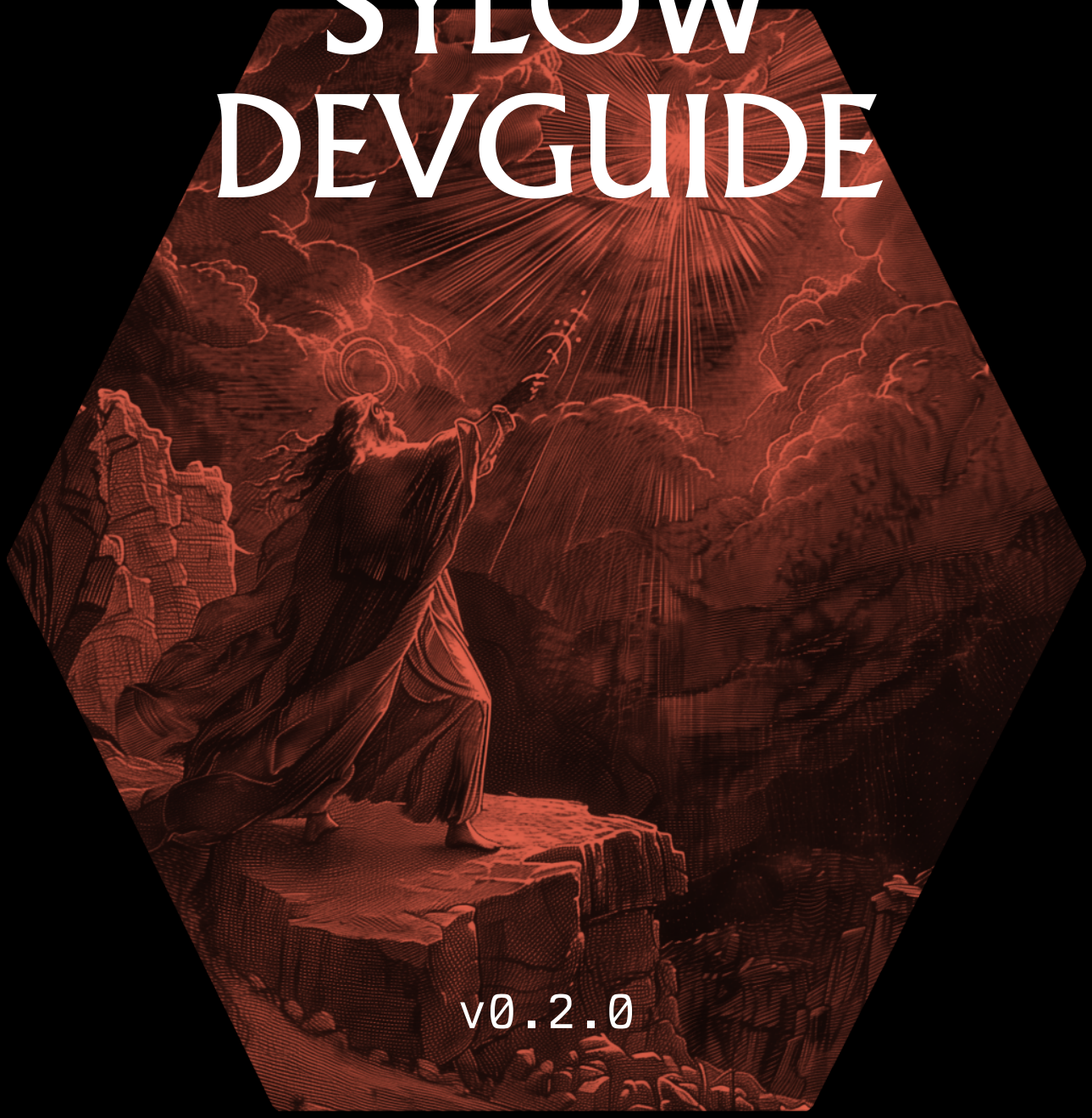
AUTHORS [3]
TRISTAN BRITT @trbritt
OxALCIBIADES @0xAlcibiades
GRUG @CapitalGrug
MARCH [2025]

[TITLE]

# SYLOW DEVGUIDE



v0.2.0

RESEARCH DIVISION — WARLOCK

# Introduction & Motivation

Secure multiparty compute and consensus mechanisms form the core of modern trustless cryptosystems such as Ethereum, underpinning frontier advancements in privacy and scalability. BLS signatures and elliptic curve cryptography provide the cryptographic foundations which allow Ethereum to scale to hundreds of thousands of validators whilst maintaining security, efficiency, and decentralization.

Elliptic curves support bilinear pairings which underpin key components on the frontier of crypto research – from ZK-SNARKs to generalized secure multi-party compute such as the consensus mechanisms in Ethereum 2.0. The importance of these cryptographic techniques in Ethereum's ecosystem cannot be overstated. Products like ZCash, IBM's Identity Mixer, Voltage SecureMail and iMessage all leverage ECC technology to provide strong privacy guarantees, credential verification, forward secrecy and more. There are many other such use cases which one can imagine, such as building out ZK rollups which are faster and more performant or enhancing pairing efficiency to accelerate the block signing process for Ethereum block builders, thereby reducing latency within the PBS auction. This optimization consequently facilitates improved throughput at the consensus layer, leading to overall network performance enhancement.

**Efficient Validator Identification and Accountability** Each validator has a public/private key pair relying on BLS cryptography, with the public key serving as the validator's identity in the protocol. Validators sign messages (attestations, blocks, etc.) with their private key, allowing other participants to verify that specific validators made particular attestations or proposed blocks. This ultimately holds validators accountable for their actions in the protocol.

**Signature Aggregation** BLS signatures have a unique property which allows them to be aggregated, namely hundreds or even thousands of individual signatures can be combined into a single aggregate signature in a secure and unbiased fashion. This aggregate signature can be verified singularly while still proving that all the individual validators signed. This dramatically reduces the amount of signature data that needs to be stored and transmitted, which are expensive operations.

**Committee Selection** ECC enables deterministic, pseudo-random selection of validators into committees, giving the protocol the ability to divide work amongst subsets of validators in an unpredictable, but importantly verifiable, manner.

**Randomness Generation** BLS signatures are used in the `RANDAO` mechanism to generate verifiable randomness for the protocol, which is crucial for committee selection and other core protocol functions, though notably not available on L2 networks more generally.

The general perception in the space about this area of research and development is that it is akin to esoteric black magic. Thus we endeavor here, for both ourselves and you, the reader, to provide elucidations on the rudiments of ECC cryptography, bilinear pairings, and other related topics. At WARLOCK, `Sylow` underpins our multi-party compute thesis for how we securely generate data feeds between nodes in the network via threshold signatures, which may then be verified on-chain with SolBLS.

This document provides a comprehensive mathematical and cryptographic primer for the `Sylow` library, which implements elliptic curve cryptography for the `BN_254` curve at present, with future support planned for `BLS_12_381` as that curve becomes more widely supported within the ecosystem. At present, `BN_254` is the only pairing capable curve supported widely in `EVM` environments. Notably, though there are many off-chain implementations of `BN_254` floating around, none are cryptographically secure and audited – which is why WARLOCK created `Sylow` and released it as a public good. The material covered ranges from foundational concepts in abstract algebra to advanced topics in pairing-based cryptography and distributed key generation.

Readers are expected to have a background in mathematics and computer science. The material is dense and technical in nature, reflecting the complexity of modern elliptic curve cryptography. To help with overall transparency of our design, code snippets and examples are provided to illustrate key concepts and their implementation in `Sylow`. By providing this comprehensive overview of the relevant mathematics and algorithms, we aim to enable deeper understanding and effective use of the `Sylow` library.

# Contents

[PAGE]

Introduction & Motivation 1

Glossary of Notation 5

Every Good Journey Starts With a Map 7

Mathematical Preliminaries 7
- Set Theory 7
  - Basic Definitions 7
  - Set Operations 7
  - Cartesian Product 8
  - Relations 8
  - Functions 8
  - Cardinality 8
- Group Theory 8
  - Groups 8
  - Abelian Groups 8
  - Finite Groups 8
  - Lagrange's Theorem 9
  - Subgroups 9
  - Homomorphisms and Isomorphisms 9
  - Cosets and Normal Subgroups 9
  - Cyclic Groups 9
  - Quotient Groups 9
- Ring Theory 10
  - Rings 10
  - Ideals 10
  - Quotient Rings 10
  - Polynomial Rings 10
- Number Theory 11
  - Prime Numbers and Divisibility 11
  - Greatest Common Divisor (GCD) 11
  - Integral Domains 11
  - Euclidean Domains 11
  - Euclidean Division 12
  - Extended Euclidean Division 12
- Field Theory 13
  - Fields 13
  - Division in Infinite Fields 13
  - Finite Fields 13
  - Modular Arithmetic in $\mathbb{Z}_p$ 13
  - Polynomial Modular Arithmetic in $\mathbb{Z}_p[x]$ 16
  - Basic Operations 16
  - Modular Reduction 16
  - Examples 16
  - Irreducible Polynomials 16
  - Subfields 17
- Field Extensions and Towers over Finite Fields 17
  - Basic Definitions 17
  - Field Extensions 18
  - Towers of Field Extensions 18

Algebraicity 18
Automorphisms and the Frobenius Endomorphism 19
Affine and Projective Varieties 19
Coordinate Rings 19
Properties of Algebraic Varieties 19
Roots of unity 20

Elliptic Curves **21**
Definition and Basic Properties 21
Group Law 22
Scalar Multiplication 23
Elliptic Curves over Finite Fields 23
Pairings 24

Optimal Ate Pairing **26**
Implementation 27
Final exponentiation 27
Easy part 28
Hard part 28
Glued Miller loops 29

BN, BN254, and BLS **30**
Barreto-Naehrig (BN) Curves 30
Definition and Properties 30
Parameterization 30
Construction and Usage 31
BN254 Specifics 31
Security 31
Representing Finite Fields, the scalar `Fp` 32
Field Definition 32
Internal Representation 32
Field Operations 32
Modular Arithmetic 33
Prime Order Finite Fields 33
The r-torsion 33
Representing Field Extensions 33
2nd Degree Extension 34
6th Degree Extension 34
12th Degree Extension 34
Twisting 34
Groups 35
$\mathbb{G}_1$ 35
$\mathbb{G}_2$ 35
$\mathbb{G}_T$ 36
Coordinate Geometries 36
Affine 36
Projective 37
Optimized Optimal Ate Pairing for BN_254 37
BLS Signature Scheme 38
Public and Private Keys 38
Signing a message 39
Verifying a signed message 39

Use Cases 39
Distributed Key Generation 39
Shamir's Secret Sharing 40
Feldman's VSS 40
Pedersen VSS 40
Feldman DKG 41
Pedersen DKG 41
A worked example 42
Threshold signatures 46
Generating Field Scalars 47
Generating Partial Private Keys and Cumulative Public Key 47
Partial Signing 48
Partial Verification 53
Signature Aggregation 53
Final Verification 54
Ethereum precompiles 54

Performance, Benchmarks & Scaling 62

Conclusion 63

# 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 $f$ |
| $\mathrm{id}_A$ | 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 |
| $\Leftrightarrow$, 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 |

| Notation | Meaning |
|---|---|
| $\Delta$ | 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 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 $G$ |
| $H \leqslant G$ | $H$ is a subgroup of $G$ |
| $H \triangleleft G$ | $H$ 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 $\varphi$ |
| $Aut(G)$ | Group of automorphisms of the group $G$ |
| $\rho$ | Canonical homomorphism |
| $\ker(\varphi)$ | Kernel of the homomorphism $\varphi$ |
| $N(H)$ | Normalizer of the subgroup $H$ |
| $R \oplus S$ | Direct sum of the rings $R$ and $S$ |
| $M_2(\mathbb{R})$ | Ring of all $2 \times 2$ 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 $F$ |
| $\text{irr}(a/F)$ | Irreducible polynomial of $a$ over $F$ |
| $\deg(a/F)$ | Degree of $a$ over $F$ |
| $[E : F]$ | 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 $F$ |

# Every Good Journey Starts With a Map

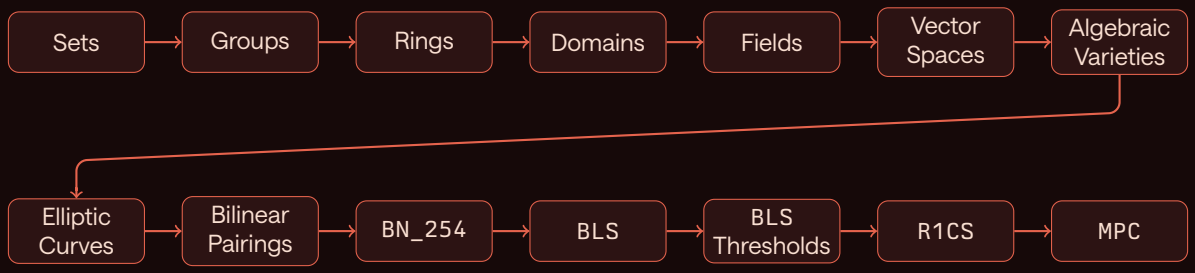
We'll begin with fundamental concepts in set theory, group theory, and ring theory. These will provide the basis for understanding more advanced structures like fields and vector spaces. Number theory and Euclidean domains will be introduced to provide essential tools for cryptographic applications.

