Master's Thesis - Light Node Catchup Using Incrementally Verifiable Chain of Signatures*

Supervisor: Diego Aranha, **Co-supervisor:** Hamidreza Khoshakhlagh

Rasmus Kirk Jakobsen - 201907084

2025-11-07 - 13:24:36 UTC

Abstract

This thesis implements and evaluates Incrementally Verifiable Computation (IVC) using the Plonk proof system, inspired by the Halo framework. We build on prior work on polynomial commitment and accumulation schemes to create an IVC-friendly Plonk Protocol, supporting generic elliptic curves over a cycle of curves, and develop an arithmetization layer with custom gates for elliptic curve operations, Boolean logic, and Poseidon-based hashing.

Benchmarks show that the prover runs in ~300 seconds (parallel) and the verifier runs in ~3 seconds, with proof sizes around 10 kB. While naive signature verification remains faster in the short term, IVC proofs become more efficient in size after only 87 days, and further optimizations could reduce costs. These results suggest that IVC is close to practical viability for blockchain applications, and that optimized frameworks such as Kimchi or Halo2 are well-positioned for real-world deployment.

Contents

*We would like to express our gratitude to Hamidreza Khoshakhlagh and Diego Aranha for their generous advice.

1 Introduction

Valiant originally described IVC - Incrementally Verifiable Computation - in his 2008 paper[Valiant 2008]:

> *Suppose humanity needs to conduct a very long computation which will span superpolynomially many generations. Each generation runs the computation until their deaths when they pass on the computational configuration to the next generation. This computation is so important that they also pass on a proof that the current configuration is correct, for fear that the following generations, without such a guarantee, might abandon the project. Can this be done?*

If a computation runs for hundreds of years and ultimately outputs 42, how can we check its correctness without re-executing the entire process? In order to do this, the verification of the final output of the computation must be much smaller than simply running the computation again. Valiant creates the concept of IVC and argues that it can be used to achieve the above goal.

Recently, IVC has seen renewed interest with cryptocurrencies, as this concept lends itself well to the structure of blockchains. It allows a blockchain node to omit all previous transaction history in favour of only a single state, for example, containing all current account balances. Each state-transition, where transactions are processed, can then be verified with a *SNARK*, Succinct Non-interactive Argument of Knowledge, very small proofs of large statements. This type of blockchain is commonly called a *succinct blockchain*.

A softer approach is to use IVC to prove that the current block corresponds to the most recent block in a chain originating from the genesis block. This allows for near-instant blockchain catch-up for light nodes. A light node is a blockchain node with lower security standards than a full node, but it allows the node to dedicate fewer computational resources and hard drive space. This light node still need to catch-up to the current block in the blockchain, which involves downloading and verifying all previous blocks. This yields less security than in a succinct blockchain, but has the advantage of being much simpler than proving the validity of a the transactions in a block in-circuit. It also requires minimal changes to an existing blockchain.

IVC has notably been used by the Mina[2025] succinct blockchain blockchain. This is enabled by increasingly efficient recursive proof systems, one of the most used in practice is based on Halo[Bowe et al. 2019], which includes Halo2 by the Electric Coin Company (to be used in Zcash) and Kimchi developed and used by Mina. Both can be broken down into the following main components:

- **Plonk**: A general-purpose, potentially zero-knowledge, SNARK.
- $\mathbf{PC_{DL}}$: A Polynomial Commitment Scheme in the Discrete Log setting.
- $\mathbf{AS_{DL}}$: An Accumulation Scheme for Evaluation Proof instances in the Discrete Log setting.
- **Pasta**: A cycle of elliptic curves, Pallas and Vesta, collectively known as Pasta.

A previous project[Jakobsen 2025] by one of the authors of this thesis, analyzed and implemented the accumulation and polynomial commitment schemes.

We aim to create a simplified recursive SNARK based on Halo and use it to create a *chain of signatures*. We argue that this chain of signatures can be used in certain modern blockchains to achieve near-instant blockchain catch-up.

In section 3 we define $\mathrm{PC_{DL}}$. In section 4 we define $\mathrm{AS_{DL}}$. In section 5 we define a modified Plonk based on $\mathrm{PC_{DL}}$ and $\mathrm{AS_{DL}}$ with all custom gates needed to achieve an IVC-friendly SNARK. In section 6 define an IVC-circuit for proving the validity of a chain of signatures. In section 7 we formally define the arithmetization pipeline needed for the defined Plonk Scheme. In section 8 we discuss the implementation of the IVC circuit for verifying a chain of signatures, and benchmark the performance. Both this document and the implementation can be found in the project's repository[Jakobsen and Latiff 2025].

2 Prerequisites

Basic knowledge of elliptic curves, groups and interactive arguments is assumed in the following text. Basic familiarity with SNARKs is also assumed.

The following subsections introduce the concept of Incrementally Verifiable Computation (IVC) along with some background concepts. These concepts lead to the introduction of accumulation schemes and polynomial commitment schemes, the main focus of this paper. Accumulation schemes, in particular, will be demonstrated as a means to

create more flexible IVC constructions compared to previous approaches, allowing IVC that does not depend on a trusted setup.

As such, these subsections aim to provide an overview of the evolving field of IVC, the succinct proof systems that lead to its construction, and the role of accumulation schemes as an important cryptographic primitive with practical applications.

