Chaos Background

Tristan Britt (tristan@warlock.xyz), oxAlcibiades (alcibiades@warlock.xyz)

July 2024

Contents

Glossary of Notation

Notation	Meaning
$\{\}$	Set delimiters
$\varnothing$	Empty set
$\in$	Element of
$\notin$	Not an element of
$\subset$	Proper subset
$\subseteq$	Subset or equal to
$\supset$	Proper superset
$\supseteq$	Superset or equal to
$\cup$	Union
$\cap$	Intersection
$\setminus$	Set difference
$\overline{A}$	Complement of set A
A^c	Complement of set A (alternative notation)
$\mathcal{P}(A)$	Power set of A
$A \times B$	Cartesian product of sets A and B
$\lvert A \rvert$	Cardinality (size) of set A
$\aleph_0$	Cardinality of the natural numbers (countable infinity)
$\mathfrak{c}$	Cardinality of the real numbers (continuum)
$\forall$	For all
$\exists$	There exists
$\exists!$	There exists a unique
: or $\mid$	Such that
$\{x \in A \mid P(x)\}$	Set-builder notation: set of all x in A such that $P(x)$ is true
$[a, b]$	Closed interval from a to b
(a, b)	Open interval from a to b
$[a, b)$ or $(a, b]$	Half-open intervals
$A \triangle B$	Symmetric difference of sets A and B
$\bigsqcup$	Disjoint union
$\bigcup_{i \in I} A_i$	Union of a family of sets
$\bigcap_{i \in I} A_i$	Intersection of a family of sets
A^n	Cartesian product of A with itself n times
$f : A \to B$	Function f from set A to set B
$f(A)$	Image of set A under function f
$f^{-1}(B)$	Preimage of set B under function f
$\mathrm{dom}(f)$	Domain of function f
$\mathrm{cod}(f)$	Codomain of function f
$\mathrm{range}(f)$	Range of function f
id_A	Identity function on set A
$f \circ g$	Composition of functions f and g
$f\vert_A$	Restriction of function f to set A
$f : A \twoheadrightarrow B$	Surjective function from A to B
$f : A \hookrightarrow B$	Injective function from A to B
$f : A \xrightarrow{\sim} B$	Bijective function from A to B
$\mathbb{Z}$	set of all integers
$\mathbb{Q}$	set of all rational numbers
$\mathbb{R}$	set of all real numbers
$\mathbb{C}$	set of all complex numbers

Notation	Meaning
$\Leftrightarrow$, iff	if and only if
$\mathbb{Z}^+, \mathbb{Q}^+, \mathbb{R}^+$	sets of all positive integers, rational numbers, and real numbers, respectively
$a \mid b$	a divides b
$*$	binary operation
Δ	symmetric difference
e	identity element of a group
$GL(2, \mathbb{R})$	general linear group of degree 2 over $\mathbb{R}$
$P(X)$	set of subsets X
$\mathbb{Z}_n$	the set $\{0, 1, 2, \ldots, n-1\}$
$a \equiv b \pmod{n}$	the integers a and b are congruent modulo n
$\oplus, \otimes$	addition and multiplication modulo n
$o(x)$	order of the element x
$\langle x \rangle$	set of powers of the element x
$\|G\|$	order of the group G
V	Klein's 4-group
$Z(G)$	center of the group G
$GL(2, \mathbb{C})$	general linear group of degree 2 over $\mathbb{C}$
Q_8	group of unit quaternions
$SL(2, \mathbb{R})$	special linear group of degree 2 over $\mathbb{R}$
$Z(g)$	centralizer of the element g
$G \times H$	direct product of G and H
$f : S \to T$	f is a function from S to T
f^{-1}	the inverse of the function f
$g \circ f$	composite function
i_X	identity function on the set X
S_X	symmetric group on X
S_n	symmetric group of degree n
A_n	alternating group of degree n
D_4	group of symmetries of a square
$x \equiv_H y$	means $xy^{-1} \in H$
$x_H \equiv y$	means $x^{-1}y \in H$
$[G : H]$	the index of H in G
$H \triangleleft G$	H is a normal subgroup of G
G/H	quotient group of G by H
$G \cong H$	G and H are isomorphic
$\varphi^{-1}(J)$	inverse image of J under φ
$\mathrm{Aut}(G)$	group of automorphisms of the group G
ρ	canonical homomorphism
$\ker(\varphi)$	kernel of the homomorphism φ
$N(H)$	normalizer of the subgroup H
$R \oplus S$	direct sum of the rings R and S
$M_2(\mathbb{R})$	ring of all 2×2 real matrices
$\mathbb{Z}[i]$	ring of Gaussian integers
$\mathbb{H}$	ring of quaternions
R/I	quotient ring of R by I
$R[X]$	polynomial ring over R
$F(a)$	field obtained by adjoining a to the field F
$\mathrm{irr}(a/F)$	irreducible polynomial of a over F
$\deg(a/F)$	degree of a over F

Notation	Meaning
$[E : F]$	degree of the field E over the field F
$\mathbb{C}_c$	field of constructible complex numbers
$\Gamma(E/F)$	Galois group of E over F
$\Phi(H)$	fixed field of the subgroup H of $\Gamma(E/F)$
$\Gamma(f(X)/F)$	Galois group of $f(X)$ over F

Math Primer for BLS and BN254

We'll begin with fundamental concepts in set theory, group theory, and ring theory. These will provide the basis for understanding more advanced structures like fields and vector spaces. Number theory and Euclidean domains will be introduced to provide essential tools for cryptographic applications.

As we progress, we'll delve into algebraic varieties and elliptic curves, which are crucial for understanding the BN254 curve. We'll then explore bilinear pairings, which are fundamental to the BLS signature scheme.

Finally, we'll apply these concepts to the specific case of the BN254 curve and the BLS signature scheme, culminating in an exploration of threshold signatures, R1CS, and distributed multi-party computation.

By the end of this primer, readers will have a solid grasp of the mathematical concepts necessary to understand and implement BLS threshold signatures using the BN254 pairing. This knowledge is crucial for developing secure and efficient cryptographic protocols in various applications, including blockchain technology and distributed systems.

The roadmap of our journey will be Sets → Groups → Rings → Domains → Fields → Vector Spaces → Algebraic Varieties → Elliptic Curves → Bilinear Pairings -> BN254 -> BLS -> BLS Thresholds -> R1CS -> Distributed MPC, rougly.

Set Theory

Set theory forms the bedrock of modern mathematics. It provides us with a language to discuss collections of objects and the relationships between them. For a more in-depth treatment of set theory, Halmos' "Naive Set Theory" and Suppes' "Axiomatic Set Theory" are superb resources.

Basic Definitions

1. A set is a collection of distinct objects, called elements or members of the set.
2. If a is an element of set A, we write $a \in A$.
3. The empty set, denoted $\varnothing$, is the unique set with no elements.
4. A set A is a subset of set B, denoted $A \subseteq B$, if every element of A is also an element of B.
5. Every set $A \subseteq A$, or in other words, every set is contained by itself.

Set Operations

Set theory defines several operations on sets:

1. Union: $A \cup B = \{x : x \in A \text{ or } x \in B\}$ The union of two sets contains all elements that are in either set.
2. Intersection: $A \cap B = \{x : x \in A \text{ and } x \in B\}$ The intersection contains all elements common to both sets.
3. Difference: $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$ The difference contains elements in A but not in B.
4. Symmetric Difference: $A \triangle B = (A \setminus B) \cup (B \setminus A)$ This operation results in elements that are in either set, but not in both.

Cartesian Product

The Cartesian product of two sets A and B, denoted $A \times B$, is the set of all ordered pairs where the first element comes from A and the second from B:

$$A \times B = \{(a, b) : a \in A \text{ and } b \in B\}$$

Functions

A function f from set A to set B, denoted $f : A \to B$, is a rule that assigns to each element of A exactly one element of B. We call A the domain and B the codomain of f. The set of all $f(a)$ for $a \in A$ is called the range of f.

Functions can have special properties:

1. Injective (one-to-one): $\forall a_1, a_2 \in A, f(a_1) = f(a_2) \implies a_1 = a_2$
2. Surjective (onto): $\forall b \in B, \exists a \in A : f(a) = b$
3. Bijective: Both injective and surjective

Cardinality

The cardinality of a set A, denoted $|A|$, is the number of elements in A if A is finite. For infinite sets, cardinality becomes more complex:

- Countably infinite: A set with the same cardinality as the natural numbers, denoted $\aleph_0$.
- Uncountable: An infinite set that is not countably infinite, such as the real numbers, with cardinality denoted $\mathfrak{c}$.

Group Theory

Group theory is the study of symmetries and algebraic structures. Professor Macauley's Visual Group Theory lectures on YouTube and Nathan Carter's "Visual Group Theory" book provide a beautiful and approachable exposition. Saracino's "Abstract Algebra" is approachable but in need of fresh typesetting. Lang's "Algebra" is also a good resource here and more generally on rings and fields to come.

Groups

A group is an ordered pair $(\mathbb{G}, *)$ where $\mathbb{G}$ is a set and $*$ is a binary operation on $\mathbb{G}$ satisfying four axioms:

1. Closure: $\forall a, b \in \mathbb{G}, a * b \in \mathbb{G}$
2. Associativity: $\forall a, b, c \in \mathbb{G}, (a * b) * c = a * (b * c)$
3. Identity: $\exists e \in \mathbb{G}, \forall a \in \mathbb{G} : a * e = e * a = a$
4. Inverse: $\forall a \in \mathbb{G}, \exists a^{-1} \in \mathbb{G} : a * a^{-1} = a^{-1} * a = e$

The identity element is often denoted as e, and the inverse of an element a is written as a^{-1}. We also have "subtraction" defined through the binary operator of the inverse of an element.

Abelian Groups

An Abelian group, named after Norwegian mathematician Niels Henrik Abel, is a group which is commutative under the binary operation $*$. A group $\mathbb{G}$ is abelian if $a * b = b * a, \forall a, b \in \mathbb{G}$.

Finite Groups

A group $\mathbb{G}$ is finite if the number of elements in $\mathbb{G}$ is finite, which then has cardinality or order $|\mathbb{G}|$.

Lagrange's Theorem

For a finite group $\mathbb{G}$ with $a \in \mathbb{G}$ and let there exist a positive integer d such that a^d is the smallest positive power of a that is equal to e, the identity of the group. Let $n = |\mathbb{G}|$ be the order of $\mathbb{G}$, and let d be the order of a, then $a^n = e$ and $d \mid n$.

Subgroups

A subset H of a group $\mathbb{G}$ is a subgroup if it forms a group under the same operation as $\mathbb{G}$. We denote this as $H \leq \mathbb{G}$. The order of a subgroup always divides the order of the group (Lagrange's Theorem). Similar to a set, every group $\mathbb{G} \subseteq \mathbb{G}$, and for every group there is a trivial subgroup containing only the identity.

Homomorphisms and Isomorphisms

A function $f : G \to H$ between groups is a homomorphism if it preserves the group operation: $f(ab) = f(a)f(b) \forall a, b \in G$.

An isomorphism is a bijective homomorphism. If there exists an isomorphism between groups G and H, we say they are isomorphic and write $G \cong H$.

Cosets and Normal Subgroups

For a subgroup H of G and an element $a \in G$, we define:

- Left coset: $aH = \{ah : h \in H\}$
- Right coset: $Ha = \{ha : h \in H\}$

A subgroup N of G is called normal if $gN = Ng \forall g \in G$. We denote this as

$$N \triangleleft G$$

.

Cyclic Groups

A group $\mathbb{G}$ is cyclic if there exists an element $g \in \mathbb{G}$ such that every element of $\mathbb{G}$ can be written as a power of g:

$$\mathbb{G} = \langle g \rangle = \{g^n : n \in \mathbb{Z}\}$$

Here, g is called a generator of $\mathbb{G}$. Cyclic groups have several important properties:

1. Every element $x \in \mathbb{G}$ can be written as $x = g^n$ for some integer n.
2. If $\mathbb{G}$ is infinite, it is isomorphic to $(\mathbb{Z}, +)$.
3. If $\mathbb{G}$ is finite with $|\mathbb{G}| = n$, it is isomorphic to $(\mathbb{Z}/n\mathbb{Z}, +)$.
4. All cyclic groups are Abelian.
5. Subgroups of cyclic groups are cyclic.
6. The order of $\mathbb{G}$ is the smallest positive integer m such that $g^m = e$.

Quotient Groups

If $N \triangleleft G$, we can form the quotient group G/N, whose elements are the cosets of N in G.

Ring Theory

Rings

A ring $(R, +, \cdot)$ is an algebraic structure consisting of a set R with two binary operations, addition $(+)$ and multiplication $(\cdot)$, satisfying the following axioms:

1. $(R, +)$ is an abelian group:
   - Closure: $\forall a, b \in R, a + b \in R$
   - Associativity: $\forall a, b, c \in R, (a + b) + c = a + (b + c)$
   - Commutativity: $\forall a, b \in R, a + b = b + a$
   - Identity: $\exists 0 \in R, \forall a \in R, a + 0 = 0 + a = a$
   - Inverse: $\forall a \in R, \exists (-a) \in R, a + (-a) = (-a) + a = 0$
2. $(R, \cdot)$ is a monoid:
   - Closure: $\forall a, b \in R, a \cdot b \in R$
   - Associativity: $\forall a, b, c \in R, (a \cdot b) \cdot c = a \cdot (b \cdot c)$
3. Distributivity:
   - Left distributivity: $\forall a, b, c \in R, a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
   - Right distributivity: $\forall a, b, c \in R, (a + b) \cdot c = (a \cdot c) + (b \cdot c)$

A ring is called commutative if multiplication is commutative, i.e., $\forall a, b \in R, a \cdot b = b \cdot a$. If a ring has a multiplicative identity element $1 \neq 0$ such that $\forall a \in R, 1 \cdot a = a \cdot 1 = a$, it is called a ring with unity.

Ideals

An ideal of a ring R is a subset $I \subseteq R$ where:

1. $(I, +)$ is a subgroup of $(R, +)$, meaning:
   a. I is non-empty
   b. For all $a, b \in I$, $a - b \in I$
2. For all $r \in R$ and $i \in I$, both $r \cdot i \in I$ and $i \cdot r \in I$ (absorption property)

The absorption property of ideals interacts with both ring operations, as it involves multiplication by any ring element and the result remains in the ideal.

Quotient Rings

For a ring R and ideal I, the quotient ring R/I is defined as:

$R/I = \{r + I : r \in R\}$

where $r + I = \{r + i : i \in I\}$ is the coset of r modulo I.

Operations in R/I are defined as:

1. Addition: $(a + I) + (b + I) = (a + b) + I$
2. Multiplication: $(a + I) \cdot (b + I) = (a \cdot b) + I$

These operations are well-defined because of the ideal properties, particularly the absorption property.