As we progress, we'll delve into algebraic varieties and elliptic curves, which are crucial for understanding the Barreto-Naehrig (BN) 254 curve. We'll then explore bilinear pairings, which are fundamental to the Boneh–Lynn–Shacham (BLS) signature scheme.

Finally, we'll apply these concepts to the specific case of the `BN_254` curve and the BLS signature scheme, culminating in an exploration of threshold signatures, rank-one constraint system (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 `BN_254` 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 roughly:




# Mathematical Preliminaries

## 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

- A set is a collection of distinct objects, called elements or members of the set.
- If $a$ is an element of set $A$, we write $a \in A$.
- The empty set, denoted $\varnothing$, is the unique set with no elements.
- A set $A$ is a subset of set $B$, denoted $A \subseteq B$, if every element of $A$ is also an element of $B$.
- If a set $A$ is a subset of set $B$ but the two sets are not equal, then $A$ is a proper subset of $B$, denoted $A \subset B$.
- Every set $A \subseteq A$, or in other words, every set is contained by itself.

### Set Operations

Set theory defines several operations on sets:

- **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.
- **Intersection:** $A \cap B = \{x : x \in A \text{ and } x \in B\}$ The intersection contains all elements common to both sets.
- **Difference:** $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$ The difference contains elements in $A$ but not in $B$.
- **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 $B$:

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

### Relations

Given a Cartesian product $A \times B$, a relation between $A$ and $B$ is some subset **R** of $A \times B$. If ordered pair $(a, b)$ is in **R** we can write $a\mathbf{R}b$. Functions, discussed below, are a type of relation with certain properties.

### 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:

- **Injective (one-to-one):** $\forall a_1, a_2 \in A, f(a_1) = f(a_2) \implies a_1 = a_2$
- **Surjective (onto):** $\forall b \in B, \exists a \in A : f(a) = b$
- **Bijective (one-to-one correspondence):** 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 or denumerable:** 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 as the binary operator of the inverse of an element.

### Abelian Groups

As 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 $H \leqslant \mathbb{G}$. 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 $f : \mathbb{G} \to \mathbb{H}$ between groups is a homomorphism if it preserves the group operation: $f(a * b) = f(a) *' f(b) \forall a, b \in \mathbb{G}$.

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

### Cosets and Normal Subgroups

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

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

A subgroup $\mathbb{N}$ of $\mathbb{G}$ is called normal if $g\mathbb{N} = \mathbb{N}g \forall g \in \mathbb{G}$. We denote this as $\mathbb{N} \triangleleft \mathbb{G}$.

### 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:

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

### Quotient Groups

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

# 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:

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

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 $g(x) = \sum b_i x^i$,

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

Key properties:

- The zero polynomial, denoted $0$, has all coefficients equal to $0$.
- If $R$ has a unity $1 \neq 0$, then $R[x]$ has a unity, which is the constant polynomial $1$.
- $R$ is embedded in $R[x]$ as the set of constant polynomials.
- If $R$ is commutative, then $R[x]$ is commutative.
- 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:

- $\gcd(a, b) = \gcd(|a|, |b|)$
- $\gcd(a, b) = \gcd(b, a \bmod b)$ (basis for the Euclidean algorithm)
- 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 $b = 0$. (An example of zero divisors would be 2 and 3 in $\mathbb{Z}_6$, because both 2 and 3 are nonzero but $2 \cdot 3 = 0$ in this ring. 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:

- For all non-zero $a, b \in R$, $\delta(a) \leqslant \delta(ab)$
- 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 $0 \leqslant r < |b|$.

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

```
def 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 $y$.

```
def extended_euclidean_division(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        (gcd, xp, yp) = extended_gcd(b, a mod b)
        x = yp
        y = xp - floor(a / b) * yp
        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 four axioms:

1. $(F, +)$ is an abelian group with identity element $0$
2. $(F, +, \cdot)$ is a commutative ring with identity element $0$
3. $(F \setminus \{0\}, \cdot)$ or $F^*$ 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:

- The order (number of elements) of a finite field is always a prime power.
- For each prime power $q$, there exists a unique (up to isomorphism) finite field of order $q$.
- The multiplicative group $\mathbb{F}_p^* = \mathbb{F}_p\{0\}$ of a finite field (as defined in field axiom 3) 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:

- For prime fields $\mathbb{F}_p$:

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

- 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 $1 \leqslant k < n$ 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**

- For a prime number $p$, $\phi(p) = p - 1$
- $\phi$ is multiplicative: if $\gcd(a, b) = 1$, then $\phi(ab) = \phi(a) \cdot \phi(b)$
- 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:

$$\begin{aligned} -1^4 &= 1 \equiv 1 \pmod{5} - 2^4 \\ &= 2 \otimes 2 \otimes 2 \otimes 2 = 4 \otimes 2 \otimes 2 = 3 \otimes 2 \\ &= 1 \pmod{5} - 3^4 = 3 \otimes 3 \otimes 3 \otimes 3 = 2 \otimes 3 \otimes 3 = 1 \otimes 3 = 3 \pmod{5} - 4^4 \\ &= 4 \otimes 4 \otimes 4 \otimes 4 = 1 \otimes 4 \otimes 4 = 4 \otimes 4 = 1 \pmod{5} \end{aligned}$$

**Applications**

1. Finding multiplicative inverses: $a^{-1} \equiv a^{p-2} \pmod{p}$
   (a) 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 $a \neq 0$
   (a) 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:

$$\begin{aligned} [0] &= \{\ldots, -5, 0, 5, 10, \ldots\} \\ [1] &= \{\ldots, -4, 1, 6, 11, \ldots\} \\ [2] &= \{\ldots, -3, 2, 7, 12, \ldots\} \\ [3] &= \{\ldots, -2, 3, 8, 13, \ldots\} \\ [4] &= \{\ldots, -1, 4, 9, 14, \ldots\} \end{aligned}$$

**Co-prime Numbers** Two integers $a$ and $b$ are considered co-prime (or relatively prime) if their greatest common divisor (GCD) is 1. In other words:

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

Some key properties of co-prime numbers include:

- If $a$ and $b$ are co-prime, there exist integers $x$ and $y$ such that:

  $$ax + by = 1$$

  This is known as Bézout's identity.
- If $a$ and $b$ are co-prime, then:

  $$(a \bmod b) \text{ has a multiplicative inverse modulo } b$$

  This means there exists an integer $x$ such that:

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

- The product of co-prime numbers is co-prime to each of the original numbers.

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

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

### 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

- **Addition and Subtraction:** For $f(x) = \sum a_i x^i$ and $g(x) = \sum b_i x^i$,

$$(f + g)(x) = \sum (a_i \oplus b_i) x^i$$

$$(f - g)(x) = \sum (a_i \ominus b_i) x^i$$

- **Multiplication:**

$$(f \cdot g)(x) = \sum \left( \sum a_i \otimes b_j \right) x^{i+j}$$

- **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 the following five properties hold:

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

- **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 $0$) or $\mathbb{F}_p$ (if $F$ has characteristic $p$).
- **Tower Law**: If $E$ is a subfield of $F$ and $F$ is a subfield of $K$, then $[K : E] = [K : F][F : E]$.
- **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 $F$.
- **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 $\mathrm{Gal}(F/K)$.

#### 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

- **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.
- **Endomorphism**: An endomorphism of a field $F$ is a ring homomorphism from $F$ to itself. Unlike automorphisms, endomorphisms are not necessarily bijective.
- **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 $E \supset F$ and an element $\vartheta \in E$, the following conditions are equivalent:

- $\vartheta$ is a root of $f(t) \neq 0 \in F[t]$
- $\{1, \vartheta, \vartheta^2, \cdots\}$ are linearly independent on $F$;
- $F[\vartheta]$ is a field

$\vartheta$ 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 $\vartheta$ 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 $\phi$ 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.

## 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.

### Roots of unity

Let $\mathbb{G}$ be a multiplicative group and $n$ be a positive integer. An element $\omega \in \mathbb{G}$ is called an $n$-th root of unity if $\omega^n = 1$, where 1 is the identity element of the group. The set of all $n$-roots of unity forms a cyclic subgroup of $\mathbb{G}$ of order dividing $n$. We can also discuss primitive roots of unity if $\omega^k \neq 1$ for all $1 \leqslant k < n$, and these are generators of the cyclic subgroup of $n$-th roots of unity. These play important roles in the pairing operation as we discuss later.

---

# Elliptic Curves

The demand for robust, efficient, and scalable cryptographic systems increases constantly. While traditional cryptographic methods like RSA have served us well for decades, they face growing challenges in terms of key size, computational efficiency, and resistance to quantum computing attacks. This is where elliptic curve cryptography (ECC) steps in, offering a powerful alternative that addresses many of these concerns.

1. **Smaller Key Sizes**: ECC can provide the same level of security as RSA with significantly smaller key sizes. For example, a 256-bit ECC key offers comparable security to a 3072-bit RSA key.
2. **Improved Computational Efficiency**: The smaller key sizes in ECC translate to faster computations, especially on resource-constrained devices like smartphones or embedded devices.
3. **Quantum Resistance**: While not completely quantum-safe, ECC is believed to be more resistant to quantum computing attacks than RSA for equivalent security levels.
4. **Versatility**: ECC can be used for a wide range of cryptographic operations, including key agreement, digital signatures, and pseudorandom number generation.

## 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 $\mathcal{O}$. In characteristic not 2 or 3, every elliptic curve can be written in the typical functional form like one familiar from calculus:

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

where $a, b \in K$. This is called the short Weierstrass form of the curve. While in theory any pair of curve parameters $(a, b)$ could be used, there are only certain classes of these parameters which prove useful for practical implementations, a key metric for which is called the discriminant:

$$\Delta = -16(4a^3 + 27b^2) \neq 0$$

Key properties of elliptic curves include:

1. **Symmetry**: The curve is symmetric about the x-axis.
2. **Smoothness**: There are no cusps, self-intersections, or isolated points.
3. **Group Structure**: Points on the curve, along with a special "point at infinity," form a mathematical group under a geometrically defined addition operation.

Elliptic curves, when combined with the addition formulae below, have a rich algebraic structure that forms the basis of modern cryptography. The security of ECC relies on the difficulty of the "elliptic curve discrete logarithm problem" – given points $P$ and $Q$ on the curve, find the integer $k$ such that $Q = kP$ (where $kP$ means $P$ added to itself $k$ times).

**Geometric Interpretation** The way we combine points on these curves, the definition of our binary operator on these curves, will determine how viable these structures are for our goal of performing cryptography with them. For points $P, Q$ on $E$, we have the following rules for addition and doubling.

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 $P + Q$

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 $2P$

However, this geometric process encounters an issue in certain scenarios:

1. **Vertical Lines**: When adding $P$ and $-P$ (its reflection across the x-axis), the line connecting them is vertical and doesn't appear to intersect the curve at a third point.

2. **Tangent at Inflection Point**: When doubling a point that is an inflection point of the curve, the tangent line appears to intersect the curve only once.

To resolve these cases and maintain closure under our addition operation, we introduce the point at infinity. Conceptually, we can think of it as the point where all vertical lines intersect "at infinity."

**The Point at Infinity** From an algebraic perspective, the point at infinity serves several crucial roles:

1. **Identity Element**: In group theory, every group must have an identity element. For elliptic curves, $\mathcal{O}$ serves as this identity, satisfying $P + \mathcal{O} = P$ for all points $P$ on the curve.
2. **Inverse Elements**: The existence of inverse elements is another group axiom. For any point $P$ on the curve, its inverse $-P$ is defined such that $P + (x, -y) = \mathcal{O}$.
3. **Closure**: By including $\mathcal{O}$, we ensure that the addition operation is always defined, even in the special cases mentioned earlier. This closure property is essential for the set of points to form a group.

