z₃ := t₂ * t₁
Z₃ := Z₃ + Z₃
Z₃ := Z₃ + Z₃
return (X₃, Y₃, Z₃)
```

Reason: Ensure Lean is evaluating the arithmetic expressions to minimize finite field arithmetic

# Scalar multiplication and GLV optimization

Given $n \in \mathbb{N}$ and $P \in C$, want to calculate $n \cdot P$.
If $C$ has prime order, and an efficiently computable endomorphism $\Phi : C \to C$ then $\Phi(P) = \Lambda \cdot P$ for some $\Lambda \in \mathbb{N}$
Use this to compute

$$n \cdot P = (n_1 + \Lambda n_2) \cdot P = n_1 \cdot P + n_2 \cdot \Phi(P)$$

where $n_1$ and $n_2$ are $O(\sqrt{n})$
This optimization was subject to the patent US7995752B2 "Method for accelerating cryptographic operations on elliptic curves" which expired in 2020
If the curve $C$ is defined as $y^2 = x^3 + b$ then $(x : y : z) \mapsto (\zeta x : y : z)$ for $\zeta \in \mu_3$ is a candidate endomorphism.

# Some very magical curves

The curve is $y^2 = x^3 + 5$ over
`0x40000000000000000000000000000000224698fc094cf91b992d30ed00000001`
(Pallas) and
`0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001`
(Vesta)
Nice properties of the Pasta curves:

1. The group of points on one is cyclic of order equal to the characteristic of the other's base field.
2. Efficiently computable endomorphism for the GLV optimization.
3. Large 2-adicity ($p - 1$ is divisible by $2^{32}$)
4. Both primes are of the form $2^{255} + \epsilon$ which helps with reduction in $\mathbb{F}_p$
5. Both have low-degree isogenies with non-zero $j$-invariant (this helps with generating random points on the curves).

The story of how the curve was discovered is fascinating.

# Multiscalar Multiplication

The multi-exponentiation problem is the following: Given a set of list of integers $[n_1, n_2, \ldots, n_r]$ and a list of group elements $[g_1, g_2, \ldots, g_r]$ calculate $\prod_i^r g_i^{n_i}$.
Multi-exponentiation in the context of elliptic curve groups asks to calculate $\sum_i^r n_i \cdot G_i$ for points $G_i \in C$.
Multiscalar multiplication is used throughout cryptography (signature schemes, commitment schemes, zero knowledge proof schemes).
In some protocols, $\approx 80\%$ of time in generating zero knowledge proofs is spent calculating multiscalar multiplications.
Typical $r$ varies greatly by application, but can get up to sizes like $2^{28}$.

# Pippenger's algorithm

Pippenger's algorithm is one of the most efficient optimizations over the naive implementation.
Suppose the elliptic curve group order is approximately on the order of $b$ bits. Pick a "window size" $c$ ($\log r$ turns out to be optimal) so that $b = k \cdot c$.

1. Decompose each $n_i = n_{i,0} + n_{i,1} \cdot 2^c + n_{i,2} \cdot 2^{2c} + \ldots + n_{i,k} \cdot 2^{kc}$
2. Split up the large $b$-bit MSM into $k$ separate $c$-bit MSMs:

$$T_i = \sum_j^r n_{j,i} G_i$$

3. Solve each $c$-bit MSM and recombine as $M = \sum_i^k 2^i \cdot T_i$. (perfect opportunity for parallelization)

# Pippenger algorithm

We have reduced to the case of an MSM $T = \sum_i^r n_i \cdot G_i$ where each $n_i$ is at most $c$-bits.

1. Keep track of $2^c - 1$ "buckets", and place each $G_i$ into the $n_i$ bucket.
2. Sum up all the sets of points to get $2^c$ bucket sums $S_j$ for $j = 0, \ldots, 2^c - 1$
3. $T = S_{2^c-1} + (S_{2^c-1} + S_{2^c-2}) + \ldots + (S_{2^c-1} + S_{2^c-2} + \ldots + S_0)$

The Lean implementation provides an $\approx 10\times$ speedup over the naive implementation.
Further optimizations exist when each $G_i$ is a fixed shape relative to a generator for the elliptic curve group.
A common case is when $G_i = \tau^i \cdot G$ for $i = 0, \ldots, r$ and $G$ is a fixed generator for the elliptic curve group.

# A small Demo!

# Lessons learned

1. Squeezing performance out of Lean can feel like squeezing water out of stone sometimes.
2. Lean 4 does not have "zero cost abstractions".
3. Lean 4 is early in its life, so there are few to no guides written down for writing efficient code.
4. There are some significant performance pits one can fall in if one isn't careful with the reference counting.
5. Lean 4 is incredibly expressive, it shines in terms of the ease by which one can prototype complex structures and algorithms.

# Future goals

1. (short term) Optimize, optimize, optimize! (deeper GMP integration, more curve forms, ...)
2. (short-medium term) Expand `FFaCiL` to include more elliptic curve cryptography (isogenies!)
3. (medium term) Begin exploring the formalization of provable security (formalizing adversarial games, and complexity)
4. (long term) Expand the Yatima Standard Library with more number theoretic primitives.
5. (Long term) Begin the process of formally verifying these algorithms!
6. (Epochs from now) Combine all of these efforts together into a `cryptolib` with formally verified cryptographic algorithms.

# Thank you for your time!

Thanks!

# References

TODO