Polynomial Rings

Given a ring R, the polynomial ring $R[x]$ is defined as the set of all formal sums of the form:

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

where:

1. $n \in \mathbb{Z}^+$
2. $a_i \in R$ (called coefficients)
3. x is an indeterminate (or variable)
4. Only finitely many a_i are non-zero

The ring structure of $R[x]$ is defined by the following operations:

1. Addition: For $f(x) = \sum a_i x^i$ and $g(x) = \sum b_i x^i$,

   $(f + g)(x) = \sum_{i=0}^{\max(\deg(f),\deg(g))} (a_i + b_i)x^i$
2. Multiplication: For $f(x) = \sum a_i x^i$\$ and \$$g(x) = \sum b_i x^i$,

   $(f \cdot g)(x) = \sum_{k=0}^{\deg(f)+\deg(g)} (\sum_{i+j=k} a_i b_j)x^k$

Key properties:

1. The zero polynomial, denoted 0, has all coefficients equal to 0.
2. If R has a unity $1 \neq 0$, then $R[x]$ has a unity, which is the constant polynomial 1.
3. R is embedded in $R[x]$ as the set of constant polynomials.
4. If R is commutative, then $R[x]$ is commutative.
5. The degree of a non-zero polynomial $f(x)$, denoted $\deg(f)$, is the highest power of x with a non-zero coefficient.

This definition treats polynomials as formal algebraic objects, not as functions. The construction can be extended to multiple variables, e.g., $R[x, y] = (R[x])[y]$.

Number Theory

Prime Numbers and Divisibility

A prime number is a natural number greater than 1 that is only divisible by 1 and itself. The fundamental theorem of arithmetic states that every integer greater than 1 can be uniquely represented as a product of prime powers.

For integers a and b, we say a divides b (denoted $a \mid b$) if there exists an integer k such that $b = ak$. If $a \mid b$ and $a \mid c$, then $a \mid (bx + cy)$ for any integers x and y.

Greatest Common Divisor (GCD)

The greatest common divisor of two integers a and b, denoted $\gcd(a, b)$, is the largest positive integer that divides both a and b. Key properties include:

1. $\gcd(a, b) = \gcd(|a|, |b|)$
2. $\gcd(a, b) = \gcd(b, a \bmod b)$ (basis for the Euclidean algorithm)
3. There exist integers x and y such that $\gcd(a, b) = ax + by$ (Bézout's identity)

Integral Domains

An integral domain is a commutative ring with unity that has no zero divisors. In other words, for all non-zero elements $a, b \in R$, if $a \cdot b = 0$, then either $a = 0$ or $b = 0$. This property is crucial as it allows for cancellation in multiplication: if $a \cdot b = a \cdot c$ and $a \neq 0$, then $b = c$.

Euclidean Domains

A Euclidean domain is an integral domain R equipped with a function $\delta : R \setminus \{0\} \to \mathbb{N} \cup \{0\}$ (called the Euclidean function) satisfying:

1. For all non-zero $a, b \in R$, $\delta(a) \leq \delta(ab)$
2. For all $a, b \in R$ with $b \neq 0$, there exist $q, r \in R$ such that $a = bq + r$ and either $r = 0$ or $\delta(r) < \delta(b)$

The second property is known as the Euclidean division algorithm, which is a generalization of the division algorithm for integers. This algorithm allows us to perform division with remainders in the domain.

Examples of Euclidean Domains

1. The integers $\mathbb{Z}$ with $\delta(a) = |a|$
2. The polynomial ring $F[x]$ over a field F with $\delta(p) = \deg(p)$

Euclidean Division

Euclidean division is the process of dividing one integer by another to produce a quotient and a remainder. In the context of modular arithmetic, we're particularly interested in the remainder.

For integers a and b with $b \neq 0$, there exist unique integers q (quotient) and r (remainder) such that:

$a = bq + r$, where $0 \leq r < |b|$

Euclidean Division Algorithm Here's a pseudocode algorithm for Euclidean division:

```
function euclidean_division(a, b):
    if b == 0:
        error "Division by zero"
    q = floor(a / b)
```

```
    r = a - b * q
    if r < 0:
        if b > 0:
            q = q - 1
            r = r + b
        else:
            q = q + 1
            r = r - b
    return (q, r)
```

In $\mathbb{Z}_5$, we're primarily concerned with the remainder r, which will always be in the set $\{0, 1, 2, 3, 4\}$.

Extended Euclidean Division

The extended Euclidean Division is a way to compute the greatest common divisor (GCD) of two numbers a and b, and also find the coefficients of Bézout's identity, which states that:

$\gcd(a, b) = ax + by$

for some integers x and y.

Extended Euclidean Division Algorithm

```
function extended_euclidean_division(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        (gcd, x', y') = extended_gcd(b, a mod b)
        x = y'
        y = x' - floor(a / b) * y'
        return (gcd, x, y)
```

This algorithm not only computes the GCD but also finds the coefficients x and y in Bézout's identity.

Field Theory

Fields

A field F is a set with two binary operations defined over it and closed under it, usually addition ($+$) and multiplication ($\cdot$). The field F must satisfy the following axioms:

1. $(F, +)$ is an abelian group with identity element 0
2. $(F, +, \cdot)$ is a commutative ring with identity element 0
3. $(F \setminus \{0\}, \cdot)$ is an abelian group with identity element 1
4. Distributivity: $a \cdot (b + c) = (a \cdot b) + (a \cdot c) \forall a, b, c \in F$

Formally, a field is a commutative ring where every non-zero element has a multiplicative inverse. For every $a \in F, a \neq 0$, there exists $b \in F$ such that $a \cdot b = 1_F$.

Examples of infinite fields include the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$.

Division in Infinite Fields

In a field F, division is defined for all non-zero elements. For any $a, b \in F$ with $b \neq 0$, we define:

$a \div b = a \cdot b^{-1}$

where b^{-1} is the unique multiplicative inverse of b. This inverse always exists for non-zero elements in a field.

Every field is automatically a Euclidean domain, where we can define $\delta(a) = 0$ for all non-zero a. The Euclidean division algorithm simplifies in fields: for any $a, b \in F$ with $b \neq 0$, we can always find unique $q, r \in F$ such that:

$a = bq + r$

where $r = 0$, and $q = a \div b$.

Finite Fields

Finite fields, also known as Galois fields, are fields with a finite number of elements. They are denoted $GF(q)$ or $\mathbb{F}_q$, where $q = p^n$ for some prime p and positive integer n.

Key properties of finite fields include:

1. The order (number of elements) of a finite field is always a prime power.
2. For each prime power q, there exists a unique (up to isomorphism) finite field of order q.
3. The multiplicative group of a finite field is cyclic.

Modular Arithmetic in $\mathbb{Z}_p$

For any prime p, we define the prime field $\mathbb{Z}_p$ as the set of integers modulo p:

$\mathbb{Z}_p = \{0, 1, 2, ..., p - 1\}$

Example: In $\mathbb{Z}_5$, we have $\{0, 1, 2, 3, 4\}$.

Modular arithmetic is a system of arithmetic for finite fields and rings, where numbers "wrap around" when reaching a certain value, called the modulus.

All operations in $\mathbb{Z}_p$ are performed modulo p.

Addition and Subtraction For $a, b \in \mathbb{Z}_p$:

$a \oplus b = (a + b) \bmod p \; a \ominus b = (a - b) \bmod p$

Example: In $\mathbb{Z}_5$, the addition table is:

$\oplus$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Multiplication and Division For $a, b \in \mathbb{Z}_p$:

$a \otimes b = (a \times b) \bmod p$

Division is defined as multiplication by the multiplicative inverse.

Example: In $\mathbb{Z}_5$, the multiplication table is:

$\otimes$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

In finite fields, division is performed as follows:

1. For prime fields $\mathbb{F}_p$: $a \div b = a \cdot b^{-1} \pmod{p}$
2. For extension fields $\mathbb{F}_{p^n}$: $a(x) \div b(x) = a(x) \cdot b(x)^{-1} \pmod{f(x)}$

where $f(x)$ is the irreducible polynomial used to construct $\mathbb{F}_{p^n}$. In both cases, the multiplicative inverse can be computed using the Extended Euclidean Algorithm.

Euler's Totient Function Euler's Totient Function, denoted as $\phi(n)$ or $\varphi(n)$, counts the number of positive integers up to n that are relatively prime to n (i.e., their greatest common divisor with n is 1).

Definition For a positive integer n, $\phi(n)$ is the count of numbers k in the range $1 \leq k < n$ where $\gcd(k, n) = 1$.

Formula For a positive integer n with prime factorization $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$:

$$\phi(n) = n \prod_{i=1}^{k} (1 - \frac{1}{p_i})$$

Properties

1. For a prime number p, $\phi(p) = p - 1$
2. ϕ is multiplicative: if $\gcd(a, b) = 1$, then $\phi(ab) = \phi(a) \cdot \phi(b)$
3. For a prime power p^k, $\phi(p^k) = p^k - p^{k-1} = p^k(1 - \frac{1}{p})$

Examples

1. $\phi(10) = 4$, as 1, 3, 7, 9 are relatively prime to 10
2. $\phi(12) = 4$, as 1, 5, 7, 11 are relatively prime to 12
3. $\phi(15) = 8$, as 1, 2, 4, 7, 8, 11, 13, 14 are relatively prime to 15

Calculation Method

1. Find the prime factorization of n
2. For each prime factor p, multiply n by $(1 - \frac{1}{p})$
3. The result is $\phi(n)$

Fermat's Little Theorem and Euler's Theorem Fermat's Little Theorem states that for any integer a not divisible by p:

$a^{p-1} \equiv 1 \pmod{p}$

Example: In $\mathbb{Z}_5$, for any non-zero a, $a^4 \equiv 1 \pmod{5}$

We can verify this using the multiplication table: - $1^4 = 1 \equiv 1 \pmod{5}$ - $2^4 = 2 \otimes 2 \otimes 2 \otimes 2 = 4 \otimes 2 \otimes 2 = 3 \otimes 2 = 1 \pmod{5}$ - $3^4 = 3 \otimes 3 \otimes 3 \otimes 3 = 2 \otimes 3 \otimes 3 = 1 \otimes 3 = 3 \pmod{5}$ - $4^4 = 4 \otimes 4 \otimes 4 \otimes 4 = 1 \otimes 4 \otimes 4 = 4 \otimes 4 = 1 \pmod{5}$

Applications: 1. Finding multiplicative inverses: $a^{-1} \equiv a^{p-2} \pmod{p}$ Example: In $\mathbb{Z}_5$, $3^{-1} \equiv 3^3 \equiv 2 \pmod{5}$ 2. Efficient exponentiation: $a^n \equiv a^{n \bmod (p-1)} \pmod{p}$ for $a \neq 0$ Example: In $\mathbb{Z}_5$, $3^{10} \equiv 3^{10 \bmod 4} \equiv 3^2 \equiv 4 \pmod{5}$

Congruences and Residue Classes In $\mathbb{Z}_p$, two integers a and b are congruent if:

$a \equiv b \pmod{p}$

The residue classes in $\mathbb{Z}_p$ are:

$[i] = \{..., i - p, i, i + p, i + 2p, ...\}$ for $i = 0, 1, ..., p - 1$

Example: In $\mathbb{Z}_5$, the residue classes are: - $[0] = \{\ldots, -5, 0, 5, 10, \ldots\}$ - $[1] = \{\ldots, -4, 1, 6, 11, \ldots\}$ - $[2] = \{\ldots, -3, 2, 7, 12, \ldots\}$ - $[3] = \{\ldots, -2, 3, 8, 13, \ldots\}$ - $[4] = \{\ldots, -1, 4, 9, 14, \ldots\}$

Coprime Numbers Two integers a and b are considered coprime (or relatively prime) if their greatest common divisor (GCD) is 1. In other words:

$$\gcd(a, b) = 1$$

Some key properties of coprime numbers include:

1. If a and b are coprime, there exist integers x and y such that:

   $$ax + by = 1$$

   This is known as Bézout's identity.

2. If a and b are coprime, then:

   $(a \bmod b)$ has a multiplicative inverse modulo b

   This means there exists an integer x such that:

   $$ax \equiv 1 \pmod{b}$$

3. The product of coprime numbers is coprime to each of the original numbers.

To find coprime numbers, one can use the Euclidean algorithm to compute the GCD. If the GCD is 1, the numbers are coprime. For example:

- 8 and 15 are coprime because $\gcd(8, 15) = 1$
- 14 and 21 are not coprime because $\gcd(14, 21) = 7$

In the context of the Chinese remainder theorem, coprimality is a crucial requirement. The CRT states that if we have a system of congruences with coprime moduli:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

where all m_i are pairwise coprime, then there exists a unique solution modulo $M = m_1 \cdot m_2 \cdot \ldots \cdot m_k$.

Chinese Remainder Theorem The Chinese Remainder Theorem (CRT) states that if one has a system of congruences with coprime moduli, there exists a unique solution modulo the product of the moduli. Given a system of congruences:

$x \equiv a_1 \pmod{m_1}$ $x \equiv a_2 \pmod{m_2}$ $\vdots$ $x \equiv a_k \pmod{m_k}$

Where all m_i are pairwise coprime, there exists a unique solution x modulo $M = m_1 * m_2 * \ldots * m_k$. The solution can be constructed as: $x = \sum_{i=1}^{k} a_i * M_i * y_i \pmod{M}$ where $M_i = M/m_i$ and $y_i = M_i^{-1} \pmod{m_i}$.

Polynomial Modular Arithmetic in $\mathbb{Z}_p[x]$

Polynomial modular arithmetic in prime fields combines concepts from modular arithmetic and polynomial arithmetic over finite fields. Let $\mathbb{F}_p$ be a prime field with p elements, where p is prime.

Polynomial Ring $\mathbb{F}_p[x]$ The polynomial ring $\mathbb{F}_p[x]$ consists of all polynomials with coefficients from $\mathbb{F}_p$. A general element of $\mathbb{F}_p[x]$ has the form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where $a_i \in \mathbb{F}_p$ for all i.

Basic Operations

1. Addition and Subtraction: For $f(x) = \sum a_i x^i$ and $g(x) = \sum b_i x^i$, $(f + g)(x) = \sum (a_i \oplus b_i) x^i$ $(f - g)(x) = \sum (a_i \ominus b_i) x^i$
2. Multiplication: $(f \cdot g)(x) = \sum (\sum a_i \otimes b_j) x^{i+j}$
3. Division with Remainder: For $f(x)$ and $g(x) \neq 0$, there exist unique $q(x)$ and $r(x)$ such that: $f(x) = g(x)q(x) + r(x)$, where $\deg(r) < \deg(g)$

Modular Reduction

When working with polynomials modulo another polynomial $m(x)$, we perform operations and then reduce the result modulo $m(x)$. This is denoted as:

$f(x) \equiv g(x) \pmod{m(x)}$

which means $m(x)$ divides $f(x) - g(x)$.

Examples

1. In $\mathbb{Z}_5[x]$ mod (x^2+1): $(x+1)^2 \equiv x^2 + 2x + 1 \equiv 2x + 2$
2. In $\mathbb{Z}_5[x]$ mod (x^2+2): $(2x+1)(x+2) \equiv 2x^2 + 4x + x + 2 \equiv 2x^2 + 2x + 2 \equiv 3x + 3$

Irreducible Polynomials

A polynomial $f(x) \in \mathbb{Z}_p[x]$ is irreducible if it cannot be factored into the product of two non-constant polynomials in $\mathbb{Z}_p[x]$. Irreducible polynomials are crucial for constructing finite field extensions.

