## Project objectives

To provide secure and efficient hardware acceleration solutions for Zero-Knowledge Proofs using Field-Programmable Gate Arrays, improving the privacy and security of digital transactions and verifiable computation. Additionally, we aim to explore the potential of formal verification of ZKP protocols for auditing purposes, to provide greater transparency and accountability in financial transactions, and other applications where trust and accountability are critical. By combining hardware acceleration with formal verification, we aim to create a powerful and reliable toolset for enhancing privacy, security, and trust in digital transactions.

## The Proof Systems

ZKPs are an important cryptographic tool that enables one party to prove to another that a certain statement is true, without revealing any additional information beyond the statement's truth. However, ZKP computations are computationally intensive and can be a bottleneck in applications that rely on them.

There are different types of zk proof systems, each with its own strengths and weaknesses. Three of the most well-known proof systems are:

- Succinct Non-Interactive Argument of Knowledge (zk-SNARK)
- Scalable Transparent Argument of Knowledge (zk-STARK)
- Succinct Non-interactive Oecumenical (Universal) aRguments of Knowledge (zk-SNORK)

**Note**

As our overall goal is to not just accelerate the MSM but the whole prover,we need to decide which of the mentioned proof systems are the best for the job.

Currently we are focusing on PLONK which was originally proposed in 2019, many extensions and variations have been developed. The family of Plonk-based proof systems has gained tremendous adoption, and is being used widely.

## The phases of (PLONK) proof generation

At a high level, proof generation consists of three phases:

1. Phase: **Write out the witness**
   - The witness (also sometimes known as the "trace") refers to some data that shows why a statement is true.
2. Phase: **Commit to the witness**
   - Committing to the witness involves outputting some succinct representation of the witness, and in this sense compressing the witness.
   - Using a polynomial commitment scheme for this step allows us to prove certain properties about the original witness referencing just the succinct commitment.
3. Phase: **Prove that the witness is correct**
   - The witness generated in phase 1 must obey certain properties to be valid.
   - A short proof that the original witness satisfies these properties can be generated. Verification of the proof does not require access to the original witness table - verification can be performed referencing only the succinct commitment generated in phase 2.

## Cost summary of (PLONK) proof generation

1. Phase: cost summary
   - Enter witness data into trace table
     - Iterate over and fill in all witness cells in trace table
     - Computing witness values requires large finite field arithmetic
   - Generate auxiliary columns for wiring and lookup constraints
     - Requires additional large finite field arithmetic (as well as sorting, in the case of lookups)
2. Phase: cost summary
   - Commit to each column (real and auxiliary) of the trace table
     - For each column with length $n$, its KZG commitment can be computed via a size $n$ MSM
3. Phase: cost summary
   - Compute the quotient polynomial in evaluation form
     - Convert each column polynomial to coefficient form via size $n$ iFFT
     - Convert each column polynomial to expanded evaluation form via size $2n$ FFT
     - Evaluate the quotient polynomial at each of the $2n$ points, using the evaluation form of each column polynomial
   - Commit to the quotient polynomial
     - Convert to coefficient form via size $2n$ iFFT, in order to split
     - Commit to each split polynomial, requiring a total of 2 size $n$ MSMs
   - Generate evaluation proofs for each polynomial evaluated at random $\alpha$
     - Each column polynomial requires a size $n$ MSM
     - The (split) quotient polynomial requires 2 size $n$ MSMs

## How to accelerate proof systems

As it can be observed, Multi-Scalar Multiplication (MSM) accounts for a significant amount of time in proof generation, and it serves as a fundamental building block in many proof generation schemes. In the subsequent chapter, we will provide a comprehensive description of MSM. However, currently, there are four primary research areas that focus on accelerating proof generation.

1. **Hardware acceleration for heavy computations** - We've seen that heavy computations such as MSM, FFT, and iFFT make up a large chunk of the total computation required for proof generation. These algorithms tend to run quite slowly on CPUs, and can be accelerated greatly by running on GPUs, FPGAs, or ASICs. Investigating how best to run these algorithms on specialized hardware is a very active area of research.

   **Note**

   Our focus is mainly here right now. But we don't want to limit ourselfs only on this step. There is a lot of space for reasearch in other three categories.

2. **Reduce the number of rows in the trace table** - We've also seen that virtually all computations involved in proof generation scale with $n$, the number of rows in the trace table (also referred to as the "number of gates," when the trace table is interpreted as a circuit). Figuring out how to represent certain complex computations while using the fewest number of rows is an area of research with great efficiency implications.

3. **Parallelize and pipeline** - Many proof systems, including the one we studied here, have natural opportunities for parallelization. For example, within the column commitment step of phase 2, each column's commitment can be computed in parallel. Going a step further, each witness column's commitment MSM can be computed concurrently with its generation. Parallelizing and pipelining computations can significantly speed up the overall process.

4. **Alternative proof systems** - This chapter covered the computational requirements of a single particular proof system (PLONK). This proof system is just one of many - there exists a very large design space of theoretical proof systems, with each proof system having its own set of computational requirements and tradeoffs. Research is actively ongoing to further explore this design space, and to design theoretical constructions that reduce or eliminate computational bottlenecks.