It is unfortunately difficult pragmatically to generate "infinity" on finite computation bounded machines, which complicates the implementation of ECC suites. This manifests itself in the fact that the point at infinity can be represented, in one form, as:

$$\mathcal{O} = \lim_{t \to 0^+} \left( \frac{x}{t}, \frac{y}{t} \right) \sim (\infty, \infty)$$

It turns out that there are other representations of the curve than short Weierstrass form that can get around this limitation.

**Affine and Projective Representations** Projective coordinates offer an alternative representation of points on an elliptic curve that circumvent the issue with infinity above. While affine coordinates $(x, y)$ are intuitive, even disregarding the issue with infinity, simple inspection of the group law formulae below reveals that arithmetic in affine coordinates requires inversion of field elements, which is in general expensive. Projective coordinates also aide to remove these expensive inversions.

In homogeneous projective coordinates, we represent a point $(x, y)$ on the curve as a triplet $(X : Y : Z)$, where:

$$x = \frac{X}{Z}, \quad y = \frac{Y}{Z}, \quad Z \neq 0$$

In this representation, The point at infinity $\mathcal{O}$ is simply $(0 : 1 : 0)$.

`Sylow` uses both representations of group elements throughout the code where necessary.

## Group Law

In the intuitive affine coordinates, the group of the elliptic curve, explicitly including the point at infinity, can be represented:

$$E = \left\{ (x, y) \in K^2 : y^2 = x^3 + ax + b \right\} \cup \{\mathcal{O}\}$$

while the group in projective coordinates can be represented:

$$E = \left\{ [X : Y : Z] \in \mathbb{P}^2(K) : Y^2 Z = X^3 + aXZ^2 + bZ^3 \right\}$$

**Algebraic Formulas** We give the basic formulae for the chord- and tangent rules below in affine coordinates, which is what you will see when encountering ECC from any textbook. `Sylow` implements optimized versions of these formulae for arithmetic, but inspection of those algorithms is left as an exercise to the reader.

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

**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, and serves as the basis on which security stands in elliptic curves. Namely, this scalar multiplication satisfies a few key properties:

1. **No known efficient algorithm:** There's no known algorithm that can solve the elliptic curve discrete log problem (ECDLP) in polynomial time for general elliptic curves. The best known general algorithms for solving ECDLP (like Pollard's rho algorithm) have exponential time complexity.
2. **Large prime order:** In cryptographic applications, elliptic curves are chosen with a large prime order. This means the number of possible scalar values is extremely large, making brute-force attempts infeasible.
3. **Obfuscated relationship:** There is no clear relationship between the individual coordinates of the group element representation and the scalar $n$, further compounding the DL problem.

**Double-and-Add Algorithm** An efficient method to compute $[n]P$ is given by the "double-and-add" algorithm:

**Algorithm 1** Double-and-Add Algorithm

**Require:** $n \in \mathbb{Z}^+$, $P \in E(\mathbb{F}_q)$
**Ensure:** $Q = [n]P$

Convert $n$ to binary: $n = \sum_{i=0}^{k} b_i 2^i$
$Q \leftarrow \mathcal{O}$ ▷ $\mathcal{O}$ is the point at infinity
**for** $i = k$ **down to** $0$ **do**
    $Q \leftarrow 2Q$
    **if** $b_i = 1$ **then**
        $Q \leftarrow Q + P$
    **end if**
**end for**
**return** $Q$

## Elliptic Curves over Finite Fields

When considering these curves for the first time, they are typically represented over the reals / complex numbers:

$$E = \left\{(x, y) \in \mathbb{R}^2 : y^2 = x^3 + ax + b\right\} \cup \{\mathcal{O}\}$$

which looks like a sideways Lululemon logo. This picture is incredibly useful for understanding key aspects of elliptic curves, such as the chord and tangent rules, therefore the group structure, as well as the impact of the curve parameters $(a, b)$ on the behaviour / singularity of the curve, and others. However, this is not the base field over which elliptic curves are generated in practice for many reasons:

1. **ECDLP:** there is no discrete log problem over the real numbers, since $\forall x \in \mathbb{R}, \exists y | \frac{x}{y} \in \mathbb{R}$, which is an easily calculable quantity.
2. **Exact represenation:** Finite field elements can be represented exactly in computers, unlike real numbers which require approximations and various IEEE standards to ensure consistency of implementation between machines.

3. **Cardinality of the base field:** In the reals, there is an uncountably large number of potential elements in the group, which makes consistency of group arithmetic difficult.

Therefore, we define elliptic curves over a finite field $K$ (typically $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$), forming a finite Abelian group $E(K)$.

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

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

and is a key quantity to consider when choosing a curve, as it determines the total possible number of elements in the group, and therefore the likelihood / difficulty of a brute force attack to the ECDLP.

## Pairings

Pairings, also known as bilinear maps, have become an essential tool in modern cryptography, particularly in the realm of ECC. Their importance stems from several key factors:

1. **Tripartite Diffie-Hellman**: Pairings enable the construction of one-round tripartite key agreements, which was a long-standing open problem before their introduction.
2. **Identity-Based Encryption (IBE)**: Pairings made practical implementations of IBE possible, allowing public keys to be derived from arbitrary strings like email addresses.
3. **Short Signatures**: Pairing-based cryptography enables the creation of very short digital signatures, such as the BLS signature scheme.
4. **Attribute-Based Encryption**: Pairings facilitate more complex access structures in encryption schemes, allowing for fine-grained access control.
5. **Homomorphic Encryption**: Some partially homomorphic encryption schemes leverage pairings for their unique properties.
6. **Zero-Knowledge Proofs**: Pairings are crucial in many efficient zero-knowledge proof systems, particularly in zk-SNARKs (Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge).

A pairing is a bilinear map:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

where $\mathbb{G}_1$ and $\mathbb{G}_2$ are typically additive groups (often subgroups of points on an elliptic curve), and $\mathbb{G}_T$ is a multiplicative group. The key property is bilinearity:

$$e(aP, bQ) = e(P, Q)^{ab}$$

for all $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}$. In general these pairings involve two steps in their implementation: the evaluation of an interior function, and the exponentiation of the result of that function to some exponent. The efficient implementation of the evaluation of the interior function is what is known as “Miller’s loops”, which is followed by the “final exponentiation” step. Generally speaking, both of these operations are expensive to compute given the size of the curve orders used.

There is a great outline of the many types of pairings here, in one of the original manuscripts on the subject.

**Weil Pairing** The Weil pairing, denoted $e_n(P, Q)$, is defined for points $P$ and $Q$ of order $n$ on an elliptic curve $E$ over a finite field $\mathbb{F}_q$. It maps to the group of $n$-th roots of unity in an extension field of $\mathbb{F}_q$:

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

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

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 Tate pairing, often denoted $\tau_n(P, Q)$, is defined similarly but is defined on a broader class of inputs, is faster to compute than the Weil pairing, and outputs elements in a quotient group, which requires an additional final exponentiation in practice:

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

where $k$ is the embedding degree.

---

# Optimal Ate Pairing

The Weil and Tate pairings are pedagogically those mentioned in any ECC reference material, but is not the pairing used in `Sylow` , which is so important to the usage of `Sylow` that it is worth mentioning in some detail herein.

The development of the optimal ate pairing is part of a continuous effort to improve the efficiency of pairing computations in ECC. To understand its significance, we need to consider the evolution from the Weil and Tate pairings.

The Weil pairing requires two Miller loop calculations, making it relatively slow, and the Tate pairing requires a costly final exponentiation step. The optimal ate mapping achieves the theoretical minimum number of iterations in the Miller loop for a given embedding degree, and is optimized for specific families of pairing-friendly curves, like Barreto-Naehrig (BN) curves. For BN curves, the loop length can be as low as $\log(r)/\phi(k)$.

Recall that the goal here is to take a point $X \in \mathbb{G}_1 = E(\mathbb{F}_p)$, and a point $Y \in \mathbb{G}_2 \subset E'(\mathbb{F}_{p^2})$, 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 $\ell$ 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:

$$e(X, Y) = \left( f_{6z+2,Y}(X)\ell_{[6z+2]\Psi(Y),\phi_p(\Psi(Y))}(X)\ell_{[6z+2]\Psi(Y)+\phi_p(\Psi(Y)),-\phi_p(\Psi(Y))}(X) \right)^{\frac{p^{12}-1}{r}}$$

Now say that 5 times fast. Here, $z$ is the parameter of the curve. The pairing is based on rational functions $f_{i,Q} : \mathbb{N} \times \mathbb{G}_2 \times \mathbb{G}_1 \to \mathbb{F}_{p^{12}}$ 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 implementation of which is referred 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 $\Psi$ 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.

## Implementation

In decimals, we know $z = 4965661367192848881$, and therefore $6z + 2 = 29793968203157093288$. Optimised implementations represent this bound in the windowned non-adjacent form (wNAF), not binary, since it has a lower Hamming weight.

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

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

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

Since $[n]\Psi(Q) = \Psi([n]Q)$ by the homomorphism, we just evaluate the line now at the point:

$$Q' = (\xi^{(p-1)/3}\overline{x}', \xi^{(p-1)/2}\overline{y}') = (x_1, y_1)$$

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

## 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 will be worse.

The following takes the lead from this and that.

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

Well, up until this point, we were precise in the 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 $\mu_r$. Why the roots of unity? Great question.

Remember that this mapping has to satisfy a few key real-world properties:

- **It has to be a trapdoor**, namely pre-image resistance (assuming DL hardness), and mapping backwards from the roots of unity is a very difficult problem.
- **There must be 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 BN_254, 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 on the 12th order extension. 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 $\overline{x} = a - bw$). Then taking $(x^{p^6-1})^{p^2}$ is just applying our Frobenius morphism $\phi$, and then finally we multiply by our already-computed value $x^{p^6-1}$, and voilà!

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

### Hard part

For BN_254, 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 $\tau$ of $w+1$ integers $\tau = (\lambda_0, \ldots, \lambda_w)$ such that $\lambda = \sum \lambda_i p^i$ minimizing the L1-norm of $\tau$. 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 $f_{easy}^z$, $(f_{easy}^z)^z$, $(f_{easy}^{z^2})^z$
2. Use the Frobenius operator, which has efficient representations in powers 1, 2, and 3 of the prime, to compute $f_{easy}^p$, $f_{easy}^{p^2}$, $f_{easy}^{p^3}$, $(f_{easy}^z)^p$, $(f_{easy}^{z^2})^p$, $(f_{easy}^{z^3})^p$, $(f_{easy}^{z^3})^{p^2}$

The evaluation then amounts to:

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

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 multi-exponentials which we take from the original manuscript:

$$\begin{aligned}
T_0 &\leftarrow (y_6)^2 \\
T_0 &\leftarrow T_0 \cdot y_4 \\
T_0 &\leftarrow T_0 \cdot y_5 \\
T_1 &\leftarrow y_3 \cdot y_5 \\
T_1 &\leftarrow T_1 \cdot T_0 \\
T_0 &\leftarrow T_0 \cdot y_2 \\
T_1 &\leftarrow (T_1)^2 \\
T_1 &\leftarrow T_1 \cdot T_0 \\
T_1 &\leftarrow (T_1)^2 \\
T_0 &\leftarrow T_1 \cdot y_1 \\
T_1 &\leftarrow T_1 \cdot y_0 \\
T_0 &\leftarrow (T_0)^2 \\
f_{hard} &\leftarrow T_0 \cdot T_1
\end{aligned}$$

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

## Glued Miller loops

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 gist 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.

# BN, BN254, and BLS

Now, having covered the preliminaries required, we begin our discussion of Barreto-Naehrig (`BN`) curves and Boneh-Lynn-Shacham (`BLS`) signature scheme. For `Sylow` we chose to implement initially over the `BN_254` curve, with a highly optimized implementation, and use the `BLS` signature schema. The reasoning here is that as of writing, in Summer of 2024, `BN_254` has support in the Ethereum Virtual Machine (EVM) at precompile `0x08`.

Recall from the front matter that our primary use cases at WARLOCK for elliptic curve cryptography, bilinear pairings, and threshold signature schemes revolve around multi-party computation and collaborative entropy generation, namely using these for verifying the integrity of values on the blockchain. Thus, we needed a method which was flexible to a dynamic network and supported on-chain. This selection met those criteria, though perhaps an alternative curve like `BLS_12_381` will be the premier choice in the near future as support rolls out for that in a future hard fork.

With that said, all of the particulars we will discuss, if not all of the low level optimizations, extend to other curves in this family more generally.