For example, $x^2 + 1$ is irreducible in $\mathbb{Z}_5[x]$ but reducible in $\mathbb{Z}_3[x]$ as $x^2 + 1 \equiv (x+1)(x+2) \pmod{3}$.

Subfields

Definition A subfield of a field F is a subset $K \subseteq F$ that is itself a field under the operations of F. More formally, K is a subfield of F if:

1. K is a subset of F
2. K is closed under the addition and multiplication operations of F
3. K contains the additive and multiplicative identities of F
4. Every element in K has an additive inverse in K
5. Every non-zero element in K has a multiplicative inverse in K

Properties

1. **Minimal Subfield**: Every field F contains a unique smallest subfield, called the prime subfield. It is isomorphic to either $\mathbb{Q}$ (if F has characteristic o) or $\mathbb{F}_p$ (if F has characteristic p).
2. **Tower Law**: If E is a subfield of F and F is a subfield of K, then $[K : E] = [K : F][F : E]$.
3. **Degree of Subfield**: If K is a subfield of F, then $[F : K]$ divides $[F : \mathbb{F}_p]$ where $\mathbb{F}_p$ is the prime subfield of F.
4. **Galois Correspondence**: In a Galois extension F/K, there is a one-to-one correspondence between the subfields of F containing K and the subgroups of the Galois group $\text{Gal}(F/K)$.

Examples

1. **Subfields of $\mathbb{C}$**:
   - $\mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$
   - $\mathbb{Q}(\sqrt{2}) \subset \mathbb{R}$
   - $\mathbb{Q}(i) \subset \mathbb{C}$
2. **Subfields of Finite Fields**: Let $\mathbb{F}_{p^n}$ be a finite field. Then $\mathbb{F}_{p^m}$ is a subfield of $\mathbb{F}_{p^n}$ if and only if m divides n.

   Example: Subfields of $\mathbb{F}_{2^4}$
   - $\mathbb{F}_2 \subset \mathbb{F}_{2^4}$

- $\mathbb{F}_{2^2} \subset \mathbb{F}_{2^4}$

3. **Algebraic Number Fields**: Consider $\mathbb{Q}(\sqrt{2}, \sqrt{3})$. Its subfields include:
   - $\mathbb{Q}$
   - $\mathbb{Q}(\sqrt{2})$
   - $\mathbb{Q}(\sqrt{3})$
   - $\mathbb{Q}(\sqrt{6})$

Field Extensions and Towers over Finite Fields

Let p be a prime number. We'll consider field extensions over $\mathbb{Z}_p = \mathbb{F}_p$, the finite field with p elements.

Basic Definitions

1. **Automorphism**: An automorphism of a field F is a bijective ring homomorphism from F to itself. The set of all automorphisms of F forms a group under composition.
2. **Endomorphism**: An endomorphism of a field F is a ring homomorphism from F to itself. Unlike automorphisms, endomorphisms are not necessarily bijective.
3. **Frobenius Endomorphism**: In a field of characteristic p, the map $\phi : x \mapsto x^p$ is an endomorphism called the Frobenius endomorphism. In finite fields, it's always an automorphism.

Field Extensions

A field extension E/F is a field E containing F as a subfield. The degree of the extension, denoted $[E : F]$, is the dimension of E as a vector space over F.

For a finite field $\mathbb{F}_p$, we can construct extensions $\mathbb{F}_{p^n}$ of degree n over $\mathbb{F}_p$.

Example in $\mathbb{Z}_5$: Let's construct $\mathbb{F}_{25}$ as an extension of $\mathbb{F}_5$.

1. Choose an irreducible polynomial $f(x) = x^2 + 2 \in \mathbb{F}_5[x]$.
2. $\mathbb{F}_{25} = \mathbb{F}_5[x]/(f(x)) = \{ax + b \mid a, b \in \mathbb{F}_5\}$
3. Arithmetic in $\mathbb{F}_{25}$ is performed modulo $f(x)$.

For instance, in $\mathbb{F}_{25}$:

$(3x + 4) * (2x + 1) = 6x^2 + 3x + 8x + 4 = 6x^2 + 11x + 4 \equiv 6(3) + x + 4 \equiv 3x + 4 \pmod{x^2 + 2}$

Towers of Field Extensions

A tower of field extensions is a sequence of fields $F_1 \subset F_2 \subset \ldots \subset F_n$ where each F_{i+1}/F_i is a field extension.

General construction for $\mathbb{F}_{p^n}$: We can build $\mathbb{F}_{p^n}$ as a tower of extensions over $\mathbb{F}_p$:

$\mathbb{F}_p \subset \mathbb{F}_{p^k} \subset \mathbb{F}_{p^m} \subset \mathbb{F}_{p^n}$

where $k|m|n$.

Example tower in $\mathbb{Z}_5$: Let's construct $\mathbb{F}_{625} = \mathbb{F}_5^4$ as a tower:

$\mathbb{F}_5 \subset \mathbb{F}_{25} \subset \mathbb{F}_{625}$

1. $\mathbb{F}_{25} = \mathbb{F}_5[x]/(x^2 + 2)$ as before
2. $\mathbb{F}_{625} = \mathbb{F}_{25}[y]/(y^2 + y + 2)$

In this tower: - $[\mathbb{F}_{25} : \mathbb{F}_5] = 2$ - $[\mathbb{F}_{625} : \mathbb{F}_{25}] = 2$

By the tower law: $[\mathbb{F}_{625} : \mathbb{F}_5] = 2 * 2 = 4$

Algebraicity

This is a topic that is advanced even for the scope of this revision, but it becomes important to consider later, and is a key concept of Galois theory. Given an extension $E \subset F$ and an element $\vartheta \in E$, the following conditions are equivalent: - ϑ is a root of $f(t) \neq \nvDash \in F[t]$ - $\{1, \vartheta, \vartheta^2, \cdots\}$ are linearly independent on F; - $F[\vartheta]$ is a field

ϑ is called algebraic over F if any of these conditions are met (and thus all of them). An extension $E \subset F$ is algebraic iff $\forall \vartheta \in E, \vartheta$ is algebraic. Also if $[E : F] < \infty$, E is algebraic over F.

You can also show that for $\vartheta \in E$, if $f(\vartheta) = 0$ for $f(t) = a_0 + a_1 t + \cdots + a_{n-1}t^{n-1} + a_n t^n$ with $a_i \in E$ algebraic, then ϑ is algebraic over F, aka addition and multiplication preserve algebraicity.

For prime order fields, all this means is that there is a unique (up to isomorphism) extension field $\mathbb{F}_q \supseteq \mathbb{F}_p$ of degree $[\mathbb{F}_q : \mathbb{F}_p] = r$ and order $q = p^r$. Namely:

$$\overline{\mathbb{F}}_p \triangleq \bigcup_{r=1}^{\infty} \mathbb{F}_{p^r}$$

Defining a morphism or curve, for example, over the algebraic closure of a finite field is a concise way to say that we're interested in points lying in all valid extensions that satisfy the curve equation, and that the mapping or what not behaves similarly for all of them.

Automorphisms and the Frobenius Endomorphism

In $\mathbb{F}_{p^n}$, the Frobenius endomorphism $\phi : x \mapsto x^p$ is an automorphism, and its powers generate the Galois group of $\mathbb{F}_{p^n}$ over $\mathbb{F}_p$.

Example in $\mathbb{F}_{25}$ over $\mathbb{F}_5$: The Frobenius automorphism ϕ on $\mathbb{F}_{25} = \mathbb{F}_5[x]/(x^2+2)$ is:

$\phi(ax+b) = (ax+b)^5 = a^5x^5 + b^5 = ax^5 + b \equiv a(3x) + b \pmod{x^2+2}$

The Galois group $\mathrm{Gal}(\mathbb{F}_{25}/\mathbb{F}_5) = \{\mathrm{id}, \phi\}$ is cyclic of order 2.

Vector Spaces

Vector spaces are fundamental algebraic structures that generalize the notion of vectors in two or three-dimensional space to any number of dimensions.

Definition of Vector Spaces

A vector space V over a field F is a set equipped with two operations:

1. Vector addition: $+ : V \times V \to V$
2. Scalar multiplication: $\cdot : F \times V \to V$

These operations must satisfy the following axioms for all $u, v, w \in V$ and $a, b \in F$:

1. $(u + v) + w = u + (v + w)$ (Associativity of addition)
2. $u + v = v + u$ (Commutativity of addition)
3. $\exists 0 \in V$ such that $v + 0 = v$ for all $v \in V$ (Additive identity)
4. For each $v \in V$, $\exists (-v) \in V$ such that $v + (-v) = 0$ (Additive inverse)
5. $a(u + v) = au + av$ (Distributivity of scalar multiplication over vector addition)
6. $(a + b)v = av + bv$ (Distributivity of scalar multiplication over field addition)
7. $(ab)v = a(bv)$ (Associativity of scalar multiplication)
8. $1v = v$ where 1 is the multiplicative identity in F

Linear Independence and Basis

A set of vectors $\{v_1, \ldots, v_n\}$ in a vector space V is linearly independent if the equation:

$$a_1 v_1 + a_2 v_2 + \cdots + a_n v_n = 0$$

implies $a_1 = a_2 = \cdots = a_n = 0$ for scalars $a_i \in F$.

A basis for a vector space V is a linearly independent set of vectors that spans V. In other words, every vector in V can be uniquely expressed as a linear combination of basis vectors.

Dimension and Subspaces

The dimension of a vector space V, denoted $\dim(V)$, is the number of vectors in any basis of V. A finite-dimensional vector space has a finite basis, while an infinite-dimensional space does not.

A subspace W of a vector space V is a subset of V that is itself a vector space under the operations inherited from V.

Concrete Example: Vector Space over $\mathbb{Z}_5$

To illustrate these concepts, let's consider a concrete example using the finite field $\mathbb{Z}_5$ (integers modulo 5). We'll explore the vector space $V = \mathbb{Z}_5 \times \mathbb{Z}_5$, which consists of ordered pairs (a, b) where $a, b \in \mathbb{Z}_5$. Our scalar field is $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

Vector Addition For $(a_1, b_1), (a_2, b_2) \in V$:

$$(a_1, b_1) + (a_2, b_2) = ((a_1 + a_2) \bmod 5, (b_1 + b_2) \bmod 5)$$

Example:

$$(2, 3) + (4, 1) = (1, 4)$$

Scalar Multiplication For $c \in \mathbb{Z}_5$ and $(a, b) \in V$:

$$c \cdot (a, b) = ((ca) \bmod 5, (cb) \bmod 5)$$

Example:

$$3 \cdot (2, 4) = (1, 2)$$

Verifying Vector Space Axioms Let's verify some of the vector space axioms using our $\mathbb{Z}_5 \times \mathbb{Z}_5$ example:

1. **Commutativity of addition:**

$$(2, 3) + (4, 1) = (1, 4) = (4, 1) + (2, 3)$$

2. **Associativity of addition:**

$$((2, 3) + (4, 1)) + (3, 2) = (1, 4) + (3, 2) = (4, 1)$$

$$(2, 3) + ((4, 1) + (3, 2)) = (2, 3) + (2, 3) = (4, 1)$$

3. **Additive identity:** The zero vector is $(0, 0)$

$$(2, 3) + (0, 0) = (2, 3)$$

4. **Additive inverse:** For $(2, 3)$, the additive inverse is $(3, 2)$

$$(2, 3) + (3, 2) = (0, 0)$$

5. **Distributivity of scalar multiplication over vector addition:**

$$3 \cdot ((2, 1) + (4, 3)) = 3 \cdot (1, 4) = (3, 2)$$

$$3 \cdot (2, 1) + 3 \cdot (4, 3) = (1, 3) + (2, 4) = (3, 2)$$

Linear Independence and Basis in $\mathbb{Z}_5 \times \mathbb{Z}_5$ In $\mathbb{Z}_5 \times \mathbb{Z}_5$, the vectors $(1, 0)$ and $(0, 1)$ form a basis. They are linearly independent because:

$$a(1, 0) + b(0, 1) = (0, 0)$$

implies $a = b = 0$ in $\mathbb{Z}_5$.

Every vector in $\mathbb{Z}_5 \times \mathbb{Z}_5$ can be uniquely expressed as a linear combination of these basis vectors:

$$(a, b) = a(1, 0) + b(0, 1)$$

Subspaces of $\mathbb{Z}_5 \times \mathbb{Z}_5$ Some examples of subspaces in $\mathbb{Z}_5 \times \mathbb{Z}_5$ include:

1. $\{(0, 0)\}$: The trivial subspace
2. $\{(a, 0) \mid a \in \mathbb{Z}_5\}$: A one-dimensional subspace
3. $\mathbb{Z}_5 \times \mathbb{Z}_5$ itself: The entire space

Dimension of $\mathbb{Z}_5 \times \mathbb{Z}_5$ The dimension of $\mathbb{Z}_5 \times \mathbb{Z}_5$ is 2, as it has a basis with two vectors. This means that any set of three or more vectors in $\mathbb{Z}_5 \times \mathbb{Z}_5$ must be linearly dependent.

Algebraic Varieties

Algebraic varieties are fundamental objects in algebraic geometry, providing a geometric perspective on solutions to systems of polynomial equations. We'll explore their definition, types, and key properties.

Definition of Algebraic Varieties

Let k be an algebraically closed field, and let $k[x_1, \ldots, x_n]$ be the ring of polynomials in n variables over k.

Definition 1 (Affine Algebraic Set): For a set of polynomials $S \subset k[x_1, \ldots, x_n]$, we define the affine algebraic set $V(S)$ as:

$$V(S) = \{(a_1, \ldots, a_n) \in k^n : f(a_1, \ldots, a_n) = 0 \text{ for all } f \in S\}$$

Definition 2 (Algebraic Variety): An algebraic variety is an irreducible algebraic set. That is, it cannot be written as the union of two proper algebraic subsets.

For any subset $X \subset k^n$, we define the ideal of X as:

$$I(X) = \{f \in k[x_1, \ldots, x_n] : f(a_1, \ldots, a_n) = 0 \text{ for all } (a_1, \ldots, a_n) \in X\}$$

Theorem 1 (Hilbert's Nullstellensatz): For any ideal $I \subset k[x_1, \ldots, x_n]$,

$$I(V(I)) = \sqrt{I}$$

where $\sqrt{I}$ is the radical of I.

This theorem establishes a fundamental correspondence between algebraic sets and radical ideals.

Affine and Projective Varieties

Affine Varieties An affine variety is an algebraic variety in affine space k^n.

Example: The parabola $y = x^2$ in k^2 is an affine variety defined by the polynomial $f(x, y) = y - x^2$.

Projective Varieties To define projective varieties, we first introduce projective space:

Definition 3 (Projective Space): The projective n-space over k, denoted $\mathbb{P}^n(k)$ or simply $\mathbb{P}^n$, is defined as:

$$\mathbb{P}^n = (k^{n+1} \setminus \{0\}) / \sim$$

where $\sim$ is the equivalence relation $(x_0, \ldots, x_n) \sim (\lambda x_0, \ldots, \lambda x_n)$ for any $\lambda \in k^*$.