2.1 Notation

The following table denotes the meaning of the notation used throughout the document:

$[n]$	Denotes the integers $\{1, ..., n\}$
$a \in \mathbb{F}$	A field element of an unspecified field
$a \in \mathbb{F}_q$	A field element in a prime field of order q
$\boldsymbol{a} \in S_q^n$	A vector of length n consisting of elements from set S
$G \in \mathbb{E}(\mathbb{F}_q)$	An elliptic Curve point, defined over field $\mathbb{F}_q$
$G \in \mathbb{E}_p(\mathbb{F}_q)$	An elliptic Curve point, defined over field $\mathbb{F}_q$, where the curve has order p
$(a_1, \ldots, a_n) = [x_i]^n = [x_i]_{i=1}^n = \boldsymbol{a} \in S_q^n$	A vector of length n
$a \in_R S$	a is a uniformly randomly sampled element of S
$(S_1, \ldots, S_n)$	In the context of sets, the same as $S_1 \times \cdots \times S_n$
$\boldsymbol{a} \mathbin{+\!\!+} \boldsymbol{b}$ where $\boldsymbol{a} \in \mathbb{F}_q^n, \boldsymbol{b} \in \mathbb{F}_q^m$	Concatenate vectors to create $\boldsymbol{c} \in \mathbb{F}_q^{n+m}$.
$a \mathbin{+\!\!+} b$ where $a \in \mathbb{F}_q$	Create vector $\boldsymbol{c} = (a, b)$.
I.K w	"I Know", Used in the context of proof claims, meaning I have knowledge of the witness w
$\mathbb{B}$	A boolean, i.e. $\{\bot, \top\}$
$\textbf{Option}(T)$	Either a value T or $\bot$, i.e. $\{T, \bot\}$
$\textbf{Result}(T, E)$	Either a value T or a value E, i.e. $\{T, E\}$

We use additive notation for the elliptic curve group. Note that the following are isomorphic $\{\top, \bot\} \cong \mathbb{B} \cong \textbf{Option}(\top) \cong \textbf{Result}(\top, \bot)$, but they have different connotations. Generally for this report, $\textbf{Option}(T)$ models optional arguments, where $\bot$ indicates an empty argument and $\textbf{Result}(T, \bot)$ models the result of a computation that may fail, especially used for rejecting verifiers.

2.2 Proof Systems

An Interactive Proof System consists of two Interactive Turing Machines: a computationally unbounded Prover, $\mathcal{P}$, and a polynomial-time bounded Verifier, $\mathcal{V}$. The Prover tries to convince the Verifier of a statement $X \in L$, with language L in NP. The following properties must be true:

- **Completeness:** $\forall \mathcal{P} \in ITM, X \in L \implies \Pr[\mathcal{V}_{out} = \bot] \leq \epsilon(X)$

  For all honest provers, $\mathcal{P}$, where X is true, the probability that the verifier remains unconvinced ($\mathcal{V}_{out} = \bot$) is negligible in the length of X.

- **Soundness:** $\forall \mathcal{P}^* \in ITM, X \notin L \implies \Pr[\mathcal{V}_{out} = \top] \leq \epsilon(X)$

  For all provers, honest or otherwise, $\mathcal{P}^*$, that try to convince the verifier of a claim, X, that is not true, the probability that the verifier will be convinced ($\mathcal{V}_{out} = \top$) is negligible in the length of X.

An Interactive Argument is very similar, but the honest and malicious prover are now polynomially bounded and receive a Private Auxiliary Input, w, not known by $\mathcal{V}$. This is such that $\mathcal{V}$ can't just compute the answer themselves. Definitions follow:

- **Completeness**: $\forall \mathcal{P}(w) \in PPT, X \in L \implies \Pr[\mathcal{V}_{out} = \bot] \leq \epsilon(X)$
- **Soundness**: $\forall \mathcal{P}^* \in PPT, X \notin L \implies \Pr[\mathcal{V}_{out} = \top] \leq \epsilon(X)$

Proofs of knowledge are another type of Proof System, here the prover claims to know a *witness*, w, for a statement X. Let $X \in L$ and $W(X)$ be the set of witnesses for X that should be accepted in the proof. This allows us to define the following relation: $\mathcal{R} = \{(X, w) : X \in L, w \in W(X)\}$

A proof of knowledge for relation $\mathcal{R}$ is a two party protocol $(\mathcal{P}, \mathcal{V})$ with the following two properties:

- **Knowledge Completeness:** $\Pr[\mathcal{P}(w) \iff \mathcal{V}_{out} = \top] = 1$, i.e. as in Interactive Proof Systems, after an interaction between the prover and verifier the verifier should be convinced with certainty.

- **Knowledge Soundness:** Loosely speaking, Knowledge Soundness requires the existence of an efficient extractor $\mathcal{E}$ that, when given a possibly malicious prover $\mathcal{P}^*$ as input, can extract a valid witness with probability at least as high as the probability that $\mathcal{P}^*$ convinces the verifier $\mathcal{V}$.