Here we'll rebuild the particulars of the library and its optimizations into the narrative of the choices made during developments of `Sylow` , what it can be used for now, and some forward looking goals as well.

## Barreto-Naehrig (BN) Curves

Before we get into that, and having general intuitions about the way these sort of elliptic curves work, let's get into the particulars of the BN family of curves.

### Definition and Properties

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

1. **The order of BN curves is always a prime number** $n$**.** What this means is that the underlying field is of a prime order, and thus forms some kind of normal algebra in a modular sense, if you think back to the discussion in the math primer.
2. **The embedding degree is** $k = 12$, which provides a good balance between security and efficiency for pairing-based cryptography. It's large enough to resist index calculus attacks on the discrete logarithm problem in the extension field, whilst also enabling secure and efficient implementation of field arithmetic in $\mathbb{F}_{p^{12}}$.
3. **The CM discriminant is** $D = 3$, which enables the use of efficiently computable endomorphisms, which provide efficient scalar multiplication, additionally making the curve pairing friendly.
4. **The equation form for BN curves is** $E : y^2 = x^3 + b$**, where** $b \neq 0$. This makes computation more efficient when compared to general Weierstrass form curves. Additionally, it allows for the efficient implementation of certain optimized pairings we'll be looking at shortly, along with compressed point representations and a host of other benefits.

### Parameterization

BN curves are parameterized by a single $x \in \mathbb{Z}^+$. The key parameters of the curve are defined as polynomials in $x$. The polynomials defining the key parameters of BN curves are carefully constructed to achieve several desirable properties simultaneously. Let's examine each in turn:

1. **Trace of Frobenius:** $t(x) = 6x^2 + 1$

   The trace of Frobenius is a fundamental concept in the theory of elliptic curves over finite fields. For a curve $E$ defined over a finite field $\mathbb{F}_p$, the trace $t$ satisfies:

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

   where $\#E(\mathbb{F}_p)$ denotes the number of points on the curve (including the point at infinity).

   For BN curves, parameterizing the trace as $t(x) = 6x^2 + 1$ allows for efficient implementation and yields desirable properties for pairing-based cryptography.

2. **Prime field order:** $p(x) = 36x^4 - 36x^3 + 24x^2 - 6x + 1$

   This polynomial defines the order of the finite field $\mathbb{F}_p$ over which the BN curve is defined. For a given value of $x$, $p(x)$ must be prime for the curve to be cryptographically suitable. The structure of this polynomial ensures that $p$ is close to a power of 2, potentially leading to efficient field arithmetic.

3. **Curve order:** $n(x) = 36x^4 - 36x^3 + 18x^2 - 6x + 1$

   This polynomial defines the number of points on the curve, including the point at infinity. For BN curves, this value is always prime, a property crucial for many cryptographic applications. The primality of the curve order ensures the existence of a large prime-order subgroup, which is essential for security.

This relationship holds for all elliptic curves over finite fields and is built into the design of the BN curve parameterization.

The meticulous construction of these polynomials allows BN curves to achieve several desirable properties concurrently:

- The field order $p$ is prime (for suitable $x$).
- The curve order $n$ is prime.
- The embedding degree (which determines pairing efficiency) is always 12.
- Both $p$ and $n$ are close in magnitude, beneficial for certain pairing-based protocols.

These properties render BN curves particularly well-suited for pairing-based cryptography, offering an optimal balance between security and efficiency. The ability to fine-tune these parameters by choosing an appropriate $x$ allows for the creation of curves tailored to specific security requirements or implementation constraints.

### Construction and Usage

To construct a BN curve for cryptographic use we:

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. Utilize the curve equation $E : y^2 = x^3 + b$, where $b$ is typically chosen to be a small integer (often 2 or 3).
4. Define the curve over $\mathbb{F}_p$, where $p = p(x)$.
5. Define the order of the curve as $n = n(x)$.

## BN254 Specifics

As we touched upon earlier, `BN_254` is the *only* supported curve for bilinear pairings at present in the Ethereum client ecosystem and specification. Thus, since our primary goal is the secure and scalable verification of data produced by the Warlock network on chain, we delve into the particulars here. The `BN_254` curve particularly has these parameters:

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

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

**Curve equation:** $y^2 = x^3 + 3$

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

**Embedding degree:** $k = 12$

### Security

In theory, the curve has $L = 128$ bit security, but it was shown to now be $\approx 100$. 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).

This is the primary reason why the `BLS_12_381` curve will be both preferred and supported in future versions of `Sylow` once there is broader support in the execution layer pre-compiles (in fact, and as we alluded to in the introduction, it's already broadly used in consensus).

## Representing Finite Fields, the scalar Fp

The `Sylow` library implements the base scalar field `Fp`, which forms the foundation for all field operations and extensions used in the `BN_254` elliptic curve cryptography. This field is the finite field of prime order $p$, where $p$ is the `BN_254` curve's base field modulus. The implementation provides optimized multi-precision arithmetic operations for use in higher level abstractions.

There are some future improvements to be made here around the `SIMD` vectorized computation of these, which will yield significant improvement in the already solid performance of `Sylow`.

### Field Definition

The scalar field `Fp` is defined using a macro that creates a new type with the necessary traits and operations:

```
define_finite_prime_field!(
    Fp,
    FpModStruct,
    FpOutputType,
    U256,
    8,
    BN254_PRIME_STRING,
    1,
    1
);
```

Here:

- `Fp` is the name of the new type representing field elements.
- `FpModStruct` and `FpOutputType` are internal types used in the implementation.
- `U256` specifies that field elements are represented using 256-bit unsigned integers.
- `8` indicates that the field elements are stored using 8 32-bit words.
- `BN254_PRIME_STRING` is a constant string representing the prime modulus p.

### Internal Representation

Internally, each `Fp` element is represented as a 256-bit unsigned integer (`U256`). This allows for efficient arithmetic operations and comparisons. The `U256` type is provided by the `crypto-bigint` crate, which offers constant-time implementations of big integer operations.

### Field Operations

The `define_finite_prime_field!` macro automatically implements various traits and operations for the `Fp` type, including:

- Addition and subtraction modulo $p$
- Multiplication modulo $p$
- Inversion modulo $p$
- Equality and comparison operations
- Conversion to and from standard Rust integer types
- Serialization and deserialization

All these operations are implemented to run in constant time to prevent timing side-channel attacks.

### Modular Arithmetic

Arithmetic operations in `Fp` are performed modulo the prime $p$. In general, there are multiple ways to perform modular arithmetic on finite fields, but the *de facto* standard for performance is Montgomery arithmetic, which is how all base field arithmetic is implemented in `Sylow`. This leverages multiple precomputed constants to accelerate binary operations on the underlying `U256` types.

Furthermore, algorithms for arithmetic in the Montgomery scheme can be implemented in $\mathcal{O}(1)$, contributing to the overall constant-time execution of `Sylow`. Explicit traits from the `subtle` crate that provide constant-time functionality are included throughout the library.

### Prime Order Finite Fields

As discussed in the primer, there are a number of both key and desirable properties for choosing a prime modulus $p$ which make this all work. However, within these fields, there is additional structure leveraged, which is worth discussing here.

### The r-torsion

Let $G$ be a group. The $r$-torsion of the group is defined by $\{x \in G \mid rX = \mathcal{O}\}$, 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.

## Representing Field Extensions

We will eventually want to use elements ascending upwards from a simple scalar to more complex structures, climbing towards the dizzying heights of the dodectic extension. Therefore, 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 $\{a_{m-1}x^{m-1} + \cdots + a_1 x + a_0 \mid a_i \in \mathbb{F}_p\}$

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 `BN_254` is given by the following (see here or here):

$$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 - \beta)$$

$$\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi)$$

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

where $\beta$ is a quadratic non-residue in $\mathbb{F}_p$ and $\xi$ 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.

In `Sylow`, extensions are created by structs satisfying the following trait:

```rust
pub trait FieldExtensionTrait<const D: usize, const N: usize>:
    Sized + Copy + Clone + std::fmt::Debug + Default
    + Add<Output = Self> + AddAssign
    + Sub<Output = Self> + SubAssign
    + Mul<Output = Self> + MulAssign
    + Div<Output = Self> + DivAssign
    + Inv<Output = Self>
    + Neg<Output = Self>
    + PartialEq
    + ConstantTimeEq + ConditionallySelectable
    + Zero + One
    + From<u64>
{ ... }

#[derive(Copy, Clone, Debug)]
pub struct FieldExtension<const D: usize, const N: usize, F: FieldExtensionTrait<D, N>>(
    pub(crate) [F; N],
);
```

### 2nd Degree Extension

There are many additional helper functions that aid in efficient operations on the quadratic extension, defined in `Sylow` as:

```rust
pub type Fp2 = FieldExtension<2, 2, Fp>;
impl FieldExtensionTrait<2, 2> for Fp2 { ... }
```

### 6th Degree Extension

Likewise, we choose to represent the sextic extension as 3 elements from $\mathbb{F}_{p^2}$:

```rust
pub type Fp6 = FieldExtension<6, 3, Fp2>;
impl FieldExtensionTrait<6, 3> for Fp6 { ... }
```

### 12th Degree Extension

And finally, we represent the dodectic extension as 2 elements from $\mathbb{F}_{p^6}$:

```rust
pub type Fp12 = FieldExtension<12, 2, Fp6>;
impl FieldExtensionTrait<12, 2> for Fp12 { ... }
```

It is worth mentioning that there are, in multiple instances, places in `Sylow` where alternative representations of an element from $\mathbb{F}_{p^{12}}$ are used for efficiency.

## Twisting

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 `BN_254`, 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 elements of in $\mathbb{Z}_p^+$, while points on the twist are pairs of complex positive integers $\mathbb{C}_p^{++}$, 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!

---

## Groups

In `Sylow` , groups are designed to be flexible with arbitrary base fields, and there are different representations of group elements that are allowed in various places. In general, a group is a struct satisfying the relevant trait:

```
pub trait GroupTrait<const D: usize, const N: usize, F: FieldExtensionTrait<D, N>>:
    Sized + Copy + Clone + std::fmt::Debug + Default
    + ConstantTimeEq + ConditionallySelectable + PartialEq
    + Neg
{ ... }
```

$\mathbb{G}_1$

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 naïve 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:

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

**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 $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ 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 (i.e. $X^d \equiv Y \implies [d]X = Y$). 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 $\ell$ 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.

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, i.e. these are all points of order dividing $r$, we finally define

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

**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 be shown that the set of points fixed by $\phi$ 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.

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, i.e. $\#E'(\mathbb{F}_{p^2}) = c_2 r$, 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 was not adequately performant for our implementation.

Thankfully, faster algorithms exist. For instance, we define 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 $\Psi$ is the twist mapping, and recall $\xi = 9 + u$. Membership in $\mathbb{G}_2$ therefore boils down to verifying if the following holds: $\psi(Q) = [6x^2]Q$, and more recent work improves this to the verifying the following equality:

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

which is the version of subgroup membership check used in `Sylow`, but you can go even further too at the cost of readability and maintainability.