A projective variety is an algebraic variety in projective space $\mathbb{P}^n$.

Definition 4 (Projective Variety): For a set of homogeneous polynomials $S \subset k[x_0, \ldots, x_n]$, we define the projective algebraic set $V(S)$ as:

$$V(S) = \{[a_0 : \ldots : a_n] \in \mathbb{P}^n : f(a_0, \ldots, a_n) = 0 \text{ for all } f \in S\}$$

A projective variety is an irreducible projective algebraic set.

Example: The projective conic $x^2 + y^2 = z^2$ in $\mathbb{P}^2$ is a projective variety.

Coordinate Rings

The coordinate ring of an algebraic variety encodes its algebraic structure.

Definition 5 (Coordinate Ring): For an affine variety $X \subset k^n$, the coordinate ring of X is:

$$k[X] = k[x_1, \ldots, x_n]/I(X)$$

For a projective variety $X \subset \mathbb{P}^n$, we define the homogeneous coordinate ring as:

$$S(X) = k[x_0, \ldots, x_n]/I(X)$$

where $I(X)$ is the homogeneous ideal of polynomials vanishing on X.

Properties of Algebraic Varieties

1. **Dimension:** The dimension of an algebraic variety X is defined as the transcendence degree of its function field $k(X)$ over k.
2. **Singular Points:** A point p on a variety X is singular if the rank of the Jacobian matrix at p is less than the dimension of X.
3. **Zariski Topology:** The Zariski topology on k^n or $\mathbb{P}^n$ is defined by taking algebraic sets as closed sets. This topology is fundamental in algebraic geometry.
4. **Morphisms:** A morphism between varieties $X \subset k^m$ and $Y \subset k^n$ is a function $\phi : X \to Y$ such that each component is given by a polynomial function.

Elliptic Curves

Definition and Basic Properties

An elliptic curve E over a field K is a smooth, projective algebraic curve of genus one, with a specified point O. In characteristic not 2 or 3, every elliptic curve can be written in short Weierstrass form:

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

where $a, b \in K$, and the discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$.

The Point at Infinity The point O, called the point at infinity, serves as the identity element for the group law. In projective coordinates, it can be represented as $[0 : 1 : 0]$.

Affine and Projective Representations

- Affine form: $E = \{(x, y) \in K^2 : y^2 = x^3 + ax + b\} \cup \{O\}$
- Projective form: $E = \{[X : Y : Z] \in \mathbb{P}^2(K) : Y^2Z = X^3 + aXZ^2 + bZ^3\}$

Group Law

Elliptic curves have an abelian group structure, with the point at infinity O serving as the identity element.

Geometric Interpretation For points P, Q on E:

1. $O + P = P$ for all P
2. If $P = (x, y)$, then $P + (x, -y) = O$ (inverse)
3. To add P and Q (chord rule):
    - Draw a line through P and Q
    - Find the third intersection point R with E
    - Reflect R across the x-axis to get $P + Q$
4. To double P (tangent rule):
    - Draw the tangent line to E at P
    - Find the second intersection point R with E
    - Reflect R across the x-axis to get $2P$

Algebraic Formulas For $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, $P_3 = (x_3, y_3) = P_1 + P_2$:

If $P_1 \neq P_2$: $x_3 = \lambda^2 - x_1 - x_2$ $y_3 = \lambda(x_1 - x_3) - y_1$ where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$

If $P_1 = P_2$: $x_3 = \lambda^2 - 2x_1$ $y_3 = \lambda(x_1 - x_3) - y_1$ where $\lambda = \frac{3x_1^2 + a}{2y_1}$

Scalar Multiplication

For a point P on E and an integer n, scalar multiplication $[n]P$ is defined as:

$$[n]P = \underbrace{P + P + \cdots + P}_{n \text{ times}}$$

This operation is fundamental in elliptic curve cryptography.

Double-and-Add Algorithm An efficient method to compute $[n]P$:

1. Convert n to binary: $n = \sum_{i=0}^{k} b_i 2^i$
2. Initialize $Q = O$
3. For i from k down to 0:
    - $Q = 2Q$

- If $b_i = 1$, $Q = Q + P$

4. Return Q

Elliptic Curves over Finite Fields

When the field K is finite (typically $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$), the elliptic curve $E(K)$ forms a finite abelian group.

Order of the Curve The number of points on $E(\mathbb{F}_q)$, denoted $\#E(\mathbb{F}_q)$, satisfies the Hasse bound:

$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q}$

Structure Theorem For an elliptic curve E over $\mathbb{F}_q$:

$E(\mathbb{F}_q) \cong \mathbb{Z}/n_1\mathbb{Z} \oplus \mathbb{Z}/n_2\mathbb{Z}$

where $n_1 | n_2$ and $n_1 | q - 1$.

Pairings

Bilinear pairings on elliptic curves are crucial for many advanced cryptographic protocols.

Weil Pairing For an elliptic curve E over $\mathbb{F}_q$ and a prime $l|\#E(\mathbb{F}_q)$, the Weil pairing is a map:

$e : E[l] \times E[l] \to \mu_l$

where $E[l]$ is the l-torsion subgroup and μ_l is the group of l-th roots of unity in $\bar{\mathbb{F}}_q$.

Properties: 1. Bilinearity: $e([a]P, [b]Q) = e(P, Q)^{ab}$ 2. Non-degeneracy: If $e(P, Q) = 1$ for all $Q \in E[l]$, then $P = O$ 3. Alternating: $e(P, P) = 1$ for all $P \in E[l]$

Tate Pairing The reduced Tate pairing is a more efficient alternative to the Weil pairing:

$t : E(\mathbb{F}_{q^k})[l] \times E(\mathbb{F}_{q^k})/lE(\mathbb{F}_{q^k}) \to \mathbb{F}_{q^k}^*/(\mathbb{F}_{q^k}^*)^l$

where k is the embedding degree.

Barreto-Naehrig (BN) Curves

Definition and Properties

Barreto-Naehrig (BN) curves are a family of pairing-friendly elliptic curves defined over a prime field $\mathbb{F}_p$. They have the following key properties:

1. Prime order: The order of the curve is a prime number n.
2. Embedding degree: $k = 12$
3. CM discriminant: $D = 3$
4. Equation form: $E : y^2 = x^3 + b$, where $b \neq 0$

BN curves are particularly significant because: - They support curves of prime order, which is crucial for certain applications and efficient implementations. - They have an embedding degree of 12, which provides a good balance between security and efficiency for pairing-based cryptography.

Parameterization

BN curves are parameterized by a single integer x. The key parameters of the curve are defined as polynomials in x:

1. Trace of Frobenius: $t(x) = 6x^2 + 1$
2. Prime field order: $p(x) = 36x^4 - 36x^3 + 24x^2 - 6x + 1$
3. Curve order: $n(x) = 36x^4 - 36x^3 + 18x^2 - 6x + 1$

For cryptographic use, x is chosen such that both $p(x)$ and $n(x)$ are prime numbers of the desired bit-length.

Embedding Degree

The embedding degree k of a curve E over $\mathbb{F}_p$ with respect to a subgroup of prime order r is the smallest positive integer k such that $r|(p^k - 1)$.

For BN curves: - The embedding degree is always $k = 12$. - This means that the smallest extension field $\mathbb{F}_{p^k}$ that contains the r-th roots of unity is $\mathbb{F}_{p^{12}}$.

The embedding degree is crucial for pairing-based cryptography because: 1. It determines the field in which pairing computations take place. 2. It affects the security level of the pairing-based system. 3. It influences the efficiency of pairing computations.

For BN curves, $k = 12$ provides a good balance: - It's large enough to provide sufficient security against index calculus attacks on the discrete logarithm problem in the extension field. - It's small enough to allow for efficient implementation of field arithmetic in $\mathbb{F}_{p^{12}}$.

Construction and Usage

To construct a BN curve for cryptographic use:

1. Choose an integer x with low Hamming weight to optimize certain operations.
2. Compute $p(x)$ and $n(x)$. If both are prime, proceed; otherwise, choose a different x.
3. The curve equation is $E : y^2 = x^3 + b$, where b is typically chosen to be a small integer (often 2 or 3).
4. The curve is defined over $\mathbb{F}_p$, where $p = p(x)$.
5. The order of the curve is $n = n(x)$.

BN254 Specifics

The curve BN254 is specified by the following parameters.

Curve generator $z = 2^{62} - 2^{54} + 2^{44}$

Prime

$$p = 36z^4 + 36z^3 + 24z^2 + 6z + 1$$

$$= 21888242871839275222246405745257275088696311157297823662689037894645226208583$$

Curve equation

$$y^2 = x^3 + 3$$

Order

$$r = 36z^4 + 36z^3 + 18z^2 + 6z + 1$$

$$= 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

Embedding degree $k = 12$

Security

In theory, the curve has $L = 128$ bit security, but it was shown to now be closer to ~100. The curve also has a prime base field of 254-bits, which is huge by our conception of what a large number is, but there are curves with bigger primes (and thus higher security in a real way). Also, the curve as we show below has a cofactor decomposition that require subgroup checks to avoid invalid curve or small subgroup attacks. And as always, operations should run constant time to avoid side channel attacks.

Torsions

Let G be a group. The r-torsion of the group is defined by $\{x \in G \,|\, rX = \mathcal{O}\}$, where $\mathcal{O}$ is the identity element (we use the notation $\mathcal{O}$ to be consistent with the notation that an r-torsion on an elliptic curve group is the point at infinity $\mathcal{O}$). A great example is an analog clock, where the 12-torsion of the clock group is every hour on the clock, since adding an hour 12 times to any hour brings you to the same time on the clock.

$\mathbb{G}_1$

In order to use this curve for cryptography, we need two different versions of the curve, which are then manipulated / used by the pairings discussed earlier. The first version of the curve is the naive expectation you have, namely the pairs of points $\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \,|\, y^2 = x^3 + 3\}$. This is almost right what we need! We add one more condition though that the group we want to deal with is actually the smallest prime cyclic subgroup of order r. Therefore, the group we want to deal with is actually the r-torsion of this elliptic curve group:

$\mathbb{G}_1 \triangleq E(\mathbb{F}_p)[r]$ is the only subgroup of the r-torsion of E on $\mathbb{F}_p$ of order r

Membership check for $\mathbb{G}_1$ In our scheme, to hash a message to E, we use `hash_to_field` and `field_to_curve`, and then multiply the mapped curve point by the generator of the curve to create a point in $\mathbb{G}_1$. Fortunately, by Theorem 2.3.1 of Silverman, we have

$$|E(\mathbb{F}_p)| = p + 1 - t$$

and for BN curves generated by a value $z = 2^{62} - 2^{54} + 2^{44}$, we have $p(z) + 1 - t(z) = r(z)$, implying that $|E(\mathbb{F}_p)| = r \implies \mathbb{G}_1 = E(\mathbb{F}_p)[r] = E(\mathbb{F}_p)$! In this way, since r-torsions give us some notion of structure, this means that the "prime factorization" of the curve is simply the curve itself, so its smallest possible prime order subgroup is just the group, no extra structure to be found.

We therefore only need to check if a pair $(x, y) \in \mathbb{F}_r \times \mathbb{F}_r$ is on the curve $E(\mathbb{F}_p)$ for membership in $\mathbb{G}_1$.

$\mathbb{G}_2$

We now need a second version of the curve $\mathbb{G}_2$ such that $|\mathbb{G}_1| = |\mathbb{G}_2| = r$, which requires us to now go to $\mathbb{F}_{p^{12}}$. Why?

Well, we're technically dealing with not the entire EC group, but the cyclic subgroup $\langle X \rangle$ generated by the point X which is the base of our DL problem (ie $X^d \equiv Y \implies [d]X = Y$). The embedding process (aka taking points from $\mathbb{G}_1$ and map them into $\mathbb{F}_{p^m}$) is done by the pairing $e(x, y)$ which, for a point x in the n-order subgroup of the curve, will be an n-th root of unity for some y, required by the condition that $e(ax, by) = e(x, y)^{ab}$. In order for this to hold, there must be enough roots of unity in the field, which happens when $p^k \equiv 1 \mod \ell$, where ℓ is the order of the cyclic subgroup. For us, this is $k = 12$, so the embedding must go from the curve to $\mathbb{F}_{p^{12}}$.

We need to deal with this massive extension for the pairing operation because the curve defined over this extension is the smallest extension which contains subgroups of order r that we can use for pairings, one subgroup in which contains only points with zero trace, which we choose to be $\mathbb{G}_2$.

So we have $\mathbb{G}_1 \subset E(\mathbb{F}_p)$ with $|\mathbb{G}_1| = r$, and $\mathbb{G}_2 \subset E(\mathbb{F}_{p^{12}})$ with $|\mathbb{G}_2| = r$ which we want to use for our pairing.

Recall that given an irreducible polynomial $N \in \mathbb{F}_p[x]$ of degree $m = 12$, the elements of this extension are those given by $\{a_{m-1}x^{m-1} + \cdots + a_1 x + a_0 \,|\, a_i \in \mathbb{F}_p\}$

Multiplication is defined by multiplying the two polynomials, then using polynomial long division on the polynomial N to get the remainder, and inverses are defined via the extended euclidean algorithm.

You can "tower" extensions if the order of one divides the order of the other, so if $m_j | m_{j+1}$, then $\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}}$.

The standard tower for BN254 is given by the following (see here or here):

$$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 - \beta) \tag{1}$$
$$\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi) \tag{2}$$
$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v) \tag{3}$$

where β is a quadratic nonresidue in $\mathbb{F}_p$ and ξ neither a quadratic or cubic residue in $\mathbb{F}_{p^2}$, which amounts to saying that $X^6 - \xi$ is irreducible in the ring $\mathbb{F}_{p^2}[X]$. Here, $\beta = -1, \xi = 9 + u$, which brings about $u^2 = -1, w^2 = v, v^3 = 9 + u$, and therefore:

$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^2}[w]/(w^6 - (9 + u))$$

This brings about the following nice points - any element in this extension can be written as $g + hw$ with $g, h \in \mathbb{F}_{p^6}$, which means that the p^6-th power of any element in the extension $x^{p^6} = g - hw$ is free to compute - Likewise writing each g, h in terms of coefficients from $\mathbb{F}_{p^2}$ lets you compute the p-th, p^2-th, and p^3-th powers easily as well

Dealing with elements directly in $\mathbb{F}_{p^{12}}$ is very unruly and inefficient, but it is possible to define a coordinate transformation such that such that the curve in the 12-th order extension is mapped to a lower degree field.

For BN254, we define a sextic twist (aka drops the degree of extension by 6) such that the twisted curve is defined on $\mathbb{F}_{p^2}$ instead of $\mathbb{F}_{p^{12}}$. Defining $u^6 = (1+i)^{-1}$, the twist performs $(x, y) \to (x/u^2, y/u^3)$ to produce our new curve $E'(\mathbb{F}_{p^2})$:

$$y'^2 = x'^3 + \frac{3}{9+i}$$

Very nice. Note though that points in $E(\mathbb{F}_p)$ are pairs of ints, while points on the twist are pairs of complex ints, so points in $\mathbb{G}_2$ take more storage despite them being also valid as the domain for keys and signatures.