# MSM

## Motivation

Multi-Scalar Multiplication (MSM) is a fundamental computational problem. Interest in this problem was recently prompted by its application to ZK-SNARKs, where it often turns out to be the main computational bottleneck. In the table bellow, we can see some of the most popular protocols spend a significant fraction of time computing MSMs. While the exact percentage of time can vary depending on the implementation and circuit size, in general,

| **Protocol** | **No. of MSM (Prover)** | **Prover's time %** |
|---|---|---|
| Groth16 | 4 in $\mathcal{G}_1$, 1 in $\mathcal{G}_2$ | 70-80% |
| Marlin + Lunar | 11 in $\mathcal{G}_1$ | 70-80% |
| Plonk | 9 in $\mathcal{G}_1$ | 85-90 % |

MSM is the main computational bottleneck for ZK-SNARK-based proof systems.

## Definition

Let $\mathcal{G}$ be an elliptic curve group of prime order $p$. Let $G = [G_0, G_1, \ldots, G_{N-1}] \in \mathcal{G}^N$ and $x = [x_0, x_1, \ldots, x_{N-1}] \in \mathbb{F}_p^N$ be $N$-element vectors of elliptic curve points and scalars, respectively. MSM is a problem of computing

$$MSM(x, G) = \sum_{n=0}^{N-1} x_n \cdot G_n$$

Note that plus signs here refer to the *elliptic curve addition*. The scalar multiplication $x_n \cdot G_n$ refers to adding $G_n$ to itself $x_n$ times.

The state-of-the-art, asymptotically optimal algorithm to solve the MSM problem is known as the Pippenger algorithm, and its variant that is widely used in the ZK space is called the bucket method.

## The bucket method

Let us partition each $x_n$ from the equation above into $K$ parts such that each partition consists of $c$ bits. Denote by $b = \lceil \log_2 p \rceil$the maximum number of bits in $x_n$, then $K = \lceil \frac{b}{c} \rceil$. Denoting by $x_n^{[k]}$ the $k$-th partition of $x_n$, equation above can be rewritten as

$$G = \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} 2^{kc} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} \sum_{n=0}^{N-1} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} G^{[k]}$$

where $G^{[k]} = \sum_{n=0}^{N-1} x_n^{[k]} G_n$ is the result of computing multi-scalar-multiplication with $c$-bit scalars $\{x_n^{[k]}\}$. Clearly, if we computed $G^{[k]}$ for each $k$, then the last step can be completed using *Horner's rule* as

$$G = 2^c(\ldots(2^c(2^c G^{[k-1]} + G^{[k-2]}) + G^{[k-3]})\ldots) + G^0$$

## Acceleration

We can break the acceleration of MSM into three parts as follows:

1. Efficient modular multiplications
2. Elliptic curve arithmetic
3. Multi-Scalar Multiplication

## I. Efficient modular multiplication

We start with modular multiplication, a fundamental primitive for all finite field arithmetic. The standard modular multiplication problem in $\mathbb{F}_q$ is formulated as

$$r = a \cdot b \bmod q$$

where $a, b, r \in \mathbb{F}_q$, $q$ is a prime. There exist efficient modular reduction methods in the literature: Barrett reduction, Montgomery reduction, and lookup table-based methods. The challenge here is to find the most efficient, hardware-friendly method for computing the above equation.

## II. Elliptic curve arithmetic

Several coordinate systems have been developed to improve efficiency of operations on elliptic curves. We can refer to the Explicit-Formulas Database for a complete list.

As an example here we can refer to the paper from our competition JumpCrypto. They converted the underlying curve to the twisted edwards representation (as these representation yields a smaller number of modular operations to compute the point addition/doubling). Then they used extended affine coordinates $(x, y, u)$ with $u = kxy$ and extended projective $(X : Y : Z : T)$, where $x = X/Z$, $y = Y/Z$, $T = XY/Z$. This configuration will bring the following results:

- Mixed Addition takes 7 modular multiplication (addition where the second operand is in extended affine coordinates)
- Addition/Doubling takes 9 modular multiplication (unified formulas)

**Note**

There might be room for improving the overall modular operations count by using a different point representation. Right now, most of the solutions are using the combination of affine and projective representation in order to use the Mixed Addition operation. Also a the most solutions are using twisted edwards curves.
There are some strategies to improve memory usage and operation count through the use of point compression, e.g., $(x, y) = x + y$. This could be a neat area of research. There are many unexplored compositions of higher degree, for example $(x, y) = x \cdot x + y \cdot y$. These systems can be evidently transformed into projective representation to mitigate the use of inverses, and more usually mixed systems to achieve reductions in the operation count by further compressing to a common $Z$.
Also we can try to improve the situation on different curves as most of the competition are currently using BLS12-377 curve as it was given in the ZPrize competition.

*Usefull resources:*

- CycloneMSM (JumpCrypto - direct competitor):https://eprint.iacr.org/2022/1396.pdf
- Hardcaml (JaneStreet - direct competitor):https://zprize.hardcaml.com/msm-overview
- PipeMSM (Ingonyama - direct competitor):https://eprint.iacr.org/2022/999
- Arkworks is a Rust library for zkSNARK (Software Rust MSM implementation): https://github.com/arkworks-rs
- Twisted edwards arithmetic in more detail: https://rb.gy/d4jdpk
- Compression techniques applied to binary Edwards curves (the main are of research right now is to find out if something similar is possible in our case, i.e., on ZK friendly curves): https://eprint.iacr.org/2008/171, https://link.springer.com/chapter/10.1007/978-3-319-26617-6_19

## III. Multi-Scalar Multiplication

The challenges here are algorithmic optimisations of the MSM solver. Basically we want to improve the general bucket method presented above to make a best possible version for our SW/HW implementation.