$\mathbb{G}_T$

The image of the optimal ate pairing is the group of $r$-th roots of unity in the dodectic extension, and therefore an element of this group can be cleanly represented as a single $\mathbb{F}_{p^{12}}$:

```rust
#[derive(Copy, Clone, Debug, Default)]
pub struct Gt(pub(crate) Fp12);
impl GroupTrait<12, 2, Fp12> for Gt { ... }
```

## Coordinate Geometries

### Affine

As we discussed in the preliminaries, an affine coordinate is a simple ordered pair with two elements, $x$ and $y$, as one would expect in regular Cartesian geometry, however here over the rarefied spaces of the finite field extension tower. The affine representation of a group element in `Sylow` is given by the following generic struct:

```rust
#[derive(Copy, Clone, Debug)]
pub struct GroupAffine<const D: usize, const N: usize, F: FieldExtensionTrait<D, N>>
    {
    /// x-coordinate
    pub(crate) x: F,
    /// y-coordinate
    pub(crate) y: F,
    /// Flag indicating if this is the point at infinity
    pub(crate) infinity: Choice,
}
```

**G1Affine** This specializes to $\mathbb{G}_1$ for `BN_254`...

```
pub type G1Affine = GroupAffine<1, 1, Fp>;
impl GroupTrait<1, 1, Fp> for G1Affine {  ...  }
```

**G2Affine** ...and likewise for $\mathbb{G}_2$.

```
pub type G2Affine = GroupAffine<2, 2, Fp2>;
impl GroupTrait<2, 2, Fp2> for G2Affine {  ...  }
```

### Projective

Projective coordinates in the context of `Sylow`, on the other hand, are a triplet – $(x, y, z)$, where we use $z$ to represent the "infiniteness" of an affine coordinate. Thus, to cleanly account for arithmetic involving $\mathcal{O}$, we provide the projective representation of group elements:

```
#[derive(Copy, Clone, Debug)]
pub struct GroupProjective<const D: usize, const N: usize, F: FieldExtensionTrait<D,
    N>> {
    // We now define the projective representation of a point on the curve. Here,
   the point(s) at
    // infinity is (are) indicated by `z=0`. This allows the user to therefore
   directly generate
    // such a point with the associated `new` method. Affine coordinates are those
   indicated by `z=1`.
    // Therefore, the user could input a `Z>1`, but this would fail the curve check,
    and therefore
    // make the code panic.
    // Implementing addition and multiplication requires some thought. There are
   three ways that we
    // could in theory do it: (i) have both points in affine coords, (ii) have both
   points in
    // projective coords, or (iii) have mixed representations. For security, due to
   the uniqueness of
    // the representation of the point at infinity, we therefore opt to have
    // all arithmetic done in projective coordinates.
    pub(crate) x: F,
    pub(crate) y: F,
    pub(crate) z: F,
}
```

**G1Projective** This specializes to $\mathbb{G}_1$ for `BN_254` ...

```
pub type G1Projective = GroupProjective<1, 1, Fp>;
impl GroupTrait<1, 1, Fp> for G1Projective
```

**G2Projective** ... and likewise for $\mathbb{G}_2$.

```
pub type G2Projective = GroupProjective<2, 2, Fp2>;
impl GroupTrait<2, 2, Fp2> for G2Projective
```

## Optimized Optimal Ate Pairing for BN_254

As we mentioned in the previous section on the math of the optimal ate pairing, this boils down to the evaluation of Miller loops, followed by a final exponentiation step. The combination of these is handled by the user-facing `pairing` method:

```rust
pub fn pairing(p: &G1Projective, q: &G2Projective) -> Gt {
    // Convert inputs to affine coordinates
    let p = &G1Affine::from(p);
    let q = &G2Affine::from(q);

    // Handle point at infinity (zero) cases
    let either_zero = Choice::from((p.is_zero() | q.is_zero()) as u8);
    let p = G1Affine::conditional_select(p, &G1Affine::generator(), either_zero);
    let q = G2Affine::conditional_select(q, &G2Affine::generator(), either_zero);

    // Compute the Miller loop
    let tmp = q.precompute().miller_loop(&p).0;

    // Conditional selection to handle zero cases
    let tmp = MillerLoopResult(Fp12::conditional_select(&tmp, &Fp12::one(),
    either_zero));

    // Perform the final exponentiation and return the result
    tmp.final_exponentiation()
}
```

This implementation relies on the fact that each iteration of the Miller loop will involve a set of 87 constants from $\mathbb{F}_{p^2}$ that are static for a given $\mathbb{G}_2$ element in the pairing. Why 87? 64 iterations through the NAF (Non-Adjacent Form) representation, each requiring a doubling step. 9additions for 1 digits in the NAF. 12 additions for $-1$ digits in the NAF. 2 final addition steps. This totals to $64 + 9 + 12 + 2 = 87$ coefficients. This allows for a pre-computation step that can be used for optimized evaluation of multiple verifications for changing $\mathbb{G}_1$ inputs.

## BLS Signature Scheme

Sylow allows for the usage of the ECC logic to provide standard functionality for private-public key cryptography more generally. The Boneh-Lynn-Shacham BLS signature scheme is particularly useful in the context of Sylow for several reasons:

- **Short signatures** BLS produces very compact signatures, typically around 170 bits for 1024-bit security, which is about half the size of DSA signatures with comparable security. This makes BLS ideal for applications with bandwidth or storage constraints.
- **Aggregation** BLS signatures can be aggregated, meaning multiple signatures by different parties on different messages can be combined into a single short signature. This enables efficient verification of many signatures at once. This ability of aggregation also enables threshold signing, where a signature can only be produced when a threshold number of parties cooperate.
- **Efficiency** Signature generation in BLS is very fast, consisting of just a single elliptic curve point multiplication. Verification is slower but can be batched for improved performance.
- **Pairing-friendly** BLS leverages bilinear pairings on elliptic curves, which aligns well with Sylow 's focus on efficient pairing computations for the BN254 curve.

### Public and Private Keys

```rust
#[derive(Debug, Copy, Clone)]
pub struct KeyPair {
    // The secret key, represented as a scalar in the base field
    pub secret_key: Fp,
    // The public key, represented as a point on the \(\mathbb{G}_2\) group
    pub public_key: G2Projective,
}
impl KeyPair {
    pub fn generate() -> KeyPair {
```

```
        let secret_key = Fp::new(Fr::rand(&mut OsRng).value());
        let public_key = G2Projective::generator() * secret_key;
        KeyPair {
            secret_key,
            public_key,
        }
    }
}
```

### Signing a message

We can then use this key to sign an arbitrary message using the built-in `hash-to-curve` functionality. By default, `Sylow` uses the `XMD` message expander, as this is the recommended choice for `SHA3` hashers, as well as `keccak256`:

```
pub fn sign(k: &Fp, msg: &[u8]) -> Result<G1Projective, GroupError> {
    // Expand the message to a curve point using the DST and security bits
    let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
    // Hash the message to a curve point, returning the point in \(\mathbb{G}_1\)
   multiplied by the secret key or an error
    match G1Projective::hash_to_curve(&expander, msg) {
        Ok(hashed_message) => Ok(hashed_message * *k),
        _ => Err(GroupError::CannotHashToGroup),
    }
}
```

### Verifying a signed message

Having signed a message, we can confirm it was valid by executing a pairing operation:

```
pub fn verify(pubkey: &G2Projective, msg: &[u8], sig: &G1Projective) -> Result<bool,
    GroupError> {
    // Expand the message to a curve point using the DST and security bits
    let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
    // Assert that the message can be hashed to a curve point and the pairings
   compared,
    // returning a boolean or an error
    match G1Projective::hash_to_curve(&expander, msg) {
        Ok(hashed_message) => {
            let lhs = pairing(sig, &G2Projective::generator());
            let rhs = pairing(&hashed_message, pubkey);
            Ok(lhs.ct_eq(&rhs).into())
        }
        _ => Err(GroupError::CannotHashToGroup),
    }
}
```

# Use Cases

## Distributed Key Generation

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 $\lambda$ 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 VSS. 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 $t$:

$$q(x) = \sum_k a_k x^k, \quad \widetilde{q}(x) = \sum_k b_k x^k$$

both in $\in \mathbb{Z}_p[x]$, and again $q(0) = a_0$ the shared secret. The dealer then transmits to each participant $i$ their share

$$s_i = q(i), \quad \widetilde{s}_i = \widetilde{q}(i)$$

and broadcasts the values:

$$\{C_k = g^{a_k} g^{b_k} \mod p\}_{k=0,t}$$

Verification of the shared secret follows by:

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

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

### 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 build on top of 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.

### A worked example

```rust
use num_traits::{Inv, One, Zero};
use serde::{Deserialize, Serialize};
use sha3::Keccak256;
use std::collections::HashMap;
use std::path::PathBuf;
use sylow::{
    glued_miller_loop, FieldExtensionTrait, Fr, G1Affine, G1Projective, G2Affine,
    G2Projective,
    GroupTrait, Gt, XMDExpander,
};

const DST: &[u8; 30] = b"WARLOCK-CHAOS-V01-CS01-SHA-256";

const SECURITY_BITS: u64 = 128;

#[derive(Debug, Serialize, Deserialize, Default)]
struct DkgConfig {
    quorum: u32,
    n_participants: u32,
    n_rounds: u32,
}

#[derive(Clone, Debug)]
```

```rust
struct Participant {
    id: usize,
    secret_key: Fr,
    public_key: G2Projective,
}

struct DistributedKeyGeneration {
    t: usize,
    n: usize,
    participants: Vec<Participant>,
    group_public_key: G2Projective,
}

impl DistributedKeyGeneration {
    fn new(t: usize, n: usize) -> Self {
        assert!(
            t <= n,
            "Threshold must be less than or equal to the number of participants"
        );

        let mut rng = rand::thread_rng();
        let secret_polynomial: Vec<Fr> = (0..t).map(|_| Fr::rand(&mut rng)).collect
    ();

        let participants: Vec<Participant> = (1..=n)
            .map(|id| {
                let secret_key = Self::evaluate_polynomial(&secret_polynomial, id);
                let public_key = G2Projective::generator() * Fr::into(secret_key);
                Participant {
                    id,
                    secret_key,
                    public_key,
                }
            })
            .collect();

        let group_public_key = G2Projective::generator() * Fr::into(
    secret_polynomial[0]);

        DistributedKeyGeneration {
            t,
            n,
            participants,
            group_public_key,
        }
    }

    fn evaluate_polynomial(coeffs: &[Fr], x: usize) -> Fr {
        let x = Fr::from(x as u64);
        coeffs
            .iter()
            .rev()
            .fold(Fr::zero(), |acc, coeff| acc * x + *coeff)
    }

    fn partial_sign(&self, participant_id: usize, message: &[u8]) -> G1Projective {
        let participant = self
```

```
            .participants
            .iter()
            .find(|p| p.id == participant_id)
            .expect("Participant not found");

        let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
        let hashed_message = G1Projective::hash_to_curve(&expander, message)
            .expect("Failed to hash message to curve");

        hashed_message * Fr::into(participant.secret_key)
    }

    fn batch_verify_partial(
        &self,
        message: &[u8],
        signatures: &HashMap<usize, G1Projective>,
    ) -> bool {
        let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
        let hashed_message = G1Projective::hash_to_curve(&expander, message)
            .expect("Failed to hash message to curve");

        let g2_gen = G2Affine::from(G2Projective::generator());
        let mut g1_points = Vec::new();
        let mut g2_points = Vec::new();

        for (&id, signature) in signatures {
            let participant = self
                .participants
                .iter()
                .find(|p| p.id == id)
                .expect("Participant not found");

            g1_points.push(G1Affine::from(*signature));
            g1_points.push(G1Affine::from(-hashed_message));
            g2_points.push(g2_gen);
            g2_points.push(G2Affine::from(participant.public_key));
        }

        let g2_precomp: Vec<_> = g2_points.iter().map(|p| p.precompute()).collect();
        let miller_result = glued_miller_loop(&g2_precomp, &g1_points);
        miller_result.final_exponentiation() == Gt::identity()
    }

    fn aggregate(&self, partial_signatures: &HashMap<usize, G1Projective>) ->
G1Projective {
        assert!(
            partial_signatures.len() ≥ self.t,
            "Not enough partial signatures"
        );

        let mut aggregated_signature = G1Projective::default();
        let participants: Vec<usize> = partial_signatures.keys().cloned().collect();

        for (&id, signature) in partial_signatures {
            let lambda = self.lagrange_coefficient(id, &participants);
            aggregated_signature = aggregated_signature + (*signature * Fr::into(
lambda));
```

```
        }

        aggregated_signature
    }

    fn lagrange_coefficient(&self, i: usize, participants: &[usize]) -> Fr {
        let x_i = Fr::from(i as u64);
        participants
            .iter()
            .filter(|&&j| j != i)
            .fold(Fr::one(), |acc, &j| {
                let x_j = Fr::from(j as u64);
                acc * (x_j * (x_j - x_i).inv())
            })
    }

    fn verify(&self, message: &[u8], signature: &G1Projective) -> bool {
        let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
        let hashed_message = G1Projective::hash_to_curve(&expander, message)
            .expect("Failed to hash message to curve");

        let g1_points = [G1Affine::from(*signature), G1Affine::from(-hashed_message)
    ];
        let g2_points = [
            G2Affine::from(G2Projective::generator()),
            G2Affine::from(self.group_public_key),
        ];

        let g2_precomp: Vec<_> = g2_points.iter().map(|p| p.precompute()).collect();
        let miller_result = glued_miller_loop(&g2_precomp, &g1_points);
        miller_result.final_exponentiation() == Gt::identity()
    }
}
fn run_dkg_round(round_id: u32, config: &DkgConfig) -> G2Projective {
    let dkg = DistributedKeyGeneration::new(config.quorum as usize, config.
    n_participants as usize);

    let message = b"Hello, Sylow!";
    let mut partial_signatures = HashMap::new();
    for i in 1..=dkg.n {
        let signature = dkg.partial_sign(i, message);
        partial_signatures.insert(i, signature);
    }

    let signatures_valid = dkg.batch_verify_partial(message, &partial_signatures);
    if !signatures_valid {
        panic!("Some partial signatures failed verification");
    }

    let aggregated_signature = dkg.aggregate(&partial_signatures);

    let aggregated_valid = dkg.verify(message, &aggregated_signature);
    if !aggregated_valid {
        panic!("Aggregated signature verification failed");
    }

    dkg.group_public_key
```

```
}

fn main() -> Result<(), confy::ConfyError> {

    let config_path = PathBuf::from("./dkg.toml");
    let cfg: DkgConfig = confy::load_path(config_path)?;

    let mut aggregate_public_key = G2Projective::default();

    for round_id in 0..cfg.n_rounds {
        let round_public_key = run_dkg_round(round_id, &cfg);
        aggregate_public_key = aggregate_public_key + round_public_key;
    }
    Ok(())
}
```

## Threshold signatures

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.