See this for industry definition of this twist.

Since $X^6 - \xi$ is irreducible, with roots $w \in \mathbb{F}_{p^{12}}$, we therefore have a homomorphism

$$\Psi : E'(\mathbb{F}_{p^2}) \to E(\mathbb{F}_{p^{12}})\,;\, (x', y') = (w^2 x', w^3 y')$$

which is injective, but not surjective, and defines the twist mapping!

Recalling that the r-torsion points of a curve are all the points X such that $rX = \mathcal{O}$, with $\mathcal{O}$ the point at infinity, ie these are all points of order dividing r, we finally define

- $\mathbb{G}_2 \triangleq E'(\mathbb{F}_{p^2})[r]$ is the only subgroup of the r-torsion of E' on $\mathbb{F}_{p^2}$ of order r

membership check in $\mathbb{G}_2$

First, recall that there is a mapping $\phi_p : E(\overline{\mathbb{F}}_p) \to E(\overline{\mathbb{F}}_p); (x, y) \to (x^p, y^p)$ called the Frobenius morphism. It can can be shown that the set of points fixed by ϕ are *exactly* the finite group $E(\mathbb{F}_p)$, so application of this mapping to the curve defined on the base field will leave things structurally unchanged. This mapping will crop back up later in our definition of optimal ate pairing.

Now, actually checking membership is a bit trickier. You can check easily if the point $(x, y) \in \mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$ lies on $E'(\mathbb{F}_{p^2})$, but unfortunately the order of the twist curve is not given by the order of the r-torsion, ie $|E'(\mathbb{F}_{p^2})| = c_2 r$, where c_2 is the $\mathbb{G}_2$ cofactor. You can show that $c_2 = p + t - 1$. Thinking about r-torsions as structure again, it makes sense that this is the case even just from the consideration of the total number of elements in the preimage of $\mathbb{G}_2$ ($\mathbb{F}_{p^2} \times \mathbb{F}_{p^2}$) vs $\mathbb{G}_1$ ($\mathbb{F}_p$); with that many more elements to consider, it makes sense that there is additional structure in the group to now deal with.

You can just rely on the definition of the r-torsion if you want to check if $[r](x, y) = \mathcal{O}$, but with 254 bits of r, this is slooooooow.

Faster algorithms exist. For instance, defined the untwist-Frobenius-twist endomorphism of Galbraith-Scott:

$$\psi : E'(\mathbb{F}_{p^2}) \to E'(\mathbb{F}_{p^2}) = \Psi^{-1} \circ \phi_p \circ \Psi\,;\, (x', y') \to (\xi^{(p-1)/3} x'^p, \xi^{(p-1)/2} y'^p)$$

where Ψ is the twist mapping, and recall $\xi = 9 + u$. Membership in $\mathbb{G}_2$ therefore boils down to verifying if the following holds: $Q = (x', y'); \psi(Q) = [6x^2]Q$, and more recent work improves this to the following:

$$[x+1]Q + \psi([x]Q) + \psi([x]Q) = \psi^3([2x]Q)$$

You can go even further too.

BN254 with Thresholding

We want to be able to perform signature verification on a hashed message. However, if we only have a single entity producing a signature, we require trust that they are honest, which is a hard requirement since they are indivudually responsible for the integrity of the signature. To circumvent this, we produce a *threshold* signature scheme, specifically what is known as a (t, n)-thresholding scheme. This means that out of a council of n participants, any quorum of t valid partial signatures guarantees validity of the final signature. This allows for a decentralized signaturing, fault tolerance, and validation and verification of and by participants.

Signature schemes

There are a few signatures schemes. I will mention by name, but not go into the details. There are many schemes that produce many valid signatures (DSA, ECDSA, Schnorr), but we want a single valid signature. We therefore start with the Boneh-Lynn-Shacham (BLS) signature scheme, which is pretty ubiquitous.

Key generation First choose $x \sim U(0, p) \in \mathbb{F}_p$ to be the random key, the holder of which generates a public key g^x with g a generator of $\mathbb{F}_p$.

Signing Given a message m, hash it to the target group to produce $H(m)$, and return a signature on the hash $\sigma = xH(m)$

Verificiation Assert that $e(\sigma, g) = e(H(m), g^x)$, where e is a pairing function.

We discuss all of these in detail, as well as the extension to distributed usage among n participants.

Contents

tl;dr

There are a lot of good partial implementations of everything in this document. my recommendation is to start with seda's barebones of bn254 and the pairing library here to create our skeleton. These two libraries have minimal external dependencies, and are lightwight renditions of the functionality. We then take the thresholding logic of threshold_bls for the partial signature generation and aggregation, etc. (all of its curve logic is imported from external crates so I don't recommend starting off with this).

The biggest issue in these existing repos is the security concerns regarding the `hash_to_field` and `field_to_curve` functions, which are only implemented with naïve algorithms in these repos. Fortunately, there is a clear guide to developing secure elliptic curve suites created by cloudflare called RFC 9380 which specifies very clearly, with example algorithms, references, and precise language, how to remedy these issues, and what algorithms to use for which curves, security levels, etc.

There are implementations by arkworks and zkcrypto/bls12_381. Arkworks unfortunately is extermely bloated and very massive for something that only provides the elliptic curve logic, and zkcrpto/bls12_381 is the wrong curve. However, arkworks is a good reference for our friendly bn254, and zkcrypto/bls12_381 conforms to security standards set out in RFC9380, so they should be good references while we build our product.

Step 0: Roadmap

For the BN254 curve, there are two groups we will deal with often.

$\mathbb{G}_1 \subset \mathrm{E}(\mathbb{F}_r)$ **with** E **the curve**

- This is the group of points on the base curve in short Weierstrass form $y^2 = x^3 + 3$ defined over the field $\mathbb{F}_r$

$\mathbb{G}_2 \subset \mathrm{E}'(\mathbb{F}_{r^2})$ **with** E' **the sextic twist of the curve**

- This is the group of points on the twisted curve defined over the quadratic extension field $\mathbb{F}_{r^2}$, defined by $y^2 = x^3 + \frac{3}{i+9}$

---

1. Generate scalars $\{a_0, \ldots, a_{t-1}\}$ in the field $\mathbb{F}_r$
    a. These define private key polynomial coefficients
2. Generate partial key shares by evaluating the polynomial for n shares, making sure to never evalaute at 0
    a. This creates partial priate keys $s_i \in \mathbb{F}_r$
3. Commit the private polynomial to $\mathbb{G}_2$ to create the public key polynomial
    a. Define $A : \mathbb{F}_r \to \mathbb{G}_2 : x \to xg_2$, and apply to polynomial, namely $a_i \to A(a_i) = a_i g_2$
    b. The group public key is the evalution of the public key polynomial at $0 \in \mathbb{F}_r$, namely $a_0 g_2$
4. Each node i will now create a partial signature
    a. First, hash message m into $\mathbb{F}_r$
    b. Second, take the hash and map it to the curve, generating $H(m) \in \mathbb{G}_1$
    c. Thirdly, create the partial signature by multiplying by the partial key share $s_i \in \mathbb{F}_r$ by the hash, $\sigma_i = s_i H(m) \in \mathbb{G}_1$
5. Verify the partial signatures against the public polynomial
    a. Now having the hash on the curve $H(m)$, and the partial signature σ_i, we first evaluate the public polynomial $P(x) = a_0 g_2 + a_1 g_2 x + \cdots + a_{t-1} g_2 x^{t-1}$ at each index i
    b. We then use the pairing function to verify $e(\sigma_i, g_2) = e(H(m), P(i))$
6. Aggregate the participants and their partial signatures to recover the public polynomial constant term, aka pub key $a_0 g_2$, via generation of total signature σ
7. Use same methodology as step 5 to verify the final signature $e(\sigma, g_2) = \prod_i e(H(m), P(i))$

Step 1: generate field scalars

Listing 1: Generate $s \in \mathbb{F}_r$

```rust
use rand::Rng;

#[derive(Debug, Clone, Copy)]
pub struct Scalar([u64; 4]);
```

```rust
impl Scalar {
    // The modulus q of BN254 curve
    const MODULUS: [u64; 4] = [
        0x43e1f593f0000001,
        0x2833e84879b97091,
        0xb85045b68181585d,
        0x30644e72e131a029,
    ];

    // R^2 mod q (used for conversion to Montgomery form)
    const R2: [u64; 4] = [
        0x1bb8e645ae216da7,
        0x53fe3ab1e35c59e3,
        0x8c49833d53bb8085,
        0x0216d0b17f4e44a5,
    ];

    // Generate a random Scalar
    pub fn random() -> Self {
        let mut rng = rand::thread_rng();
        let mut limbs = [0u64; 4];

        loop {
            for i in 0..4 {
                limbs[i] = rng.gen();
            }

            // Ensure the generated number is less than the modulus
            if !Self::is_above_modulus(&limbs) {
                break;
            }
        }

        // Convert to Montgomery form
        Self::to_montgomery_form(&limbs)
    }

    // Check if the generated number is above or equal to the modulus
    fn is_above_modulus(limbs: &[u64; 4]) -> bool {
        for i in (0..4).rev() {
            if limbs[i] > Self::MODULUS[i] {
                return true;
            }
            if limbs[i] < Self::MODULUS[i] {
                return false;
            }
        }
        true
    }

    // Convert to Montgomery form
    fn to_montgomery_form(limbs: &[u64; 4]) -> Self {
```

```
        let mut result = [0u64; 4];
        Self::montgomery_multiply(limbs, &Self::R2, &mut result);
        Scalar(result)
    }

    // Montgomery multiplication
    fn montgomery_multiply(a: &[u64; 4], b: &[u64; 4], result: &mut [u64; 4]) {
        let mut t = [0u64; 8];

        // Multiply
        for i in 0..4 {
            let mut carry = 0u64;
            for j in 0..4 {
                let mut product = (a[i] as u128) * (b[j] as u128) + (t[i + j] as u128) + (carry as u128)
                t[i + j] = product as u64;
                carry = (product >> 64) as u64;
            }
            t[i + 4] = carry;
        }

        // Reduce
        let mut carry = 0u64;
        for i in 0..4 {
            //rando num below is INV=(-q^{-1}mod 2^64)mod 2^64
            //its giving fast inv square root vibes
            let k = t[i].wrapping_mul(0xac96341c4ffffffb);
            let mut sum = (t[i] as u128) + (k as u128) * (Self::MODULUS[0] as u128) + (carry as u128);
            carry = (sum >> 64) as u64;
            for j in 1..4 {
                sum = (t[i + j] as u128) + (k as u128) * (Self::MODULUS[j] as u128) + (carry as u128);
                t[i + j - 1] = sum as u64;
                carry = (sum >> 64) as u64;
            }
            t[i + 3] = carry;
            carry = 0;
        }

        result.copy_from_slice(&t[4..8]);

        // Final reduction
        if Self::is_above_modulus(result) {
            let mut borrow = 0i64;
            for i in 0..4 {
                let diff = (result[i] as i128) - (Self::MODULUS[i] as i128) - (borrow as i128);
                result[i] = diff as u64;
                borrow = if diff < 0 { -1 } else { 0 };
            }
        }
    }
}
```

Many implementations exist. Best ones so far I've found that could add rto the barebones scalar above have

montgomery arithmetic added. Consider this and that.

Step 2: generate partial private shares

Listing 2: evaluate private polynomial at each index

```
#[derive(Debug, Clone, Serialze, Deserialize)]
pub struct Eval<A> {
    pub idx: u32;
    pub val: A
}
let (n, t) = (10, 6);
let coeffs: Vec<Scalar> = (0..t).map(|_|Scalar::random()).collect();

//eval polynomial f(i), but never for i=0 since that exposes the secret
let private_shares = (0..n).map(|i| {
    coeffs.iter().rev().fold(Scalar::zero(), |mut sum, coeff| {
        sum.mul(i+1);
        sum.add(coeff);
        Eval<Scalar> {
            idx: i+1,
            value: sum
        }
    }
}).collect::<Vec<_>>();
//put in eval struct or something for clarity / serialization later
```

Great. Now we have evaluated the polynomial $f(x) = a_0 + a_1 x + \cdots + a_{t-1} x^{t-1}$. Now we get to the fun stuff

Step 3: create public polynomial

First, we need to commit the scalar polynomial generated above to the group to get polynomial on the group, aka multiply each coeff by the generator. We call it committing because of the close connection to KZG polynomials in SNARKS (a good blog on it is here).

all of arkworks-rs, zkcrypto/bls12_381, an threshold_bls implement a struct specifically mapping a Scalar of the field to point on $\mathbb{G}_2$

```
let public_polynomial_g2_coeffs = coeffs.iter().map(|c|{
    let mut commit = <cofactor of G2>;
    commit.mul(c);
    commit
}).collect::<Vec<<stuct of points on the field>>>();
```

then BAM, we get the public key for "free" since its just the constant term of the polynomial

```
let pub_key:<stuct of points on the field> = public_polynomial_g2_coeffs[0];
```

We place the public key as an element of G2. Why? - prevents rogue key attacks, since more complex structure makes it harder to generate fake pub keys - subgroup structure is more complex, so harder to cofactor clear - allows for optimizations in the pairing equation

Also, note that in order to get an element of G2 we multiply by the cofactor, see membership checks. The problem is really that this cofactor is huge:

```
//|E'(F_{p^2})| =
//4790951760166228424419880452166787407992523165311008 22
```

```
//43644780225407009368635634920496921254422003348641327 1
//28356694526465084575588080521391696305835073 3
c_2 = 218882428718392752222464057452572750888442579141796129816798716027146439215 49
```

so there are faster ways to generate an element in G2, for example this. ### Step 4: partial signaturing ok, great. c'est parti à la lune . we now need to partial sign messages. this is distributed obvs in our case, but for here it'd be nice to have something like

```
let partials_sigs_g1 = private_shares.iter().map(|s| bn254::partial_sign(s, &msg));
```

but what does this actually entail? this is the good stuff.

Choose an upper bound on the target security level k, a reasonable choice of which is $\lceil \log_2(r)/2 \rceil$

Define a hash_to_field function to take byte strings to field From RFC 9380, >To control bias, hash_to_field instead uses random integers whose length is at least $\lceil \log_2(p) \rceil + k$ bits, where k is the target security level for the suite in bits. Reducing such integers mod p gives bias at most 2^-k for any p; this bias is appropriate when targeting k-bit security. For each such integer, hash_to_field uses expand_message to obtain L uniform bytes, where $L = \lceil (\lceil \log_2(p) \rceil + k)/8 \rceil$. These uniform bytes are then interpreted as an integer via OS2IP. For example, for a 255-bit prime p, and k = 128-bit security, L = ceil((255 + 128) / 8) = 48 bytes.

More on this later.

Define a field_to_curve function to take field element to $\mathbb{G}_1$

First, we need a way to take a message and hash it to an element of the field, so we use . . .