The above proof systems may be *zero-knowledge*, which in loose terms means that anyone looking at the transcript, that is the interaction between prover and verifier, will not be able to tell the difference between a real transcript and one that is simulated. This ensures that an adversary gains no new information beyond what they could have computed on their own. We now define the property more formally:

- **Zero-knowledge:** $\forall \mathcal{V}^*(\delta).\exists S_{\mathcal{V}^*}(X) \in PPT.S_{\mathcal{V}^*} \sim^C (\mathcal{P}, \mathcal{V}^*)$

$\mathcal{V}^*$ denotes a verifier, honest or otherwise, δ represents information that $\mathcal{V}^*$ may have from previous executions of the protocol and $(\mathcal{P}, \mathcal{V}^*)$ denotes the transcript between the honest prover and (possibly) malicious verifier. There are three kinds of zero-knowledge:

- **Perfect Zero-knowledge:** $\forall \mathcal{V}^*(\delta).\exists S_{\mathcal{V}^*}(X) \in PPT.S_{\mathcal{V}^*} \sim^{\mathcal{P}} (\mathcal{P}, \mathcal{V}^*)$, the transcripts $S_{\mathcal{V}^*}(X)$ and $(\mathcal{P}, \mathcal{V}^*)$ are perfectly indistinguishable.
- **Statistical Zero-knowledge:** $\forall \mathcal{V}^*(\delta).\exists S_{\mathcal{V}^*}(X) \in PPT.S_{\mathcal{V}^*} \sim^S (\mathcal{P}, \mathcal{V}^*)$, the transcripts $S_{\mathcal{V}^*}(X)$ and $(\mathcal{P}, \mathcal{V}^*)$ are statistically indistinguishable.
- **Computational Zero-knowledge:** $\forall \mathcal{V}^*(\delta).\exists S_{\mathcal{V}^*}(X) \in PPT.S_{\mathcal{V}^*} \sim^C (\mathcal{P}, \mathcal{V}^*)$, the transcripts $S_{\mathcal{V}^*}(X)$ and $(\mathcal{P}, \mathcal{V}^*)$ are computationally indistinguishable, i.e. no polynomially bounded adversary $\mathcal{A}$ can distinguish them.

Where two distributions D_1, D_2 are:

- *Perfectly indistinguishable* if they are identical, meaning no observer, even with unbounded power, can tell them apart:
  $\forall x : \Pr[D_1 = x] = \Pr[D_2 = x]$
- *Statistically indistinguishable* if their statistical distance is negligible, meaning that they may differ, but the difference is vanishingly small, even for an unbounded adversary:
  $\forall x : \Delta(D_1, D_2) := \frac{1}{2} \sum_x \left| \Pr[D_1 = x] - \Pr[D_2 = x] \right| \leq \text{negl}(\lambda)$
- *Computationally indistinguishable* if no probabilistic polynomial-time distinguisher $\mathcal{A}$ can tell them apart with more than negligible advantage, though an unbounded adversary might:
  $\forall x : |\Pr[\mathcal{A}(x) \to D_1] - \Pr[\mathcal{A}(x) = D_2]| \leq \text{negl}(\lambda)$

2.3 Fiat-Shamir Heuristic

The Fiat-Shamir heuristic turns a public-coin (an interactive protocol where the verifier only sends uniformly sampled challenge values) interactive proof into a non-interactive proof, by replacing all uniformly random values sent from the verifier to the prover with calls to a non-interactive random oracle. In practice, a cryptographic hash function, ρ, is used. Composing proof systems will sometimes require *domain-separation*, whereby random oracles used by one proof system cannot be accessed by another proof system. In practice one can have a domain specifier, for example $0, 1$, prepended to each message that is hashed using ρ:

$$\rho_0(m) = \rho(0 + m), \quad \rho_1(m) = \rho(1 + m)$$

2.4 Pedersen Commitments

A commitment scheme is a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden to others, with the ability to reveal the committed value later. Commitment schemes are designed so that the committing party cannot change the value after they have committed to it, i.e. it is *binding*. The fact that anyone that receives the commitment cannot compute the value from the it is called *hiding*.

To reveal a value one can simply send the value to a party that previously received the commitment, and the receiving party can compute the commitment themselves and compare to the previously received commitment. One such commitment scheme is the *Pedersen commitment scheme*[Pedersen 1992]:

Algorithm: CM.Commit

Inputs

$\boldsymbol{m} : \mathbb{F}^n$	The vectors we wish to commit to.
pp_{CM}	The public parameters for the commitment scheme.
$\omega : \textbf{Option}(\mathbb{F})$	Optional hiding factor for the commitment.

Output

$C : \mathbb{E}(\mathbb{F}_q)$	The Pedersen commitment.

1: Parse $\boldsymbol{G} : \mathbb{E}(\mathbb{F})^n, S : \mathbb{E}(\mathbb{F})$ from pp_{CM}.
2: Output $C := \langle \boldsymbol{m}, \boldsymbol{G} \rangle + \omega S$.

Notice, that the inputs is a vector of messages, not just a single message. Inclusion of a hiding factor makes the commitment *perfectly hiding*, but *computationally binding*. If the hiding factor is omitted, it is commonly called a *deterministic Pedersen commitment* and the commitment will be perfectly binding and computationally hiding. For $C = \text{CM.Commit}(m, \boldsymbol{G}, \omega)$

- **Perfect Hiding:** Given C, it is impossible to determine m, no matter your computational power.
- **Computational Hiding:** It is computationally infeasible to determine the value committed to, from the commitment
- **Perfect Binding:** It is impossible to change the value committed to, no matter your computational power.
- **Computational Binding:** It is computationally infeasible to change the value committed to.