**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 $\mathbb{F}_p$. The concepts of $VSS$ and $DKG$ are indispensable to perform this securely between multiple parties.

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

**Verification** 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.

There are a lot of good partial implementations of everything in this document, including seda's barebones of bn254 and the pairing library here. However, they are either abandonware, insecure, or otherwise too inefficient.

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 repositories. 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 extremely 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 `bn_254`, and `zkcrypto/bls12_381` conforms to security standards set out in RFC9380, so they should be good references for those interested.

There are several stages to execute, which are given in a high-level overview below.

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 $s_i \in \mathbb{F}_r$

(b) Commit the private polynomial to $\mathbb{G}_2$ to create the public key polynomial
   i. Define $A : \mathbb{F}_r \to \mathbb{G}_2 : x \to xg_2$, and apply to polynomial, namely $a_i \to A(a_i) = a_i g_2$
   ii. The group public key is the evalution of the public key polynomial at $0 \in \mathbb{F}_r$, namely $a_0 g_2$

3. 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, $\sigma_i = s_i H(m) \in \mathbb{G}_1$
4. Verify the partial signatures against the public polynomial
   (a) Now having the hash on the curve $H(m)$, and the partial signature $\sigma_i$, 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 $e(\sigma_i, g_2) = e(H(m), P(i))$
5. 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 $\sigma$
6. Use same methodology as step 5 to verify the final signature $e(\sigma, g_2) = \prod_i e(H(m), P(i))$

### Generating Field Scalars

```rust
// Defines the r-torsion field for BN254
define_finite_prime_field!(
    Fr,
    FrModStruct,
    FrOutputType,
    U256,
    8,
    BN254_SUBGROUP_MOD_STRING,
    1,
    1
);
let mut rng = rand::thread_rng();
let secret_polynomial: Vec<Fr> = (0..t).map(|_| Fr::rand(&mut rng)).collect();
```

### Generating Partial Private Keys and Cumulative Public Key

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

```rust
fn evaluate_polynomial(coeffs: &[Fr], x: usize) -> Fr {
    let x = Fr::from(x as u64);
    coeffs
        .iter()
        .rev()
        .fold(Fr::zero(), |acc, coeff| acc * x + *coeff)
}
let participants: Vec<Participant> = (1..=n)
    .map(|i| {
        let secret_key = evaluate_polynomial(&secret_polynomial, i);
        let public_key = G2Projective::generator() * Fr::into(secret_key);
        Participant {
            id: i,
            secret_key,
            public_key,
        }
    })
```

```
    .collect();
let group_public_key = G2Projective::generator() * Fr::into(secret_polynomial[0]);
```

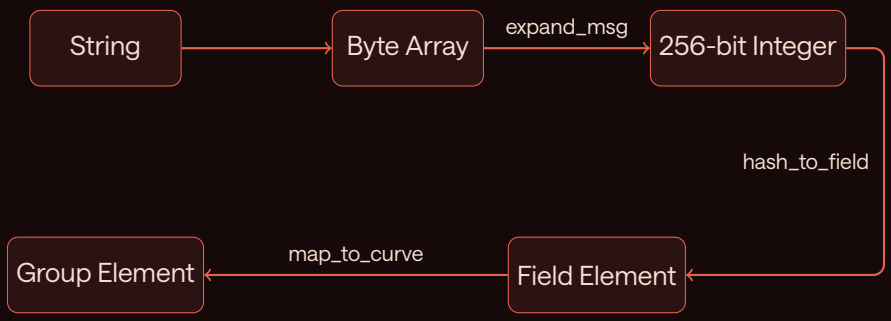
First, we need to commit the scalar polynomial generated to the group to get polynomial on the group, aka multiply each coeffecients by the generator. We call it committing because of the close connection to KZG polynomials in SNARKS (a good blog on it is here). Note that there are faster ways to generate an element in G2, for example this.

### Partial Signing

**Listing 2: evaluate private polynomial at each index** Ok, great. C'est parti à la lune. We now need to partial sign messages. It'd be nice to have something like

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

but what does this actually entail? This is the good stuff, and there are a few steps involved in going from a string to an element of the group. The overall flow is given by the flowchart below:




▷ **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 the `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, a possible implementation of which is here.

**Algorithm 2** Try-and-Increment

**Require:** $n \in \mathbb{Z}$ with $|n|_2 = k$ and $s \in \{0,1\}^*$
**procedure** Try-and-Increment($n$, $k$, $s$)
  $c \leftarrow 0$
  **repeat**
    $s' \leftarrow s \,\|\, c_{bits}()$
    $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \cdots + H(s')_k \cdot 2^k$
    $c \leftarrow c + 1$
  **until** $z < n$
  **return** $z$
**end procedure**
**Ensure:** $z \in \mathbb{Z}_n$

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:

```rust
impl<D: Default + FixedOutput + BlockSizeUser> Expander for XMDExpander<D> {
    fn expand_message(&self, msg: &[u8], len_in_bytes: usize) -> Result<Vec<u8>, HashError> {
        let b_in_bytes = D::output_size();
        let r_in_bytes = D::block_size();
        let ell = (len_in_bytes + b_in_bytes - 1) / b_in_bytes;
        let dst_prime = [
            self.dst_prime.as_slice(),
            &i2osp(self.dst_prime.len() as u64, 1)?,
        ]
        .concat();
        if 8 * b_in_bytes < 2 * self.security_param as usize
            || ell > 255
            || dst_prime.len() != self.dst_prime.len() + 1
        {
            return Err(HashError::ExpandMessage);
        }

        let z_pad = vec![0; r_in_bytes];
        let l_i_b_str = i2osp(len_in_bytes as u64, 2)?;

        let msg_prime = [&z_pad, msg, &l_i_b_str, &i2osp(0, 1)?, &dst_prime].concat();

        let b_0 = D::default().chain(msg_prime).finalize_fixed().to_vec();
        let mut b_vals = vec![Vec::new(); ell];
        b_vals[0] = D::default()
            .chain(b_0.clone())
            .chain(i2osp(1, 1)?.iter())
            .chain(dst_prime.iter())
            .finalize_fixed()
            .to_vec();

        for i in 1..ell {
            let xored: Vec<u8> = b_0
                .iter()
                .zip(&b_vals[i - 1])
                .map(|(&x, &y)| x ^ y)
                .collect();
            let b_i: Vec<u8> = xored
                .iter()
                .chain(i2osp((i + 1) as u64, 1)?.iter())
                .chain(dst_prime.iter())
                .cloned()
                .collect();
            b_vals[i] = D::default().chain(b_i).finalize_fixed().to_vec();
        }

        Ok(b_vals.into_iter().flatten().take(len_in_bytes).collect())
    }
}
pub trait Expander {
    fn hash_to_field(&self, msg: &[u8], count: usize, size: usize) -> Result<[Fp; 2], HashError> {
```

```
        let len_in_bytes = count * size;

        let exp_msg = self.expand_message(msg, len_in_bytes)?;

        let mut retval = [Fp::ZERO; 2];
        for (i, f) in retval.iter_mut().enumerate() {
            let elm_offset = size * i;
            let tv = &exp_msg[elm_offset..elm_offset + size];

            let mut bs = [0u8; 64];
            bs[16..].copy_from_slice(tv);

            let cast_value = U512::from_be_bytes(bs);
            let modulus = NonZero::<U512>::new(u256_to_u512(&Fp::characteristic())).
    unwrap();

            let scalar = U256::from_words(
                (cast_value % modulus).to_words()[0..4]
                    .try_into()
                    .map_err(|_e: TryFromSliceError| HashError::CastToField)?,
            );
            *f = Fp::new(scalar);
        }
        Ok(retval)
    }
}
```

Now having the message in the field, we need to map it to $\mathbb{G}_1$, aka a pair of $(x, y) \in E(\mathbb{F}_r)$. It seems the nicest way would be the "Simplified Shallue-van de Woestijne" method. Unfortunately, despite there being an existing implementation 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** This encoding algorithm has a few prerequisites of the curve. Namely for the constant $Z \in \mathbb{F}_r$ such that, for $y^2 = g(x) = x^3 + Ax + B$:

1. $g(Z) \neq 0$ in the field
2. $-\frac{3Z^2+4A}{4g(Z)} \neq 0$ in the field
   (a) this quantity must be a square in the field
3. At least one of $g(Z)$ and $g(-Z/2)$ is square in the field

```
fn find_z_svdw(a: Fp, b: Fp) -> Fp {
    let g = |x: &Fp| -> Fp { (*x) * (*x) * (*x) + a * (*x) + b };
    let h = |x: &Fp| -> Fp { -(Fp::THREE * (*x) * (*x) + Fp::FOUR * a) / (Fp::FOUR *
    g(x)) };
    let mut ctr = 1;
    loop {
        for z_cand in [Fp::from(ctr), -Fp::from(ctr)] {
            if g(&z_cand).is_zero() {
                continue;
            }
            if h(&z_cand).is_zero() {
                continue;
            }
            if !bool::from(h(&z_cand).is_square()) {
                continue;
            }
            if bool::from(g(&z_cand).is_square())
                | bool::from(g(&(-z_cand / Fp::from(2))).is_square())
```

```
            {
                return z_cand;
            }
        }
        ctr += 1;
    }
}
```

and funnily enough, all of this just shows that for `BN_254`, $Z = 1 \in \mathbb{F}_r$ ...

Using the notation and utility functions from here, I summarise the SvW algorithm for input $u \in \mathbb{F}_r$. It requires a few constants based on the particular curve of interest.

$$c1 = g(Z)$$

$$c2 = -Z/2$$

$$c3 = \sqrt{-g(Z) * (3Z^2 + 4A)}$$

$$c4 = -4g(Z)/(3Z^2 + 4A)$$

Note that the constant $c3$ above *must* be chosen such that `sgn0(c3) = 0`. In other words, if the square-root computation returns a value `cx` such that `sgn0(cx) = 1`, set `c3 = -cx`; otherwise, set `c3 = cx`.

```rust
fn unchecked_map_to_point(&self, u: Fp) -> Result<[Fp; 2], MapError> {

    // Conditional move operation.
    // Selects `x` if `b` is false, `y` if `b` is true.
    let cmov = |x: &Fp, y: &Fp, b: &Choice| -> Fp {
        Fp::from(!bool::from(*b) as u64) * (*x) + Fp::from(bool::from(*b) as u64) * (*y)
    };

    // Step 1: Compute intermediate values
    let tv1 = u * u;
    let tv1 = tv1 * self.c1;
    let tv2 = Fp::from(1) + tv1;
    let tv1 = Fp::from(1) - tv1;
    let tv3 = tv1 * tv2;
    let tv3 = tv3.inv();
    let tv4 = u * tv1;
    let tv4 = tv4 * tv3;
    let tv4 = tv4 * self.c3;

    // Step 2: Compute potential x-coordinates
    let x1 = self.c2 - tv4;

    // Step 3: Evaluate curve equation at potential x-coordinates
    let gx1 = x1 * x1;
    let gx1 = gx1 + self.a;
    let gx1 = gx1 * x1;
    let gx1 = gx1 + self.b;

    // Step 4: Determine which x-coordinate to use
    // and so forth for x 1..3
    let e1 = gx1.is_square();

```

```
    let x2 = self.c2 + tv4;
    let gx2 = x2 * x2;
    let gx2 = gx2 + self.a;
    let gx2 = gx2 * x2;
    let gx2 = gx2 + self.b;
    let e2 = gx2.is_square() & !e1; // Avoid short-circuit logic ops

    let x3 = tv2 * tv2;
    let x3 = x3 * tv3;
    let x3 = x3 * x3;
    let x3 = x3 * self.c4;
    let x3 = x3 + self.z;

    let x = cmov(&x3, &x1, &e1); // x = x1 if gx1 is square, else x = x3;
    let x = cmov(&x, &x2, &e2); // x = x2 if gx2 is square and gx1 is not;
    let gx = x * x;
    let gx = gx + self.a;
    let gx = gx * x;
    let gx = gx + self.b;

    // Step 5: Compute y-coordinate
    let y = match gx.sqrt().into_option() {
        Some(d) => d,
        _ => return Err(MapError::SvdWError),
    };

    // Step 6: Ensure correct sign of y
    let e3 = Choice::from((bool::from(u.sgn0()) == bool::from(y.sgn0())) as u8);
    let y = cmov(&(-y), &y, &e3); // Select correct sign of y;
    Ok([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
   - `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!
   - `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
   - $\sigma_i : \mathbb{G}_1 \to \mathbb{G}_1; H(m) \to s_i H(m)$ 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:

```
fn partial_sign(&self, participant_id: usize, message: &[u8]) -> G1Projective {
    let participant = self
        .participants
        .iter()
        .find(|p| p.id == participant_id)
        .expect("Participant not found");

    let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
    let hashed_message = G1Projective::hash_to_curve(&expander, message)
        .expect("Failed to hash message to curve");
    hashed_message * Fr::into(participant.secret_key)
}
```

### Partial Verification

**Listing 7: partial signaturing** This is pretty straightforward up to deciding how to implement the pairing function.... which is ... easy ... right? Wrong. See 'Field extentions' for the cluster that is pairing maths.