Listing 3: "try and increment" algorithm for hashing onto $\mathbb{Z}_n$ Require: n ∈ Z with |n|_2 = k and s ∈ {0,1}*

procedure Try-and-Increment(n, k, s)

c ← 0

repeat

s' ← s || c_bits()

z ← H(s')_0 · 2^0 + H(s')_1 · 2^1 + . . . + H(s')_k · 2^k

c ← c + 1

until z < n

return z

end procedure

Ensure: z ∈ Z_n

possible impl here

tl;dr `try-and-increment` : $\{0,1\}^* \to \mathbb{Z}_r; m_2 \to m_{\mathbb{Z}_r} \simeq m_{\mathbb{F}_r}$, which is what is given in moon math manual.

This seems easy enough, but would fail security audits. We should implement a more rigorous method for a given level of security, which for us is 128-bit. An example might be `expand_message_xmd` specified again by RFC 9380, an example impl of which could be:

Listing 4: expand_message_xmd for hash_to_field

```rust
use sha2::{Sha256, Digest};
use num_bigint::BigUint;
use num_traits::Num;

const B_IN_BYTES: usize = 32; // 256 bits for SHA-256
const S_IN_BYTES: usize = 64; // Input block size for SHA-256
const L: usize = 48; // ceil((254 + 128) / 8) = 48 bytes

const P: &str = "21888242871839275222246405745257275088696311157297823662689037894645226208583";
fn expand_message_xmd(msg: &[u8], dst: &[u8], len_in_bytes: usize) -> Vec<u8> {
    let ell = (len_in_bytes + B_IN_BYTES - 1) / B_IN_BYTES;

    assert!(ell <= 255, "ell is too large");
    assert!(len_in_bytes <= 65535, "len_in_bytes is too large");
    assert!(dst.len() <= 255, "DST is too long");

    let dst_prime: Vec<u8> = [dst, &(dst.len() as u8).to_be_bytes()].concat();
    let z_pad = vec![0u8; S_IN_BYTES];
    let l_i_b_str = (len_in_bytes as u16).to_be_bytes();

    let msg_prime: Vec<u8> = [
        &z_pad[..],
        msg,
        &l_i_b_str,
        &[0u8],
        &dst_prime[..]
    ].concat();

    let mut b_0 = Sha256::digest(&msg_prime);
    let mut b_1 = Sha256::digest(&[&b_0[..], &[1u8], &dst_prime[..]].concat());

    let mut uniform_bytes = b_1.to_vec();

    for i in 2..=ell {
        let b_i = Sha256::digest(
            &[
                &xor(&b_0, &b_1)[..],
                &[i as u8],
                &dst_prime[..]
            ].concat()
        );
        uniform_bytes.extend_from_slice(&b_i);
        b_1 = b_i;
    }

    uniform_bytes.truncate(len_in_bytes);
    uniform_bytes
}

fn xor(a: &[u8], b: &[u8]) -> Vec<u8> {
```

```
    a.iter().zip(b.iter()).map(|(&x, &y)| x ^ y).collect()
}

fn i2osp(x: usize, len: usize) -> Vec<u8> {
    x.to_be_bytes()[std::mem::size_of::<usize>() - len..].to_vec()
}

fn hash_to_field(msg: &[u8], dst: &[u8]) -> BigUint {
    let uniform_bytes = expand_message_xmd(msg, dst, L);
    let mut integer = BigUint::from_bytes_be(&uniform_bytes);
    let p = BigUint::from_str_radix(P, 10).unwrap();
    integer %= &p;
    integer
}
```

Now having the message in the field, we need to map it to $\mathbb{G}_1$, aka a pair of $(x, y) \in \mathrm{E}(\mathbb{F}_\mathrm{r})$

It seems the nicest would be the Simplified Shallue-van de Woestijne method. I won't waste time on this one unfortunately, because despite there being an existing impl of this, it requires that in its short affine Weierstrass form that $A \neq 0$ and $B \neq 0$, so we instead present the full . . .

Shallue-van de Woestrijne method Needed constants: - A=0, B=3 for bn254 - $Z \in \mathbb{F}_r$ such that - for $y^2 = g(x) = x^3 + Ax + B$, $g(Z) \neq 0$ in the field - $-\frac{3Z^2+4A}{4g(Z)} \neq 0$ in the field - ALSO this quantity must be a square in the field - At least one of $g(Z)$ and $g(-Z/2)$ is square in the field

Listing 5: A sage script to find such a Z

```
# Arguments:
# - F, a field object, e.g., F = GF(2^521 - 1)
# - A and B, the coefficients of the curve y^2 = x^3 + A * x + B
def find_z_svdw(F, A, B, init_ctr=1):
    g = lambda x: F(x)^3 + F(A) * F(x) + F(B)
    h = lambda Z: -(F(3) * Z^2 + F(4) * A) / (F(4) * g(Z))
    # NOTE: if init_ctr=1 fails to find Z, try setting it to F.gen()
    ctr = init_ctr
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            # Criterion 1:
            #   g(Z) != 0 in F.
            if g(Z_cand) == F(0):
                continue
            # Criterion 2:
            #   -(3 * Z^2 + 4 * A) / (4 * g(Z)) != 0 in F.
            if h(Z_cand) == F(0):
                continue
            # Criterion 3:
            #   -(3 * Z^2 + 4 * A) / (4 * g(Z)) is square in F.
            if not is_square(h(Z_cand)):
                continue
            # Criterion 4:
            #   At least one of g(Z) and g(-Z / 2) is square in F.
```

```
            if is_square(g(Z_cand)) or is_square(g(-Z_cand / F(2))):
                return Z_cand
        ctr += 1
```

LOL all this to show that for BN254, $Z = 1 \in \mathbb{F}_r$ ...

Using the notation and utility functions from here, I summarise the SvW algorithm for input $u \in \mathbb{F}_r$.

Note that the constant c3 below MUST be chosen such that sgno(c3) = 0. In other words, if the square-root computation returns a value cx such that sgno(cx) = 1, set c3 = -cx; otherwise, set c3 = cx.

Constants: 1. $c1 = g(Z)$ 2. $c2 = -Z/2$ 3. $c3 = \sqrt{-g(Z) * (3Z^2 + 4A)}$ # sgno(c3) MUST equal 0 4. $c4 = -4g(Z)/(3Z^2 + 4A)$

Listing 6: the SvW algorithm $A : \mathbb{F}_r \to \mathbb{F}_r \times \mathbb{F}_r$ `rust vscode={"languageId": "plaintext"} tv1 = u^2 tv1 = tv1 * c1 tv2 = 1 + tv1 tv1 = 1 - tv1 tv3 = tv1 * tv2 tv3 = inv0(tv3) tv4 = u * tv1 tv4 = tv4 * tv3 tv4 = tv4 * c3 x1 = c2 - tv4 gx1 = x1^2 gx1 = gx1 + A gx1 = gx1 * x1 gx1 = gx1 + B e1 = is_square(gx1) x2 = c2 + tv4 gx2 = x2^2 gx2 = gx2 + A gx2 = gx2 * x2 gx2 = gx2 + B e2 = is_square(gx2) AND NOT e1 # Avoid short-circuit logic ops x3 = tv2^2 x3 = x3 * tv3 x3 = x3^2 x3 = x3 * c4 x3 = x3 + Z x = CMOV(x3, x1, e1) # x = x1 if gx1 is square, else x = x3 x = CMOV(x, x2, e2) # x = x2 if gx2 is square and gx1 is not gx = x^2 gx = gx + A gx = gx * x gx = gx + B y = sqrt(gx) e3 = sgn0(u) == sgn0(y) y = CMOV(-y, y, e3) # Select correct sign of y return (x, y)`

Then poof! We have the following procedure:

1. **Hashing to element of the field:** use listing 4 to convert the bits of the message to an integer of desired size and field via try-and-increment
   a. hash_to_field : $\{0,1\}^* \to \mathbb{F}_r; m_2 \to m_{\mathbb{F}_r}$
2. **Hashing element of the field to the curve:** use listings 5-6 to then map hashed message to the curve!
   a. field_to_curve : $\mathbb{F}_r \to \mathbb{G}_1; m_{\mathbb{F}_r} \to H(m)$
3. **Signing of the hash:** now, take the hash and sign it with the partial private key of this node
   a. $\sigma_i : \mathbb{G}_1 \to \mathbb{G}_1; H(m) \to s_i H(m)$ with s_i the partial private key $\in \mathbb{F}_r$ from step 2

Each participant has now signed the hashed message to the curve.

Step 5: partial verification

This is pretty straightforward up to deciding how to implement the pairing function.... which is ... easy ... right? Wrong. See 'Field extentions' for the clusterfuck that is pairing maths.

```
let public_polynomial_per_share = (0..n).map(|i| {
    public_polynomial_g2_coeffs.iter().rev().fold(Scalar::zero(), |mut sum, coeff| {
        sum.mul(i+1);
        sum.add(coeff);
        Eval<Scalar> {
            idx: i+1,
            value: sum
        }
    }
}).collect::<Vec<_>>(); //these are the values of public poly we'll use for verification
let all_verified = (0..n).map(|i|{
        let lhs = pairing(partials_sigs_g1[i], <generator of g_2>);
        let rhs = pairing(<hash to be saved from previous calculation>, public_polynomial_per_share[i])
        lhs == rhs
    }).sum() == n - 1;
```

Step 6: Aggregation

First, get lagrange coeffs λ_i to recombine the partial signatures

```
fn lagrange_coefficient(i: usize, indices: &[usize]) -> Scalar {
    let x_i = Scalar::from(i as u64);
    indices.iter().filter(|&&j| j != i).fold(Scalar::one(), |acc, &j| {
        let x_j = Scalar::from(j as u64);
        acc * (x_j * (x_j - x_i).inverse().unwrap())
    })
}
```

Then, we can aggregate the signatures to create $\sigma = \sum_i \lambda_i \sigma_i$

```
fn aggregate_signatures(partial_sigs: &[(usize, <point in G1>)]) -> <point in G1> {
    let indices: Vec<usize> = partial_sigs.iter().map(|&(i, _)| i).collect();

    partial_sigs.iter().map(|&(i, sig)| {
        let lambda_i = lagrange_coefficient(i, &indices);
        sig.mul(lambda_i)
    }).sum()
}
```

In reality we don't need all partials (handle edge cases, plus verify each partial individual first for data integrity, etc)

Step 7: Final verify

Use same code as step 5 to verify the final signature σ.

optimal ate pairing

finally, we're here!! fuck.

There are many choices here, and I'm only choosing one (the "best" one), but the motivation for this I won't go into here. There is a great outline of the many types of pairings here, in one of the orignal manuscripts on the subject.

So our goal here is to take a point $X \in \mathbb{G}_1 = E(\mathbb{F}_p)$, and a point $Y \in \mathbb{G}_2 \subset E'(\mathbb{F}_{p^{12}})$, and map them to a point in a target group $\mathbb{G}_T \subset \mathbb{F}_{p^{12}}$, denoted by the map e, and corresponds qualitatively to multiplying a point in $\mathbb{G}_1$ by a point in $\mathbb{G}_2$.

We need bilinearity, therefore requiring:

$$e([a]X, [b]Y) = e(X, [b]Y)^a = e(X, Y)^{ab} = e(X, [a]Y)^b = e([b]X, [a]Y)$$

The "best" way to create this e is the "optimal ate pairing", which has an *excellent* guide for high speed calculations in software.

Before we dig into the pairing itself, we need to know how to define a line passing through two points on the twisted curve, and what the line is evaluated at a point on the curve. Specifcally, $R_1 = (x'_1, y'_1), R_2 = (x'_2, y'_2) \in E'(\mathbb{F}_{p^2})$, and $T = (x, y) \in E(\mathbb{F}_p)$, we have the line ℓ defined as:

$$\ell_{\Psi(R_1),\Psi(R_2)}(T) = \begin{cases} w^2(x'_2 - x'_1)y + w^3(y'_1 - y'_2)x + w^5(x'_1y'_2 - x'_2y'_1) & R_1 \neq R_2 \\ (3x'^3 - 2y'^2)(9 + u) + w^3(2yy') + w^4(-3xx'^2) & R_1 = R_2 \end{cases}$$

Armed with this knowledge, we now can define the optimal ate pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ to be:

$$\begin{aligned} e(X, Y) = \Bigg(& f_{6z+2,Y}(X) && (4) \\ & \times \ell_{[6z+2]\Psi(Y),\phi_p(\Psi(Y))}(X) && (5) \\ & \times \ell_{[6z+2]\Psi(Y)+\phi_p(\Psi(Y)),-\phi_p(\Psi(Y))}(X) \Bigg)^{\frac{p^{12}-1}{r}} && (6) \end{aligned}$$

now THATS a mouthful, say that 5 times fast. Here, z is the parameter of the curve. The pairing is based on rational functions $f_{i,Q} : \mathbb{N} \times \mathbb{G}_2 \to \mathbb{F}_{p^{12}}$ that are evaluated iteratively in what's called Miller's algorithm.

Fantastically, the paper that describes this process was never published, but the algorithm, the imeplementation of which is refered to as "Miller's loops", says that:

$$f_{i+j,Y} = f_{i,Y} f_{j,Y} \ell_{[i]\Psi(Y),[j]\Psi(Y)}$$

TECHNICALLY there is another factor in the denominator of these iterations that describes the evaluation of the point $\Psi(Y)$ on the vertical line passing through X. However, we can ignore this evaluation, for reasons summarized well by this:

To compute a Tate pairing, a quotient is iteratively calculated (Miller's algorithm) and then raised to power of $(p^k - 1)/r$, the Tate exponent. Each factor of the denominator is the equation of a vertical line evaluated at a particular point, i.e. the equation $X - a$ evaluated at some point (x, y), which gives the factor $(x - a)$.

Because of the way we have selected our groups, $x \in \mathbb{F}_{p^d}$, (note that the map Ψ leaves the x-coordinate of its input in the same field), and $a \in \mathbb{F}_p$, hence $(x - a) \in \mathbb{F}_{p^d}$.

Any element $a \in \mathbb{F}_{p^d}$ satisfies a^{p^d-1}. Observe $p^d - 1$ divides $(p^k - 1)/r$, because r cannot divide $p^d - 1$ (otherwise d would be the embedding degree, not k). Thus each factor $(x - a)$ raised to the Tate exponent is 1, so it can be left out of the quotient. Hence, there is no need to compute the denominator at any time in Miller's algorithm.

Slick.

Toy implementation

In decimals, we know $z = 4965661367192848881$, and therefore $6z + 2 = 29793968203157093288$. Optimised implementations represent this bound in $\{-1, 0, 1\}$ basis, not binary, since it has a lower Hamming weight, just fyi, so in that case we get the following.

There will be miller's loop to determine the first term in the optimal ate pairing. Then for the final two terms, we notice that:

$\ell_{[6z+2]\Psi(Y),\phi_p(\Psi(Y))}(X)$ Notice that:

$$\phi_p(\Psi(Y)) = \left((w^2x')^p, (w^3y')^p\right) = \left(w^2\xi^{(p-1)/3}x'^p, w^3\xi^{(p-1)/2}y'^p\right) = \Psi\left(\xi^{(p-1)/3}\bar{x}', \xi^{(p-1)/2}\bar{y}'\right)$$

Since $[n]\Psi(Q) = \Psi([n]Q)$ by the homomorphism, we just evaluate the line now at the point $Q' = (\xi^{(p-1)/3}\bar{x}', \xi^{(p-1)/2}\bar{y}') = (x_1, y_1)$.