The corresponding setup algorithm is:

Algorithm: CM.Setup^{ρ}

Inputs

λ	The security parameter, in unary form.
L	The maximum size vector that can be committed to.

Output

pp_{CM}	The public parameters to be used in CM.Commit

1: $(\mathbb{E}(\mathbb{F}_q), q, G) \leftarrow \text{SampleGroup}^{\rho}(1^{\lambda})$
2: Choose independently uniformly-sampled generators in $\mathbb{E}(\mathbb{F}_q)$, $\boldsymbol{G} \in_R \mathbb{E}(\mathbb{F}_q)^L, S \in_R \mathbb{E}(\mathbb{F}_q)$ using ρ_0.
3: Output $\text{pp}_{\text{CM}} = ((\mathbb{E}(\mathbb{F}_q), q, G), \boldsymbol{G}, S)$

Pedersen commitments are an instance of a very useful type of commitment scheme for proof systems is that of a *homomorphic commitment scheme*, where:

$$\text{CM.Commit}(m_1, r_1) + \text{CM.Commit}(m_2, r_2) = \text{CM.Commit}(m_1 + m_2, r_1 + r_2)$$

That is, you can add the commitments which corresponds to adding the committed inputs and then commititing. This lets a verifying party check the properties of committed values without needing to know them. Since the public parameters can be chosen uniformly randomly, this type of setup is *untrusted*.

2.5 Trusted and Untrusted Setups

Many SNARK constructions, such as the original Plonk specification, depend on a *trusted setup* to ensure soundness. A trusted setup generates a *Structured Reference String* (SRS) with a particular internal structure. For Plonk, this arises from the KZG[Kate et al. 2010] commitments used. These commitments allow the SNARK verifier to achieve sub-linear verification time. However, this comes at the cost of requiring a trusted setup, whereas PC_{DL} for example, uses an *untrusted setup*.

An untrusted setup, creates a *Uniform Random String* of the form:

$$\text{URS} = \{a_1 G, a_2 G, \ldots, a_D G\}$$

Where D represents the maximum degree bound of a polynomial (in a PCS context) and G is a generator of $\mathbb{E}(\mathbb{F})$. The URS must consist solely of generators and all the scalars must be uniformly random. PC_{DL} is then sound,

provided that no adversary knows the scalars. Extracting $\boldsymbol{a}$ from the URS would require solving the Discrete Logarithm problem (DL), which is assumed to be hard.

To generate the URS transparently, a collision-resistant hash function $\mathcal{H} : \mathbb{B}^* \to \mathbb{E}(\mathbb{F}_q)$ can be used to produce the generators. The URS can then be derived using a genesis string s:

$$\text{URS} = \{\mathcal{H}(s + 1), \mathcal{H}(s + 2), \ldots, \mathcal{H}(s + D)\}$$

The genesis string can be any arbitrary string, that convinces outsiders that it's not maliciously chosen. This is commonly referred to as nothing-up-my-sleeve numbers. We used the string:

> *To understand recursion, one must first understand recursion.*

Anyone can verify that the URS was generated from this string, and the probability that such a specific string, hashed, would lead to a known discrete log, should be negligible.

2.6 SNARKS

SNARKs - Succinct Non-interactive Arguments of Knowledge - have seen increased usage due to their application in blockchains and cryptocurrencies. They also typically function as general-purpose proof schemes. This means that, given any solution to an NP-problem, the SNARK prover will produce a proof that they know the solution to said NP-problem. Most SNARKs also allow for zero-knowledge arguments, making them zk-SNARKs.

More concretely, imagine that Alice has today's Sudoku problem $X \in \text{NP}$: She claims to have a solution to this problem, her witness, w, and wants to convince Bob without having to reveal the entire solution. She could then use a SNARK to generate a proof for Bob. To do this she must first encode the Sudoku verifier as a circuit R_X, then let x represent public inputs to the circuit, such as today's Sudoku values/positions, etc, and then give the SNARK prover the public inputs and her witness, $\pi = \text{SNARK.}\textsc{Prover}(R_X, x, w)$. Finally she sends this proof, π, to Bob along with the public Sudoku verifying circuit, R_X, and he can check the proof and be convinced using the SNARK verifier ($\text{SNARK.}\textsc{Verifier}(R_X, x, \pi)$).

Importantly, the 'succinct' property means that the proof size and verification time must be sub-linear. This allows SNARKs to be directly used for *Incrementally Verifiable Computation*.

2.7 Bulletproofs

In 2017, the Bulletproofs paper[Bünz et al. 2017] was released[1]. Bulletproofs rely on the hardness of the Discrete Logarithm problem, and uses an untrusted setup. It has logarithmic proof size, linear verification time and lends itself well to efficient range proofs. It's also possible to generate proofs for arbitrary circuits, yielding a zk-NARK. It's a NARK since we lose the succinctness in terms of verification time, making Bulletproofs less efficient than SNARKs.