```rust
fn batch_verify_partial(
    &self,
    message: &[u8],
    signatures: &HashMap<usize, G1Projective>,
) -> bool {
    let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
    let hashed_message = G1Projective::hash_to_curve(&expander, message)
        .expect("Failed to hash message to curve");

    let g2_gen = G2Affine::from(G2Projective::generator());
    let mut g1_points = Vec::new();
    let mut g2_points = Vec::new();

    for (&id, signature) in signatures {
        let participant = self
            .participants
            .iter()
            .find(|p| p.id == id)
            .expect("Participant not found");

        g1_points.push(G1Affine::from(*signature));
        g1_points.push(G1Affine::from(-hashed_message));
        g2_points.push(g2_gen);
        g2_points.push(G2Affine::from(participant.public_key));
    }

    let g2_precomp: Vec<_> = g2_points.iter().map(|p| p.precompute()).collect();
    let miller_result = glued_miller_loop(&g2_precomp, &g1_points);
    miller_result.final_exponentiation() == Gt::identity()
}
```

### Signature Aggregation

First, get Lagrange coefficients $\lambda_i$ to recombine the partial signatures

```rust
fn lagrange_coefficient(&self, i: usize, participants: &[usize]) -> Fr {
    let x_i = Fr::from(i as u64);
    participants
        .iter()
        .filter(|&&j| j != i)
        .fold(Fr::one(), |acc, &j| {
            let x_j = Fr::from(j as u64);
            acc * (x_j * (x_j - x_i).inv())
        })
}
```

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

```rust
fn aggregate(&self, partial_signatures: &HashMap<usize, G1Projective>) ->
   G1Projective {
    assert!(
        partial_signatures.len() ≥ self.t,
```

```
        "Not enough partial signatures"
    );

    let mut aggregated_signature = G1Projective::default();
    let participants: Vec<usize> = partial_signatures.keys().cloned().collect();

    for (&id, signature) in partial_signatures {
        let lambda = self.lagrange_coefficient(id, &participants);
        aggregated_signature = aggregated_signature + (*signature * Fr::into(lambda)
    );
    }

    aggregated_signature
}
```

### Final Verification

We now verify the aggregated signatures:

```
fn verify(&self, message: &[u8], signature: &G1Projective) -> bool {
    let expander = XMDExpander::<Keccak256>::new(DST, SECURITY_BITS);
    let hashed_message = G1Projective::hash_to_curve(&expander, message)
        .expect("Failed to hash message to curve");

    let g1_points = [G1Affine::from(*signature), G1Affine::from(-hashed_message)];
    let g2_points = [
        G2Affine::from(G2Projective::generator()),
        G2Affine::from(self.group_public_key),
    ];

    let g2_precomp: Vec<_> = g2_points.iter().map(|p| p.precompute()).collect();
    let miller_result = glued_miller_loop(&g2_precomp, &g1_points);
    miller_result.final_exponentiation() == Gt::identity()
}
```

## Ethereum precompiles

Because of the flexibility provided by `Sylow`, it is also convenient to replace the current implementations of curve arithmetic on `BN_254` used in various instantiations of the Ethereum virtual machine, like `revm`. The fully functional example below shows how to integrate `Sylow` into `revm`, using the module at `crates/precompile/src/bn128.rs`:

```
use crate::{
    utilities::{bool_to_bytes32, right_pad},
    Address, Error, Precompile, PrecompileResult, PrecompileWithAddress,
};
use sylow::{G1Affine, Fp, Fp2, Fr, Gt, G1Projective, G2Projective, glued_pairing};
use revm_primitives::PrecompileOutput;
use std::vec::Vec;
use num_traits::{One, identities::Zero};

pub mod add {
    use super::*;

    const ADDRESS: Address = crate::u64_to_address(6);

    pub const ISTANBUL_ADD_GAS_COST: u64 = 150;
```

```
    pub const ISTANBUL: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
        Precompile::Standard(|input, gas_limit| run_add(input, ISTANBUL_ADD_GAS_COST, gas_limit)),
    );

    pub const BYZANTIUM_ADD_GAS_COST: u64 = 500;
    pub const BYZANTIUM: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
        Precompile::Standard(|input, gas_limit| run_add(input, BYZANTIUM_ADD_GAS_COST, gas_limit)),
    );
}

pub mod mul {
    use super::*;

    const ADDRESS: Address = crate::u64_to_address(7);

    pub const ISTANBUL_MUL_GAS_COST: u64 = 6_000;
    pub const ISTANBUL: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
        Precompile::Standard(|input, gas_limit| run_mul(input, ISTANBUL_MUL_GAS_COST, gas_limit)),
    );

    pub const BYZANTIUM_MUL_GAS_COST: u64 = 40_000;
    pub const BYZANTIUM: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
        Precompile::Standard(|input, gas_limit| run_mul(input, BYZANTIUM_MUL_GAS_COST, gas_limit)),
    );
}

pub mod pair {
    use super::*;

    pub const ADDRESS: Address = crate::u64_to_address(8);

    pub const ISTANBUL_PAIR_PER_POINT: u64 = 34_000;
    pub const ISTANBUL_PAIR_BASE: u64 = 45_000;
    pub const ISTANBUL: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
        Precompile::Standard(|input, gas_limit| {
            run_pair(
                input,
                ISTANBUL_PAIR_PER_POINT,
                ISTANBUL_PAIR_BASE,
                gas_limit,
            )
        }),
    );

    pub const BYZANTIUM_PAIR_PER_POINT: u64 = 80_000;
    pub const BYZANTIUM_PAIR_BASE: u64 = 100_000;
    pub const BYZANTIUM: PrecompileWithAddress = PrecompileWithAddress(
        ADDRESS,
```

```
        Precompile::Standard(|input, gas_limit| {
            run_pair(
                input,
                BYZANTIUM_PAIR_PER_POINT,
                BYZANTIUM_PAIR_BASE,
                gas_limit,
            )
        }),
    );
}

pub const ADD_INPUT_LEN: usize = 64 + 64;

pub const MUL_INPUT_LEN: usize = 64 + 32;

pub const PAIR_ELEMENT_LEN: usize = 64 + 128;

#[inline]
pub fn read_fp(input: &[u8]) -> Result<Fp, Error> {
    Fp::from_be_bytes(<&[u8; 32]>::try_from(&input[..32]).unwrap()).into_option().
   ok_or(
    Error::Bn128FieldPointNotAMember)
}

#[inline]
pub fn read_point(input: &[u8]) -> Result<G1Projective, Error> {
    let px = read_fp(&input[0..32])?;
    let py = read_fp(&input[32..64])?;
    new_g1_point(px, py)
}

pub fn new_g1_point(px: Fp, py: Fp) -> Result<G1Projective, Error> {
    if px == Fp::zero() && py == Fp::zero() {
        Ok(G1Projective::zero())
    } else {
        G1Affine::new([px, py])
            .map(Into::into)
            .map_err(|_| Error::Bn128AffineGFailedToCreate)
    }
}

pub fn run_add(input: &[u8], gas_cost: u64, gas_limit: u64) -> PrecompileResult {
    if gas_cost > gas_limit {
        return Err(Error::OutOfGas.into());
    }

    let input = right_pad::<ADD_INPUT_LEN>(input);

    let p1 = read_point(&input[..64])?;
    let p2 = read_point(&input[64..])?;
    let output = G1Affine::from(p1 + p2).to_be_bytes_scrubbed();
    Ok(PrecompileOutput::new(gas_cost, output.into()))
}

pub fn run_mul(input: &[u8], gas_cost: u64, gas_limit: u64) -> PrecompileResult {
    if gas_cost > gas_limit {
        return Err(Error::OutOfGas.into());
```

```rust
    }

    let input = right_pad::<MUL_INPUT_LEN>(input);

    let p = read_point(&input[..64])?;

    // `Fr::from_slice` can only fail when the length is not 32.
    let fr = Fp::from(Fr::from_be_bytes(<&[u8; 32]>::try_from(&input[64..96]).unwrap
    ()).unwrap());

    let output = G1Affine::from(p * fr).to_be_bytes_scrubbed();
    Ok(PrecompileOutput::new(gas_cost, output.into()))
}

pub fn run_pair(
    input: &[u8],
    pair_per_point_cost: u64,
    pair_base_cost: u64,
    gas_limit: u64,
) -> PrecompileResult {
    let gas_used = (input.len() / PAIR_ELEMENT_LEN) as u64 * pair_per_point_cost +
    pair_base_cost;
    if gas_used > gas_limit {
        return Err(Error::OutOfGas.into());
    }

    if input.len() % PAIR_ELEMENT_LEN != 0 {
        return Err(Error::Bn128PairLength.into());
    }

    let success = if input.is_empty() {
        true
    } else {
        let elements = input.len() / PAIR_ELEMENT_LEN;

        let mut points = Vec::with_capacity(elements);

        // read points
        for idx in 0..elements {
            let read_fp_at = |n: usize| {
                debug_assert!(n < PAIR_ELEMENT_LEN / 32);
                let start = idx * PAIR_ELEMENT_LEN + n * 32;
                // SAFETY: We're reading `6 * 32 == PAIR_ELEMENT_LEN` bytes from `
    input[idx..]`
                // per iteration. This is guaranteed to be in-bounds.
                let slice = unsafe { input.get_unchecked(start..start + 32) };
                Fp::from_be_bytes(<&[u8; 32]>::try_from(slice).unwrap()).into_option
    ().ok_or(
                Error::Bn128FieldPointNotAMember)
            };
            let ax = read_fp_at(0)?;
            let ay = read_fp_at(1)?;
            let bay = read_fp_at(2)?;
            let bax = read_fp_at(3)?;
            let bby = read_fp_at(4)?;
            let bbx = read_fp_at(5)?;

```

```
            let a = new_g1_point(ax, ay)?;
            let b = {
                let ba = Fp2::new(&[bax, bay]);
                let bb = Fp2::new(&[bbx, bby]);

                if ba.is_zero() && bb.is_zero() {
                    G2Projective::zero()
                } else {
                    G2Projective::new([ba, bb, Fp2::one()]).map_err(|_|
                    Error::Bn128AffineGFailedToCreate)?
                }
            };

            points.push((a, b));
        }
        let (g1s, g2s): (Vec<_>, Vec<_>) = points.into_iter().unzip();
        let mul = glued_pairing(&g1s, &g2s);

        mul == Gt::identity()
    };
    Ok(PrecompileOutput::new(gas_used, bool_to_bytes32(success)))
}

#[cfg(test)]
mod tests {
    use crate::bn128::add::BYZANTIUM_ADD_GAS_COST;
    use crate::bn128::mul::BYZANTIUM_MUL_GAS_COST;
    use crate::bn128::pair::{BYZANTIUM_PAIR_BASE, BYZANTIUM_PAIR_PER_POINT};
    use revm_primitives::{hex, PrecompileErrors};

    use super::*;