$\ell_{[6z+2]\Psi(Y)+\phi_p(\Psi(Y)),-\phi_p(\Psi(Y))}(X)$ You can likewise show that this is easily evaluated at the point $-Q$.

```
fn e(p: &G1Affine, q: &G2Affine) -> Fq12 {
    //membership checks, see sections above
    assert!(p.is_in_g1());
    assert!(q.is_in_g2());

    if p.is_identity().into() || q.is_identity.into() {
        return Fq12::One();
    }

    let mut r = *q;
    let mut f = Fq12::One(); //starting point of iteration

    //begin miller's loop, calculating f_{[6z+2],q}(p)
    for i in (0..BOUND.len() - 1).rev() {
        f = f * f * line(twist(&r), twist(&r), p);
        r = r.double();
        match BOUND[i] {
            1 => {
                f = f * line(untwist(&r), untwist(q), p);
                r.add_assign(q);
            },
            -1 => {
                f = f * line(untwist(&r),untwist(-q), p);
                r.sub_assign(q);
            },
            0 => {},
            _ => panic!("digit not in correct basis")
        }
    }
```

```
    let qp = q.frobenius_map();
    f = f * line(twist(&r), twist(&qp), p);
    r.add_assign(qp);
    let qpp = -qp.frobenius_map();
    f = f* line(twist(&r), twist(&qpp), p);

    final_exponentiation(&f)
}
```

Final exponentiation

Arguably, this is the most computationally expensive step since the bit size of the exponent in the pairing is huge, so the naïve approach would be silly. I mean, there are issues with the $\mathbb{G}_2$ cofactor clearing to create elements in $\mathbb{G}_2$ from the field because of the size of the cofactor, so if multiplication is slow, exponentiation is not guaranteed to be better *a priori*.

The following takes the lead from this and that.

The most efficient calculation of these pairings relies on notions of *cyclotomic subgroups*. oof.

Up until this point, we were precise in our definitions of $\mathbb{G}_1$ and $\mathbb{G}_2$, but have been unclear about what exactly the target group of the pairing should be. We now formally define the target group $\mathbb{G}_T$ to be the group of r-th roots of unity over the multipicative group $\mathbb{F}_{p^k}^* = (\mathbb{F}_{p^k}/\{0\}, *)$, denoted commonly by μ_r. Why the roots o f unity? Great question. Remember that this mapping has to satisfy a few key real-world properties. First, it has to be a trapdoor, namely preimage resistance (assuming DL hardness), and mapping backwards from the roots of unity is a very difficult problem. Second, it allows for an easy metric against which we can compare two mappings. For example, in the case of signature verification $e(\sigma_i, g_2) = e(H(m), P(i))$, it is very natural to want to set the actual value of each side of this equation to "one", therefore implying the image domain to be the roots of unity. Note that this implies right away that $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = r$, which makes the pairing retain the correct group structure while still mapping to a cryptographically secure image, aka a group of "ones".

In the following, let $(\mathbb{F}_{p^k}^*)^r$ is the subgroup of r-th powers, namely all elements in $\mathbb{F}_{p^k}^*$ that are expressed as x^r for some x. We can then define the quotient group $\mathbb{F}_{p^k}^*/(\mathbb{F}_{p^k}^*)^r$ which represent the coset of r powers, namely each element in this quotient group differ by a power of r.

Note that $\forall x \in \mathbb{F}_{p^k}^*$, $\left(x^{\frac{p^k-1}{r}}\right)^r = x^{p^k-1} = 1$, which means elements of the form $x^{\frac{p^k-1}{r}} \in \mathbb{F}_p^*/(\mathbb{F}_p^*)^r$, which is precisely what the pairing function e does. There is a natural isomorphism between the quotient group and the roots of unity, so we can equivalently talk about either. For the purposes of the following, however, we'll keep to the quotient group representation since it admits a few additional insights we can use to our advantage.

Aside: this is not a light topic to cover, even for the level of depth in this document (hard to believe, I know), but is a result from Galois theory and the so called *Kummer theorems*, see this

---

Since the optimal ate pairing is mapping our "multiplication" of an element from $\mathbb{G}_1$ and an element from $\mathbb{G}_2$ to the group of roots of unity by means of exponentiation of an element from the base field extension $\mathbb{F}_{p^{12}}$, it would be useful to represent our exponent $(p^{12}-1)/r$ in a form closer related to the roots of unity to which we're mapping. To do this, we define what's called the cyclotomic polynomial, defined by an order n. This polynomial contains all of the irreducible factors of $x^n - 1$ (which defines the roots of unity), and is therefore the polynomial whose roots are the roots of unity. In this sense, this polynomial captures all of the structure of the roots of unity, and because of its irreducibility, we can use it to build other mappings that deal with the group of roots of unity.

Specifically, we define the k-th cyclotomic polynomial $\Phi(x)$ to be:

$$\frac{p^k - 1}{\Phi_k(p)} = \prod_{j|k, j \neq k} \Phi_j(p)$$

which allows us to break down the exponent of the "final exponentiation" step. Writing the embedding degree $k = ds$, where d is a positive integer, we can write:

$$\frac{p^k - 1}{r} = \underbrace{\left[(p^s - 1) \cdot \frac{\sum_{i=0}^{d-1} p^{is}}{\Phi_k(p)}\right]}_{\text{easy part}} \cdot \underbrace{\left[\frac{\Phi_k(p)}{r}\right]}_{\text{hard part}}$$

Easy part

For BN254, this decomposes the easy part into $(p^6-1)(p^2+1)$ for $k=12$ (note that we choose this decomposition because exponentiation by powers of p are very efficient, see the discussion earlier). The easy part will involve something like $x^{p^6-1} = x^{p^6} \cdot x^{-1}$, which is one conjugation, one inversion, and one multiplication (remember that conjugation in $\mathbb{F}_{p^{12}}$ for an element $x = a + bw$ is simply $\bar{x} = a - bw$). Then taking $\left(x^{p^6-1}\right)^{p^2}$ is just applying our Frobenius morphism ϕ, and then finally we multipy by our already-computed value x^{p^6-1}, and voila!

Easy part = 1 conjugation + 1 inversion + 1 multiplication + 5 multiplications + 1 multiplication

Hard part

For BN254, the hard part decomposes into $\frac{p^4-p^2+1}{r}$. It seems that the typical way to go here is to take a base-p expansion, namely defining $\lambda \triangleq m\varphi_k(p)/r$ with $r \nmid m$, and finding a vector τ of $w+1$ integers $\tau = (\lambda_0, \ldots, \lambda_w)$ such that $\lambda = \sum \lambda_i p^i$ minimizing the L1-norm of τ. Recall that for us:

$$p(z) = 36z^4 + 36z^3 + 24z^2 + 6z + 1$$

$$r(z) = 36z^4 + 36z^3 + 18z^2 + 6z + 1$$

$$t(z) = 6z^2 + 1$$

so substituting these into the hard part of the polynomial as a function of the curve family generator z yields $\lambda_3 p^3 + \lambda_2 p^2 + \lambda_1 p + \lambda_0$ with:

$$\lambda_3(z) = 1$$

$$\lambda_2(z) = 6z^2 + 1$$

$$\lambda_1(z) = -36z^3 - 18z^2 - 12z + 1$$

$$\lambda_0(z) = -36z^3 - 30z^2 - 18z - 2$$

We now then compute the hard part as a series of multiplications in terms of powers of the easy part.

1. Compute $f_{\text{easy}}^z, (f_{\text{easy}}^z)^z, (f_{\text{easy}}^{z^2})^z$
2. Use the Frobenius operator, which has efficient representations in powers 1, 2, and 3 of the prime, to compute $f_{\text{easy}}^p, f_{\text{easy}}^{p^2}, f_{\text{easy}}^{p^3}, (f_{\text{easy}}^z)^p, (f_{\text{easy}}^{z^2})^p, (f_{\text{easy}}^{z^3})^p, (f_{\text{easy}}^{z^3})^{p^2}$

The evaluation then amounts to:

$$\underbrace{[f_{\text{easy}}^p \cdot f_{\text{easy}}^{p^2} \cdot f_{\text{easy}}^{p^3}]}_{\equiv y_0} \cdot \underbrace{[1/f_{\text{easy}}]^2}_{\equiv y_1^2} \cdot \underbrace{[(f_{\text{easy}}^{z^2})^p]^6}_{\equiv y_2^6} \cdot \underbrace{[1/(f_{\text{easy}}^z)^p]^{12}}_{\equiv y_3^{12}} \cdot \underbrace{[1/(f_{\text{easy}}^z \cdot (f_{\text{easy}}^{z^2})^p)]^{18}}_{\equiv y_4^{18}} \cdot \underbrace{[1/f_{\text{easy}}^{z^2}]^{30}}_{\equiv y_5^{30}} \cdot \underbrace{[1/(f_{\text{easy}}^{z^3} \cdot (f_{\text{easy}}^{z^3})^p)]^{36}}_{\equiv y_6^{36}}$$

These evaluations have efficient algorithms that have been around for a long time. We take the vector addition chain approach, which is more or less the equivalent of "flattening" that we also see crop up in the reduction of polynomial constraints in instance-witness definitions of R1CS systems. You can show that the following definitions yield efficient computation of these multiexponentials which we take from the original manuscript:

$$T_0 \leftarrow (y_6)^2$$

$$T_0 \leftarrow T_0 \cdot y_4$$

$$T_0 \leftarrow T_0 \cdot y_5$$

$$T_1 \leftarrow y_3 \cdot y_5$$

$$T_1 \leftarrow T_1 \cdot T_0$$

$$T_0 \leftarrow T_0 \cdot y_2$$
$$T_1 \leftarrow (T_1)^2$$
$$T_1 \leftarrow T_1 \cdot T_0$$
$$T_1 \leftarrow (T_1)^2$$
$$T_0 \leftarrow T_1 \cdot y_1$$
$$T_1 \leftarrow T_1 \cdot y_0$$
$$T_0 \leftarrow (T_0)^2$$
$$f_{\text{hard}} \leftarrow T_0 \cdot T_1$$

which is only a few multiplications and squarings! Efficient. There's extensions and improvements, but that's the basic stuff.

BONUS - glued miller loop and improved signature performance

This is pretty sick. Remember that eventually we want to check the relation $e(\sigma_i, g_2) = e(H(m), P(i))$ for verification. You could just naively evaluate lhs and rhs and check for equality. Right? Or notice that:

$$\begin{aligned} & e(\sigma_i, g_2) = e(H(m), P(i)) \\ \implies & e(\sigma_i, g_2) e(H(m), P(i))^{-1} = 1 \\ \implies & e(\sigma_i, g_2) e(H(m), -P(i)) = 1 \\ \implies & \left(f_{[6z+2],\sigma_i}(g_2) f_{[6z+2],H(m)}(-P(i))\right)^{\frac{p^{12}-1}{r}} = 1 \end{aligned}$$

which results in only having to evaluate a single Miller loop, followed by a single exponentiation at the end! Also during the recursion we don't need to track $f_{i,s}(G)$ nor $f_{i,H(m)}(-xG)$, just their product, which saves a multiplication in $\mathbb{F}_{12}$ in each iteration of this *glued* Miller loop. The savings compound since we're aggregating many partial signatures, and the idea works exactly the same for the aggregated signatures. Namely, verification is equivalent to:

$$\left(\prod_i^t f_{[6z+2],\sigma_i}(g_2) f_{[6z+2],H(m)}(-P(i))\right)^{\frac{p^{12}-1}{r}} = 1$$

This is the jist of it. There's so many more things to work with, like different pairings like the Xate pairing, but that's beyond the scope here. I'll just mention maybe that there are many more cleverer things you can do to take full advantage of the cyclotomic subgroup stuff like efficient compression and arithmetic on compressed representations of elements, but that's over the top for now.

Distributed Keys

When dealing with cryptography in general, the source of truth you use for encrypting your secrets is usually a single point of failure, in any naïve implementations. This could be the private key used to generate / verify signatures for example. Also, when decentralizing cryptography protocols, traditional ideas of verification must be extended to encompass many parties, with the intention of removing single point failures, not adding more. This goes over the basics of related distribution protocols for secret sharing and validation.

Shamir's Secret Sharing

This is the foundation on which many distributed key generation (DKG) protocols are based. The punchline is that it allows for t individuals, out of n total, to verify a signature. Not only does this allow for fault tolerance (e.g. node in the network goes down), but allows for a quorum of attestations, improving trustlessness. Maximum security would require $n > \lfloor 2t - 1 \rfloor$.

At the core of this concept is the following truth. Given a set $\{(x_i, y_i) \in \mathbb{R}_2 \,|\, i \in [0, t]\}$, there exists a UNIQUE polynomial $q(x)$ of degree $t - 1$ such that $q(x_i) = y_i$. We then express this polynomial as:

$$q(x) = a_0 + a_1 x + \cdots + a_{t-1} x^{t-1}$$

where a_0 is the secret to be shared. Discretization is then done as $s_i = q(i)$, where s_i is the partial shared secret. The partial signature is therefore $s_i H(m)$, where $H(m)$ is the hashed message in the group $\mathbb{G}_1$. The uniqueness of polynomial interoplation requires knowledge of exactly t of these partial shares to recover the shared secret. Knowledge of only $t - 1$ means that, of the candidate $a_0' \in [0, p)$, there will be again only a unique polynomial fitting $(0, a_0')$ and (i, a_i), each of these p unique polynomials being equally likely, thus maintaining security of the shared secret.

Recovery of the shared secret is done via Lagrange interpolation:

$$a_0 = q(0) = \sum_{j=0}^{t-1} y_t \prod_{m=0, m\neq j}^{t-1} \frac{x_m}{x_m - x_j}$$

The problems with this, while admittedly simple and extensible, is that it assumes honesty of the participants (at no point is there an ability to verify partial shares), and further that the shared secret is known singularly at some point in space-time, both at instantiation of the sharing and at the recovery of the shares. To address these issues, we move beyond to . . .

Feldman's VSS

This is a new approach to secret sharing that addresses the concern that the partials are not verifiable by the council. We again now generate $q(x)$, with $q(0) = a_0$ which is the secret, and transmit to all i participants a partial share $s_i = f(i)$. The dealer also sends a committment $\{A_k = g^{a_k}\}_{k=0,t}$, with g a generator of $\mathbb{Z}_p^*$. This allows for verification (that the partials form a secret) by checking:

$$g^{s_i} \stackrel{?}{=} \prod_{k=0}^{t} (A_k)^{i^k}$$

performed in the field (aka mod p). If participant i does not satisfy the above, a complaint is made publically against the dealer, who is required to then reveal s_i that satisfy the above, else they're fired!