At the heart of Bulletproofs lies the Inner Product Argument (IPA), wherein a prover demonstrates knowledge of two vectors, $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_q^n$, with commitment $P \in \mathbb{E}(\mathbb{F}_q)$, and their corresponding inner product, $c = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. It creates a non-interactive proof, with only $\lg(n)$ size, by compressing the point and vectors $\lg(n)$ times, halving the size of the vectors each iteration in the proof. Unfortunately, since the IPA, and by extension Bulletproofs, suffer from linear verification time, Bulletproofs are unsuitable for IVC.

2.8 Incrementally Verifiable Computation

In order to achieve IVC, you need a function $F(x) \in S \to S$ along with some initial state $s_0 \in S$. Then you can call $F(x)$ n times to generate a series of s's, $\boldsymbol{s} \in S^{n+1}$:

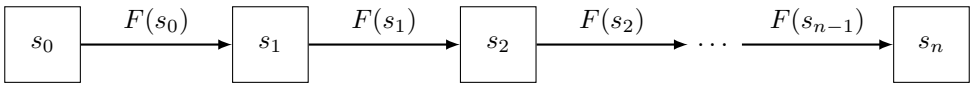


Figure 1: A visualization of the relationship between $F(x)$ and $\boldsymbol{s}$ in a non-IVC setting.

[1] A gentle introduction can be found in "From Zero (Knowledge) to Bulletproofs"[Gibson 2022], which also describes Pedersen commitments and the concept of zero-knowledge.

In a blockchain setting, you might imagine any $s_i \in \boldsymbol{s}$ as a set of accounts with corresponding balances, and the transition function $F(x)$ as the computation happening when a new block is created and therefore a new state, or set of accounts, s_i is computed[2].

In the IVC setting, we have a proof, π, associated with each state, so that anyone can take only a single pair (s_m, π_m) along with the initial state and transition function $(s_0, F(x))$ and verify that said state was computed correctly.

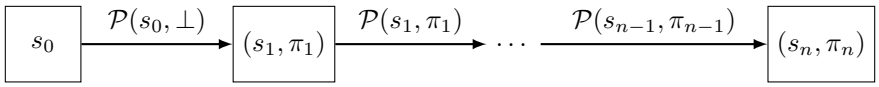


Figure 2: A visualization of the relationship between $F, \boldsymbol{s}$ and $\boldsymbol{\pi}$ in an IVC setting using traditional SNARKs. $\mathcal{P}(s_i, \pi_i)$ denotes running the SNARK.$\textsc{Prover}(R_F, x = \{s_0, s_i\}, w = \{s_{i-1}, \pi_{i-1}\}) = \pi_i$ and $F(s_{i-1}) = s_i$, where R_F is the transition function F expressed as a circuit.

The proof π_i describes the following claim:

> *"The current state s_i is computed from applying the function, F, i times to s_0 $(s_i = F^i(s_0) = F(s_{i-1}))$ and the associated proof π_{i-1} for the previous state is valid."*

Or more formally, π_i is a proof of the following claim, expressed as a circuit R:

$$R := \text{I.K. } w = \{\pi_{i-1}, s_{i-1}\} \text{ s.t. } s_i \stackrel{?}{=} F(s_{i-1}) \ \wedge \ (s_{i-1} \stackrel{?}{=} s_0 \vee \text{SNARK.}\textsc{Verifier}(R_F, x = \{s_0, s_i\}, \pi_{i-1}) \stackrel{?}{=} \top)$$

Where I.K. denotes "I Know". Note that R_F, s_i, s_0 are not quantified above, as they are public values. Each state, s_i, including the genesis state, s_0, must also contain the current iteration, i, for soundness to hold. The SNARK.$\textsc{Verifier}$ represents the verification circuit in the proof system we're using. This means, that we're taking the verifier, representing it as a circuit, and then feeding it to the prover. This is not a trivial task in practice! Note also, that the verification time must be sub-linear to achieve an IVC scheme, otherwise the verifier could just have computed $F^n(s_0)$ themselves, as s_0 and $F(x)$ necessarily must be public.

To see that the above construction works, observe that $\pi_1, \ldots, \pi_n$ proves:

$$\begin{aligned}
&\text{I.K. } \pi_{n-1} \text{ s.t. } && s_n &&= F(s_{n-1}) \ \wedge \ (s_{n-1} = s_0 \vee \text{SNARK.}\textsc{Verifier}(R, x, \pi_{n-1}) = \top),\\
&\text{I.K. } \pi_{n-2} \text{ s.t. } && s_{n-1} &&= F(s_{n-2}) \ \wedge \ (s_{n-2} = s_0 \vee \text{SNARK.}\textsc{Verifier}(R, x, \pi_{n-2}) = \top), \ \ldots\\
& && s_1 &&= F(s_0) \quad \wedge \ (s_0 = s_0 \quad \vee \text{SNARK.}\textsc{Verifier}(R, x, \pi_0) = \top)
\end{aligned}$$

Which means that:

$$\begin{aligned}
&\text{SNARK.}\textsc{Verifier}(R, x, \pi_n) = \top \implies\\
&s_n = F(s_{n-1}) \wedge\\
&\text{SNARK.}\textsc{Verifier}(R, x, \pi_{n-1}) = \top \wedge\\
&s_{n-1} = F(s_{n-2}) \implies \ldots\\
&\text{SNARK.}\textsc{Verifier}(R, x, \pi_1) = \top \implies\\
&s_1 = F(s_0)
\end{aligned}$$