    #[test]
    fn test_alt_bn128_add() {
        let input = hex::decode(
            "\
             18b18acfb4c2c30276db5411368e7185b311dd124691610c5d3b74034e093dc9\
             063c909c4720840cb5134cb9f59fa749755796819658d32efc0d288198f37266\
             07c2b7f58a84bd6145f00c9c2bc0bb1a187f20ff2c92963a88019e7c6a014eed\
             06614e20c147e940f2d70da3f74c9a17df361706a4485c742bd6788478fa17d7",
        )
        .unwrap();
        let expected = hex::decode(
            "\
            2243525c5efd4b9c3d3c45ac0ca3fe4dd85e830a4ce6b65fa1eeaee202839703\
            301d1d33be6da8e509df21cc35964723180eed7532537db9ae5e7d48f195c915",
        )
        .unwrap();

        let outcome = run_add(&input, BYZANTIUM_ADD_GAS_COST, 500).unwrap();
        assert_eq!(outcome.bytes, expected);

        // out of gas test
        let input = hex::decode(
            "\
            0000000000000000000000000000000000000000000000000000000000000000\
            0000000000000000000000000000000000000000000000000000000000000000\
```

```
            0000000000000000000000000000000000000000000000000000000000000000\
            0000000000000000000000000000000000000000000000000000000000000000",
        )
        .unwrap();

        let res = run_add(&input, BYZANTIUM_ADD_GAS_COST, 499);
        println!("{:?}", res);
        assert!(matches!(res, Err(PrecompileErrors::Error(Error::OutOfGas))));

        // point not on curve fail
        let input = hex::decode(
            "\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111",
        )
        .unwrap();

        let res = run_add(&input, BYZANTIUM_ADD_GAS_COST, 500);
        assert!(matches!(
            res,
            Err(PrecompileErrors::Error(Error::Bn128AffineGFailedToCreate))
        ));
    }

    #[test]
    fn test_alt_bn128_mul() {
        let input = hex::decode(
            "\
            2bd3e6d0f3b142924f5ca7b49ce5b9d54c4703d7ae5648e61d02268b1a0a9fb7\
            21611ce0a6af85915e2f1d70300909ce2e49dfad4a4619c8390cae66cefdb204\
            0000000000000000000000000000000000000000000011138ce750fa15c2",
        )
        .unwrap();
        let expected = hex::decode(
            "\
            070a8d6a982153cae4be29d434e8faef8a47b274a053f5a4ee2a6c9c13c31e5c\
            031b8ce914eba3a9ffb989f9cdd5b0f01943074bf4f0f315690ec3cec6981afc",
        )
        .unwrap();

        let outcome = run_mul(&input, BYZANTIUM_MUL_GAS_COST, 40_000).unwrap();
        assert_eq!(outcome.bytes, expected);

        // out of gas test
        let input = hex::decode(
            "\
            0000000000000000000000000000000000000000000000000000000000000000\
            0000000000000000000000000000000000000000000000000000000000000000\
            0200000000000000000000000000000000000000000000000000000000000000",
        )
        .unwrap();

        let res = run_mul(&input, BYZANTIUM_MUL_GAS_COST, 39_999);
        assert!(matches!(res, Err(PrecompileErrors::Error(Error::OutOfGas))));

```

```
        // point not on curve fail
        let input = hex::decode(
            "\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            0f0000000000000000000000000000000000000000000000000000000000000000",
        )
        .unwrap();

        let res = run_mul(&input, BYZANTIUM_MUL_GAS_COST, 40_000);
        assert!(matches!(
            res,
            Err(PrecompileErrors::Error(Error::Bn128AffineGFailedToCreate))
        ));
    }

    #[test]
    fn test_alt_bn128_pair() {
        let input = hex::decode(
            "\
            1c76476f4def4bb94541d57ebba1193381ffa7aa76ada664dd31c16024c43f59\
            3034dd2920f673e204fee2811c678745fc819b55d3e9d294e45c9b03a76aef41\
            209dd15ebff5d46c4bd888e51a93cf99a7329636c63514396b4a452003a35bf7\
            04bf11ca01483bfa8b34b43561848d28905960114c8ac04049af4b6315a41678\
            2bb8324af6cfc93537a2ad1a445cfd0ca2a71acd7ac41fadbf933c2a51be344d\
            120a2a4cf30c1bf9845f20c6fe39e07ea2cce61f0c9bb048165fe5e4de877550\
            111e129f1cf1097710d41c4ac70fcdfa5ba2023c6ff1cbeac322de49d1b6df7c\
            2032c61a830e3c17286de9462bf242fca2883585b93870a73853face6a6bf411\
            198e9393920d483a7260bfb731fb5d25f1aa493335a9e71297e485b7aef312c2\
            1800deef121f1e76426a00665e5c4479674322d4f75edadd46debd5cd992f6ed\
            090689d0585ff075ec9e99ad690c3395bc4b313370b38ef355acdadcd122975b\
            12c85ea5db8c6deb4aab71808dcb408fe3d1e7690c43d37b4ce6cc0166fa7daa",
        )
        .unwrap();
        let expected =
            hex::decode("
    0000000000000000000000000000000000000000000000000000000000000001")
                .unwrap();

        let outcome = run_pair(
            &input,
            BYZANTIUM_PAIR_PER_POINT,
            BYZANTIUM_PAIR_BASE,
            260_000,
        )
        .unwrap();
        assert_eq!(outcome.bytes, expected);

        // out of gas test
        let input = hex::decode(
            "\
            1c76476f4def4bb94541d57ebba1193381ffa7aa76ada664dd31c16024c43f59\
            3034dd2920f673e204fee2811c678745fc819b55d3e9d294e45c9b03a76aef41\
            209dd15ebff5d46c4bd888e51a93cf99a7329636c63514396b4a452003a35bf7\
            04bf11ca01483bfa8b34b43561848d28905960114c8ac04049af4b6315a41678\
            2bb8324af6cfc93537a2ad1a445cfd0ca2a71acd7ac41fadbf933c2a51be344d\
            120a2a4cf30c1bf9845f20c6fe39e07ea2cce61f0c9bb048165fe5e4de877550\
```

```
            111e129f1cf1097710d41c4ac70fcdfa5ba2023c6ff1cbeac322de49d1b6df7c\
            2032c61a830e3c17286de9462bf242fca2883585b93870a73853face6a6bf411\
            198e9393920d483a7260bfb731fb5d25f1aa493335a9e71297e485b7aef312c2\
            1800deef121f1e76426a00665e5c4479674322d4f75edadd46debd5cd992f6ed\
            090689d0585ff075ec9e99ad690c3395bc4b313370b38ef355acdadcd122975b\
            12c85ea5db8c6deb4aab71808dcb408fe3d1e7690c43d37b4ce6cc0166fa7daa",
        )
        .unwrap();

        let res = run_pair(
            &input,
            BYZANTIUM_PAIR_PER_POINT,
            BYZANTIUM_PAIR_BASE,
            259_999,
        );
        assert!(matches!(res, Err(PrecompileErrors::Error(Error::OutOfGas))));

        // no input test
        let input = [0u8; 0];
        let expected =
            hex::decode("
    0000000000000000000000000000000000000000000000000000000000000001")
                .unwrap();

        let outcome = run_pair(
            &input,
            BYZANTIUM_PAIR_PER_POINT,
            BYZANTIUM_PAIR_BASE,
            260_000,
        )
        .unwrap();
        assert_eq!(outcome.bytes, expected);

        // point not on curve fail
        let input = hex::decode(
            "\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111",
        )
        .unwrap();

        let res = run_pair(
            &input,
            BYZANTIUM_PAIR_PER_POINT,
            BYZANTIUM_PAIR_BASE,
            260_000,
        );
        assert!(matches!(
            res,
            Err(PrecompileErrors::Error(Error::Bn128AffineGFailedToCreate))
        ));

        // invalid input length
```

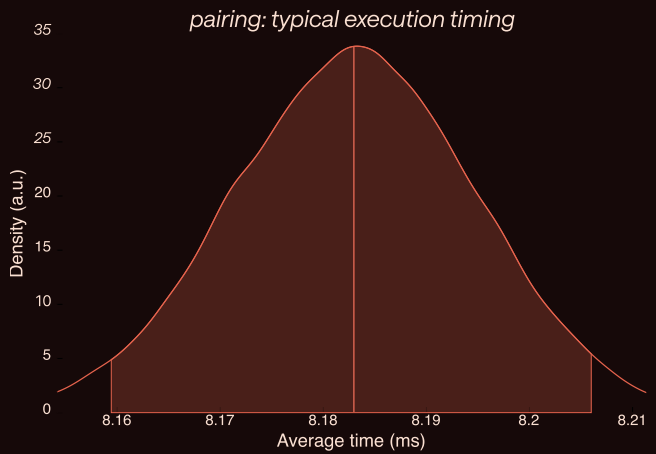
```
        let input = hex::decode(
            "\
            1111111111111111111111111111111111111111111111111111111111111111\
            1111111111111111111111111111111111111111111111111111111111111111\
            11111111111111111111111111111111\
        ",
        )
        .unwrap();

        let res = run_pair(
            &input,
            BYZANTIUM_PAIR_PER_POINT,
            BYZANTIUM_PAIR_BASE,
            260_000,
        );
        assert!(matches!(
            res,
            Err(PrecompileErrors::Error(Error::Bn128PairLength))
        ));
    }
}
```

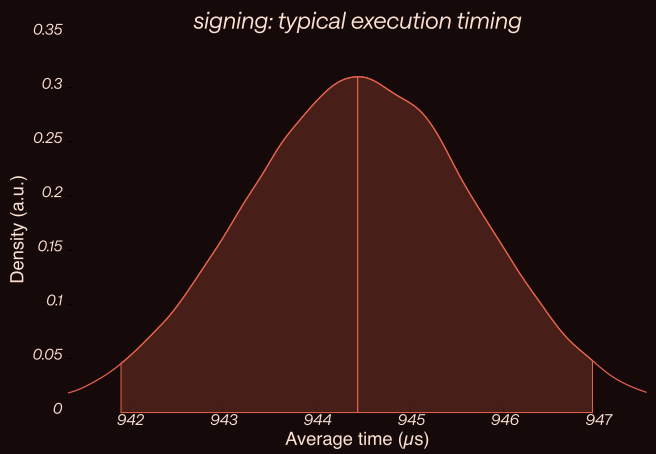
## Performance, Benchmarks & Scaling

`Sylow` *could* have been written with the ultimate goal of absolutely minimizing the execution time of a pairing operation, for instance. However, as is often the case, there is a trade-off to be made regarding speed of the implementation and the ultimate cryptographic security of the crate. WARLOCK has designed `Sylow` using the quickest algorithms available for an elliptic curve suite on j-invariant `0` curves such that cryptographic safeguards, such as constant-time execution, and resistance to timing side channel attacks, calculus attacks, invalid curve attacks, small subgroup attacks, etc., are still enforced.

The timing distributions of signature generation and optimal ate pairing execution are given below for a sample size of 100 executions based on the state of the library at the time of writing (September 2024). These will be updated in future releases on this document, and additional metrics, specifically regarding the constant-time-ness of the library, will be added after `Sylow` undergoes a formal security audit.




The timing distribution of the pairing operation on `BN_254`, yielding an average runtime of 8.183ms ± 120µs.




The timing distribution of signature generation from an arbitrary byte array on `BN_254`, yielding an average runtime of 954 ± 7µs.

# Conclusion

The journey from fundamental concepts in abstract algebra, bilinear pairings, distributed key generation, and others, has hopefully showcased the remarkable cooperation between pure mathematics, modern cryptographic protocol designs, and high-performance computer science.

`Sylow` 's implementation of the `BN_254` curve, while currently optimized for Ethereum's precompile target (and thereby our use cases for verifiable on-chain data feeds about the outside world), represents a crucial step towards secure and efficient decentralized ecosystems. We envisage a few routes of improvement for the library to assist our goal of improving the modern cryptography landscape.

**Multi-curve support** The ability to rapidly support other curves has been baked into the development process of `Sylow` wherever possible, allowing in the future for additional library features supporting predominantly `BLS_12_381`.

**ZK proofs** The efficient and secure pairing operations in `Sylow` now allow for the development of zk-SNARK protocols, such as `Groth16`, and potentially then a prover for `Bellman`, allowing for the usage of `Sylow` in privacy-preserving computations and potentially raising the bar for data feeds from verifiable to provable.

**Cross-platform optimization** Future releases of `Sylow` will likely achieve performance gains based on the target hardware, which can be accomplished via both the underlying modular arithmetic, as well as the pairing operations on `BN_254`, targeting both conventional architectures and embedded systems.

As our dark forest ecosystems evolve, so too will `Sylow` shine a light upon even the darkest corners.

Verily,

WARLOCK

# WARLOCK