The biggest issue is that this protocol leaks $A_0 = g^{a_0} = g^{\text{secret}} \mod p$. However, assume the bad actor knows t or less shares s_i and g_{a_0}. They can then compute $\{A_k\}_{k=1,t}$ via $A_k = g^{a_k} = \prod_{i=0}^{t} (g^{s_i})^{\lambda_{ki}}$, where λ satisfies $a_k = \sum_{i=0}^{t} \lambda_{ki} f(i)$, which is still not enough to learn about a_0 anymore than what you could learn from g^{a_0}.

pedersen vss

To address the concern of the secret's secrecy, we move to pedersen dkg. Now in this scheme, each participant generates coefficients, and the cooperation of participants is what allows for the collective generation of a secret. The public parameters in this setup are a large prime p, a generator g of a subgroup of $\mathbb{Z}_p^*$, and another element in the subgroup of $\mathbb{Z}_p$ generated by g, h, where no one knows $\log_g h$. Therefore, the cost of information-theoretic secrecy of the shared secret is the hard assumption that the bad actor, or a cheating dealer, cannot solve the DL problem.

The dealer generates two random polynomials of degree t, $q(x) = \sum_k a_k x^k, \tilde{q}(x) = \sum_k b_k x^k \in \mathbb{Z}_p[x]$, with again $q(0) = a_0$ the shared secret. The dealer then transmits to each participant i their share $(s_i = q(i), \tilde{s}_i = \tilde{q}(i))$, and broadcasts the values $\{C_k = g^{a_k} g^{b_k} \mod p\}_{k=0,t}$. Verification of the shared secret follows by:

$$g^{s_i} h^{\tilde{s}_i} \stackrel{?}{=} \prod_{k=0}^{t} (C_k)^{i^k} \mod p$$

As before, complaints against the dealer are made public, and are rectified by the dealer transmitting $(s_i, \tilde{s}_i)$ satisfying the above lest they face disqualification.

DKG

feldman dkg

feldman dkg is built on top of the idea's above to create distribution of the process among participants. Each participant i is now a dealer running a version of Feldman vss, and glued together according to the following. .

Firstly, each participant i chooses a secret value $a_i \sim U(0,p) \in \mathbb{Z}_p$, as well as a random polynomial $q_i(x)$ of degree t in $\mathbb{Z}_p[x]$:

$$q_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + \cdots + a_{it}x^t$$

The participant then computes and broadcasts $C_{ij} = g^{a_{ij}}$ for $j \in [0,t]$

Then, everyone computes the shares of $1,\ldots,n$ of their secret polynomial $s_{ij} = q_i(j)$, and sends them secretly to participants $j = 1,\ldots,n$.

A participant is able to verify that a share received from j is consistent by assuring:

$$g^{s_{ij}} = \prod_{k=0}^{t} C_{ik}^{j^k}$$

If this fails, then participant i complains publically. If there are t complaints against j, they're automatically kicked out. If there are less than t complaints, the accused participant must broadcast the correct share (and random value if used). If this fails, they're booted (you're fired!). In this way, we can remove the assumption in Shamir's secret sharing of honesty of the participants, because we can verify when they're lying.

Finally, participant i determines their share as $s_i = \sum_{j \in \text{qualified}} s_{ji}$, and the public key is now $y = \prod_{i \in \text{qualified}} C_{i0}$, and the public verification values are $C_k = \prod_{i \in \text{qualified}} C_{ik}$. The shared secret value s itself is not computed by anyone since they don't know all the parts, but can be seen as $s = \sum_i a_{i0}$.

pedersen dkg

We now finally move onto secure distributed key generation that is verified to be secure enough for threshold signaturing (see this).

We now buil don topof the pedersen vss and feldman vss.

Generating s: Each player P_i performs a Pedersen-VSS of a random value z_i as a dealer: 1. P_i chooses two random polynomials $q_i(z), q_i'(z)$ over $\mathbb{Z}_q[z]$ of degree t:

$$q_i(z) = a_{i0} + a_{i1}z + \ldots + a_{it}z^t$$

$$q_i'(z) = b_{i0} + b_{i1}z + \ldots + b_{it}z^t$$

Let $z_i = a_{i0} = q_i(0)$. P_i broadcasts $C_{ik} = g^{a_{ik}}h^{b_{ik}} \mod p$ for $k = 0,\ldots,t$. P_i computes the shares $s_{ij} = q_i(j), s_{ij}' = q_i'(j) \mod q$ for $j = 1,\ldots,n$ and sends s_{ij}, s_{ij}' to player P_j.

3. Each player P_j verifies the shares he received from the other players. For each $i = 1,\ldots,n$, P_j checks if

$$g^{s_{ij}}h^{s_{ij}'} = \prod_{k=0}^{t}(C_{ik})^{j^k} \mod p$$

   If the check fails for an index i, P_j broadcasts a complaint against P_i.

4. Each player P_i who, as a dealer, received a complaint from player P_j broadcasts the values s_{ij}, s_{ij}' that satisfy the above.

5. Each player marks as disqualified any player that either

- received more than t complaints in Step 1b, or
- answered to a complaint in Step 1c with values that falsify the relation above.

Each player then builds the set of non-disqualified players QUAL. The distributed secret value s is not explicitly computed by any party, but it equals $s = \sum_{i \in QUAL} z_i \mod q$. Each player P_i sets his share of the secret as $s_i = \sum_{j \in QUAL} s_{ji} \mod q$ and the value $s'_i = \sum_{j \in QUAL} s'_{ji} \mod q$.

Extracting $y = g^s \mod p$: Each player $i \in QUAL$ exposes $y_i = g^{z_i} \mod p$ via Feldman VSS:

1. Each player P_i, $i \in QUAL$, broadcasts $A_{ik} = g^{a_{ik}} \mod p$ for $k = 0, \ldots, t$.
2. Each player P_j verifies the values broadcast by the other players in QUAL, namely, for each $i \in QUAL$, P_j checks if
$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} \mod p$$
If the check fails for an index i, P_j complains against P_i by broadcasting the values s_{ij}, s'_{ij} that satisfy the relation in part 1 but do not satisfy the relation immediately above.
3. For players P_i who receive at least one valid complaint, i.e. values which satisfy the equation in part 1 and not the equation above, the other players run the reconstruction phase of Pedersen-VSS to compute z_i, $q_i(z)$, A_{ik} for $k = 0, \ldots, t$ in the clear. For all players in QUAL, set $y_i = A_{i0} = g^{z_i} \mod p$. Compute $y = \prod_{i \in QUAL} y_i \mod p$.

What might this look like? I'd recommend looking at the implementation here for the basics. The idea is that the reduction of single point failures via the secret generation requires a more secure protocol like pedersen dkg, which allows later steps to be less secure and more efficient, namely by using feldman instead.

However, it was shown that using Feldman vss for both stages is actually secure enough when subsequently used in a thresholding signature scheme! This is why cloudflare uses it, and saves them the pain of implementing two different VSS schemes.

publically verifiable secret sharing and DKG

The schemes above rely on internal validation of the council for maintaining honesty and rigor, but if the entire idea is to create a scheme that removes trust, this is a natural point to consider for further improvements. We move to define a publically verifiable secret sharing protocol:

1. Setup

- The initial parameters contain information about base field, (t, n) and the relation defining valid key pairs (pairing relation)
- Generate public private key pair with SNARK proof asserting the validity of the pair
- Execution of proof - verify validity

2. Distribution

- This takes a secret, and outputs encrypted partial shares and a SNARK proof asserting sharing correctness

3. Distribution verification

- Execution of previous proof - verify distribution of shares

4. Reconstruction

- outputs a decrypted share and a proof of decryption
- reconstruct the secert or return an error

5. Verify the reconstruction (namely if decryted share is valid decryption of the partial share)

You'll notice that there are many SNARKs involved in this process, which adds must complications and headaches. For this reason, it's difficult to implement this protocol, so if you don't need this level of scrutiny, then it's probably better to not do this. I'd read this for a nice overview.

VRF

The problem with a pseudorandom oracle is that it is, by its construction, NOT verifiable. Namely, without knowledge of the seed s, you cannot distinguish the return value of a pseudorandom function f_s from an independently selected random string. This therefore requires trust on behalf of the oracle to produce and faithfully execute the evaluation of the oracle function $f_s(x)$. You could remove trust by publishing the seed, but then you remove all unpredictability.

You can instead use a proof to validate the faithful execution of $f_s(x)$ without actually revealing s. A proof proof_x would say that a unique value v is provable as the value of $f_s(x)$. This is what is called a verifiable random function (VRF).

Issues with proofs

If we allow interaction, then we can use a ZK proof and a committment scheme of the oracle to the seed s. To prove $v = f_s(x)$, the oracle gives a ZK proof to the verifier of the above, and that c is a committement to s. But we don't want interaction ideally.

We could move to a noninteracting ZK proof, but this requires consensus on a bit string between prover and verifier that is GUARANTEED to be random. We want to avoid all the ways we coudl generate this random string:

- If the prover chooses the string, this breaks guarantee of the proof and introduces bias
- If the verified chooses the string, the ZK assumption is broken, and the proof system is not guaranteed (namely, the prover could leak info on s and break unpredictability by proof)
- If both jointly choose the string, that's interacting, which we don't want
- If a third party chooses the string, we add a further requirement of trust.

See here for more details.

Signatures as VRFs

Assume that a scheme exists to generate signatures $\sigma = sH(m)$, with a mapped-to-group hash of a message m, and private key s. This is unpredictable (by assumption of DL hardness), but verifiable (with knowledge of the pub key and pairing operations on the curve for instance). However: - There may be many valid signatures for a given string, violating the requirement that elements in the image of the VRF are provably unique - The signature is unpredictable, not necessary random

The unique provability depends on the actual scheme you use, and you'd hope that your scheme satisfies this, but if your scheme is probabilistic or hysteretic, people have shown that you cannot guaratee unique provability.

These functions are called verifiable unpredictable functions (VUFs).

Is it possible to turn a VUF into a VRF? Yes, and no.

See here for the construction of what this could look like,but this only works IFF f satisfies the property that input lengths are logarithmically related to the security, so in that construction, it suggests that for a BLS signature scheme on BN254 that $|m| = \mathcal{O}(\log L)$, which limits the size of the messages we can securely sign in this approach. So, next!

There are other ways to turn a VUF into VRF, for example hardcore bit extraction. We can use the following construction to replace the oracle function with $f \hookrightarrow f' = \langle f(x), r\rangle_2$, where r is a random binary string and the brackets are inner products. The proof is therefore saying $f'(x) = b$ is a string v such that $\langle r, v\rangle_2 = b$ with a proof of $f(x) = v$. This might look like:

```
fn hardcore_bit_extraction(signature: &noether::U256)-> Result<noether::U256>{
    let mut rng = rand::thread_rng();
    let rando = noether::U256::from_limbs(rng.gen());
    let mut retval = noether::U256::ZERO;
```

```
    for i in 0..256 {
        if (signature.bit(i) ^ rando.bit(i)) == 1 {
            retval = retval.bit_or(noether::U256::ONE.shl(i));
        }
    }
    //add final hash
    let hash = ethers::core::utils::keccak256(retval.to_be_bytes::<32>());
    Ok(noether::U256::from_be_slice(&hash))
}
```

Implementation considerations

Scalar definition

We need multiprecision arithmetic because of the sheer size of these numbers. This section outlines performance considerations when dealing an implementation of these arithemtics.

For a 256-bit prime modulus, the decomposition of this into 4 64-bit limbs/words/branches/fields optimizes the arithmetic beacuse of the fact that each limb fits natively into 64-bit cpu registers. The general idea for squaring / multiplying 4-limb elements is that the operation produces an 8-limb element, which could be considered as a `long` type, with any potential overflow of the 4 or 8 limb element being treated by a carry bit. There are, apparently constant time functions that exist for comparisons and modulo addition and subtraction too in this representation. Since these operations happen extremely frequently, they need to be optimized as much as possible.

Coordinates representations I cannot really get into this without talking about coordinate representations of points on the elliptic curves. There are different representations that admit different algorithms, and that is a bummmer. The biggest issue we'll come across are addition and doubling of points on curves, and these will be better or worse depending on which coordinate system we use. That being said, here is an overview of the different options that we will consider:

- Affine
  - this is the most basic and intuitive, it is simply the pair of coordinates (x, y) such that the pair represents a point that lies on the curve, aka $y^2 = x^3 + 3$.
  - Not favourable for doubling or addition because it involves inversion, which for a prime field with order as large as ours, is the most extremely ineffecient of the arithmetic operations on elliptic curves.
- projective
  - this coordinate system solves the problem of inversion by the introduction of a third element that replaces inversion with a few other operations that are cheaper
    - they are therefore defined by a tuple $[X : Y : Z]$
  - the conversion from affine to projective is simply $\texttt{proj} : \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{FP} \times \mathbb{FP}; (x, y) \to [X = x : Y = y : Z = 1]$
    - points of the form $[X : Y : 0]$ are the point(s) of infinity
- Jacobian
  - this is a special subset of projective coordinates where $(x, y) = (X/Z^2, Y/Z^3)$
  - this seems to admit very efficient operations on coordinates, specifically for addition and doubling
- extended twisted coordinates
  - this introduces a fourth variable that now unifies the addition and doubling formulae, are De Smet et al shows that this can be parallelized

We'll deal with Jacobian coordinates in the entirety of what follows, since that's where the good stuff seems to be.

Modular multiplication

Can't really be parallelized as far as I found. Fastest algorithm that exists is the Montgomery reduction algorithm, which is outlined as Algorithm 14.32 in the handbook, and is optimized for word-by-word reduction! very sweet. The algorithm is faster at the expense of a precomputation step of the parameter $\texttt{inv} = (-q^{-1} mod 2^64) mod 2^64$, but this is easy to do since the parameter is constant for every element in the field. The algorithm is given below, and an example toy implementation is given in my `Scalar` class in `bls.ipynb`, which is untested and unoptimized. An excellent implementation already exists in alloy, so let's use it! Note that this is for the multiplication of 2 256-bit (in our case) numbers, which is independent of the coordinate representation. This is just a fast modular multiplication algorithm so it's general enough to include in our implementation regardless of the choice of affine vs jacobian.

Listing 1: Montgomery multiplication **INPUT**: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.

OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
    i. $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
    ii. $A \leftarrow (A + x_i y + u_i m)/b$.
3. If $A \geq m$ then $A \leftarrow A - m$.
4. Return(A).

Modular addition and doubling

This is slow in the affine representation because of the inversion, done with either extended euclidean algorithm or modular exponentiation.

$$\text{Addition:} \quad (x_3, y_3) = \left(\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2, \left(\frac{y_2 - y_1}{x_2 - x_1} \right) (x_1 - x_3) - y_1 \right)$$

$$\text{Doubling:} \quad (x_3, y_3) = \left(\left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1, \left(\frac{3x_1^2 + a}{2y_1} \right) (x_1 - x_3) - y_1 \right)$$

This can be improved and even parallelized if we move to extended twisted edwards coordinates because it removes the branching due to the unity of the addition and doubling formulae, which is why this is called the snark-friendly representation:

$$(x_3, y_3) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right)$$

But this is a bit much. I recommend all operations on elements on curves be done in projective coordinates, since there are many optimized algorithms for this outlined here.

There's also a good reference for the modular arithmetic within different fields here.

Advanced Topics for BLS and DKG

Fast BLS Threshold Signature Aggregation

Scalable DKG Protocols

Optimizations for Large-Scale Systems