Thus, by induction $s_n = F^n(s_0)$

2.9 The Schwarz-Zippel Lemma

The Schwarz-Zippel lemma is commonly used in succinct proof systems to test polynomial identities. Formally it states:

$$\xi \in_R \mathbb{F} : \Pr[p(\xi) = 0 \mid p(X) \neq 0] = \frac{d}{\mathbb{F}}$$

Meaning that if $p(X)$ is not the zero-polynomial, the evaluation at a uniformly random point from $\mathbb{F}$, will equal zero with at most $d \,/\, \mathbb{F}$ probability. This can also be used to check equality between polynomials:

[2] In the blockchain setting, the transition function would also take an additional input representing new transactions, $F(x : S, T : \mathcal{P}(T))$.

$$\xi \in_R \mathbb{F}$$
$$r(X) = p(X) - q(X)$$
$$r(\xi) \stackrel{?}{=} 0$$

Or equivalently:

$$p(\xi) \stackrel{?}{=} q(\xi)$$

Meaning that $p(X) = q(X)$ with probability at least $d \,/\, \mathbb{F}$.

2.10 Polynomial Interpolation

It is well known that given $d+1$ evaluations, an evaluation domain, $\boldsymbol{p^{(e)}} := [p_1^{(e)}, \ldots, p_{d+1}^{(e)}]$, of a polynomial $p(X)$, you can reconstruct the polynomial using Lagrange interpolation:

$$p(X) = \text{lagrange}(\boldsymbol{p^{(e)}})$$

With a worst-case runtime of $\mathcal{O}(d^2)$. However, if the evaluation points are chosen to be the n-th roots of unity, i.e. the set:

$$\{\omega^1, \omega^2, \ldots, \omega^n\}$$

where ω is a primitive n-th root of unity, then interpolation can be reduced to applying a discrete Fourier transform. We can then choose $n \geq d+1$, and evaluate at all n points of the domain. Setting n to be the next power of 2 above $d+1$ allows us to use the very efficient radix-2 FFT:

$$\boldsymbol{p^{(e)}} := [p_1^{(e)}, \ldots, p_n^{(e)}]$$
$$p(X) = \text{ifft}(\boldsymbol{p^{(e)}})$$
$$\boldsymbol{p^{(e)}} = \text{fft}(p(X))$$

Where both the evaluation domain and polynomial can be computed efficiently using the Fast Fourier Transform in time $\mathcal{O}(n \log n)$. This approach can be used whenever the underlying field contains a primitive n-th root of unity and n is invertible in the field, which is the case for many finite fields used in cryptography including the fields used in this project.

2.11 Polynomial Commitment Schemes

In the SNARK section, general-purpose proof schemes were described. Modern general-purpose (zero-knowledge) proof schemes, such as Sonic[Maller et al. 2019], Plonk[Gabizon et al. 2019] and Marlin[Chiesa et al. 2019], commonly use *Polynomial Commitment Schemes* (PCSs) for creating their proofs. This means that different PCSs can be used to get security under weaker or stronger assumptions.

- **KZG PCSs:** Uses a trusted setup, which involves generating a Structured Reference String for the KZG commitment scheme[Kate et al. 2010]. This would give you a traditional SNARK.
- **Bulletproofs PCSs:** Uses an untrusted setup, assumed secure if the Discrete Log problem is hard, the verifier is linear.
- **FRI PCSs:** Also uses an untrusted setup, assumes secure one way functions exist. It has a higher constant overhead than PCSs based on the Discrete Log assumption, but because it instead assumes that secure one-way functions exist, you end up with a quantum secure PCS.

A PCS allows a prover to prove to a verifier that a committed polynomial evaluates to a certain value, v, given an evaluation input z. There are five main functions used to prove this (PC.Trim omitted as it's unnecessary):

- $\text{PC.Setup}(\lambda, D)^{\rho} \rightarrow \text{pp}_{\text{PC}}$

  The setup routine. Given security parameter λ in unary and a maximum degree bound D. Creates the public parameters pp_{PC}.

- $\text{PC.}\textsc{Commit}(p : \mathbb{F}_q^{d'}[X], d : \mathbb{N}, \omega : \textbf{Option}(\mathbb{F}_q)) \to \mathbb{E}(\mathbb{F}_q)$

  Commits to a degree-d' polynomial p with degree bound d where $d' \leq d$ using optional hiding ω.

- $\text{PC.}\textsc{Open}^\rho(p : \mathbb{F}_q^{d'}[X], C : \mathbb{E}(\mathbb{F}_q), d : \mathbb{N}, z : \mathbb{F}_q, \omega : \textbf{Option}(\mathbb{F}_q)) \to \textbf{EvalProof}$

  Creates a proof, $\pi \in \textbf{EvalProof}$, that the degree d' polynomial p, with commitment C, and degree bound d where $d' \leq d$, evaluated at z gives $v = p(z)$, using the hiding input ω if provided.

- $\text{PC.}\textsc{Check}^\rho(C : \mathbb{E}(\mathbb{F}_q), d : \mathbb{N}, z : \mathbb{F}_q, v : \mathbb{F}_q, \pi : \textbf{EvalProof}) \to \textbf{Result}(\top, \bot)$

  Checks the proof π that claims that the degree d' polynomial p, with commitment C, and degree bound d where $d' \leq d$, evaluates to $v = p(z)$.

Any NP-problem, $X \in NP$, with a witness w can be compiled into a circuit R_X. This circuit can then be fed to a general-purpose proof scheme prover $\mathcal{P}_X$ along with the witness and public input $(x, w) \in X$, that creates a proof of the statement "$R_X(x, w) = \top$". Simplifying slightly, they typically consists of a series of pairs representing opening proofs:

$$(q_1 = (C_1, d, z_1, v_1, \pi_1), \ldots, q_m = (C_m, d, z_m, v_m, \pi_m))$$

These pairs will henceforth be more generally referred to as *instances*, $\boldsymbol{q} \in \textbf{Instance}^m$. They can then be verified using $\text{PC.}\textsc{Check}$:

$$\text{PC.}\textsc{Check}(C_1, d, z_1, v_1, \pi_1) \stackrel{?}{=} \ldots \stackrel{?}{=} \text{PC.}\textsc{Check}(C_m, d, z_m, v_m, \pi_m) \stackrel{?}{=} \top$$

Along with some checks that the structure of the underlying polynomials $\boldsymbol{p}$, that $\boldsymbol{q}$ was created from, satisfies any desired relations associated with the circuit R_X. We can model these relations, or *identities*, using a function $I_X \in \textbf{Instance} \to \{\top, \bot\}$. If,

$$\forall j \in [m] : \text{PC.}\textsc{Check}(C_j, d, z_j, v_j, \pi_j) \stackrel{?}{=} \top \wedge I_X(q_j) \stackrel{?}{=} \top$$

Then the verifier $\mathcal{V}_X$ will be convinced that w is a valid witness for X. In this way, a proof of knowledge of a witness for any NP-problem can be represented as a series of PCS evaluation proofs, including our desired witness that $s_n = F^n(s_0)$.

A PCS has soundness and completeness properties, as well as a binding property:

Completeness: For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$ and publicly agreed upon $d \in \mathbb{N}$:

$$\Pr\left[\begin{array}{c} \deg(p) \leq d \leq D, \\ \text{PC.}\textsc{Check}^\rho(C, d, z, v, \pi) = 1 \end{array} \middle| \begin{array}{r} \rho \leftarrow \mathcal{U}(\lambda) \\ \text{pp}_{\text{PC}} \leftarrow \text{PC.}\textsc{Setup}^\rho(1^\lambda, D), \\ (p, d, z, \omega) \leftarrow \mathcal{A}^\rho(\text{pp}_{\text{PC}}), \\ v \leftarrow p(z), \\ C \leftarrow \text{PC.}\textsc{Commit}^\rho(p, d, \omega), \\ \pi \leftarrow \text{PC.}\textsc{Open}^\rho(p, C, d, z, \omega) \end{array}\right] = 1.$$

In other words, an honest prover will always convince an honest verifier.

Knowledge Soundness: For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$, polynomial-size adversary $\mathcal{A}$ and publicly agreed upon d, there exists an efficient extractor $\mathcal{E}$ such that the following holds:

$$\Pr\left[\begin{array}{c} \text{PC.}\textsc{Check}^\rho(C, d, z, v, \pi) = 1 \\ \Downarrow \\ C = \text{PC.}\textsc{Commit}^\rho(p, d, \omega) \\ v = p(z),\ \deg(p) \leq d \leq D \end{array} \middle| \begin{array}{r} \rho \leftarrow \mathcal{U}(\lambda) \\ \text{pp}_{\text{PC}} \leftarrow \text{PC.}\textsc{Setup}^\rho(1^\lambda, D) \\ (C, d, z, v, \pi) \leftarrow \mathcal{A}^\rho(\text{pp}_{\text{PC}}) \\ (p, \omega) \leftarrow \mathcal{E}^\rho(\text{pp}_{\text{PC}}) \end{array}\right] \geq 1 - \text{negl}(\lambda).$$

In other words, for any adversary, $\mathcal{A}$, outputting an instance, the knowledge extractor can recover p such that the following holds: C is a commitment to p, $v = p(c)$, and the degree of p is properly bounded. Note that for this protocol, we have *knowledge soundness*, meaning that $\mathcal{A}$, must actually have knowledge of p (i.e. the $\mathcal{E}$ can extract it).

Binding: For every maximum degree bound $D = \text{poly}(\lambda) \in \mathbb{N}$ and publicly agreed upon d, no polynomial-size adversary $\mathcal{A}$ can find two polynomials s.t:

$$\Pr\left[\begin{array}{c} p_1 \in \mathbb{F}[X]_{\leq d},\ p_2 \in \mathbb{F}[X]_{\leq d},\ p_1 \neq p_2 \\ \wedge \\ C_1 = C_2 \end{array} \middle| \begin{array}{r} \rho \leftarrow \mathcal{U}(\lambda) \\ \mathrm{pp}_{\mathrm{PC}} \leftarrow \mathrm{PC.Setup}^{\rho}(1^{\lambda}, D) \\ (p_1, p_2, d, \omega_1, \omega_2) \leftarrow \mathcal{A}^{\rho}(\mathrm{pp}_{\mathrm{PC}}) \\ C_1 \leftarrow \mathrm{PC.Commit}(p_1, d, \omega_1) \\ C_2 \leftarrow \mathrm{PC.Commit}(p_2, d, \omega_2) \end{array}\right] \leq \mathrm{negl}(\lambda).$$

In other words, the adversary cannot change the polynomial that he committed to.

2.12 Accumulation Schemes

In 2019 *Halo*[Bowe et al. 2019] was introduced, the first practical example of recursive proof composition without a trusted setup. Using a modified version of the Bulletproofs-style Inner Product Argument (IPA), they present a polynomial commitment scheme. Computing the evaluation of a polynomial $p(z)$ as $v = \langle \boldsymbol{p}^{(\text{coeffs})}, \boldsymbol{z} \rangle$ where $\boldsymbol{z} = (z^0, z^1, \ldots, z^d)$ and $\boldsymbol{p}^{(\text{coeffs})} \in \mathbb{F}^{d+1}$ is the coefficient vector of $p(X)$, using the IPA. However, since the vector $\boldsymbol{z}$ is not private, and has a certain structure, we can split the verification algorithm in two: A sub-linear $\mathrm{PC}_{\mathrm{DL}}$.SuccinctCheck and linear $\mathrm{PC}_{\mathrm{DL}}$.Check. Using the $\mathrm{PC}_{\mathrm{DL}}$.SuccinctCheck we can accumulate n instances, and only perform the expensive linear check (i.e. $\mathrm{PC}_{\mathrm{DL}}$.Check) at the end of accumulation.

In 2020 a paper[Bünz et al. 2020] was released where the authors presented a generalized version of the previous accumulation structure of Halo that they coined *Accumulation Schemes*. Simply put, given a predicate Φ : **Instance** $\rightarrow \{\top, \bot\}$, and m representing the number of instances accumulated for each proof step and may vary for each time AS.Prover is called. An accumulation scheme then consists of the following functions:

- $\mathrm{AS.Setup}(\lambda) \rightarrow \mathrm{pp}_{\mathrm{AS}}$

  When given a security parameter λ (in unary), AS.Setup samples and outputs public parameters $\mathrm{pp}_{\mathrm{AS}}$.

- $\mathrm{AS.Prover}(\boldsymbol{q} : \textbf{Instance}^m, acc_{i-1} : \textbf{Acc}) \rightarrow \textbf{Acc}$

  The prover accumulates the instances $\{q_1, \ldots, q_m\}$ in $\boldsymbol{q}$ and the previous accumulator acc_{i-1} into the new accumulator acc_i.

- $\mathrm{AS.Verifier}(\boldsymbol{q} : \textbf{Instance}^m, acc_{i-1} : \textbf{Option}(\textbf{Acc}), acc_i : \textbf{Acc}) \rightarrow \textbf{Result}(\top, \bot)$

  The verifier checks that the instances $\{q_1, \ldots, q_m\}$ in $\boldsymbol{q}$ were correctly accumulated into the previous accumulator acc_{i-1} to form the new accumulator acc_i. The second argument acc_{i-1} is modelled as an **Option** since in the first accumulation, there will be no accumulator acc_0. In all other cases, the second argument acc_{i-1} must be set to the previous accumulator.

- $\mathrm{AS.Decider}(acc_i : \textbf{Acc}) \rightarrow \textbf{Result}(\top, \bot)$

  The decider performs a single check that simultaneously ensures that all the instances $\boldsymbol{q}$ accumulated in acc_i satisfy the predicate, $\forall j \in [m] : \Phi(q_j) = \top$. Assuming the AS.Verifier has accepted that the accumulator, acc_i correctly accumulates $\boldsymbol{q}$ and the previous accumulator acc_{i-1}.

The completeness and soundness properties for the Accumulation Scheme is defined below:

Completeness. For all (unbounded) adversaries $\mathcal{A}$, where f represents an algorithm producing any necessary public parameters for Φ:

$$\Pr\left[\begin{array}{c} \mathrm{AS.Decider}^{\rho}(\mathrm{acc}_i) = \top \\ \forall j \in [m] : \Phi^{\rho}_{\mathrm{pp}_{\Phi}}(q_j) = \top \\ \Downarrow \\ \mathrm{AS.Verifier}^{\rho}(\boldsymbol{q}, \mathrm{acc}_{i-1}, \mathrm{acc}_i) = \top \\ \mathrm{AS.Decider}^{\rho}(\mathrm{acc}) = \top \end{array} \middle| \begin{array}{r} \rho \leftarrow \mathcal{U}(\lambda) \\ \mathrm{pp}_{\Phi} \leftarrow f^{\rho} \\ \mathrm{pp}_{\mathrm{AS}} \leftarrow \mathrm{AS.Setup}^{\rho}(1^{\lambda}) \\ (\boldsymbol{q}, \mathrm{acc}_{i-1}) \leftarrow \mathcal{A}^{\rho}(\mathrm{pp}_{\mathrm{AS}}, \mathrm{pp}_{\Phi}) \\ \mathrm{acc}_i \leftarrow \mathrm{AS.Prover}^{\rho}(\boldsymbol{q}, \mathrm{acc}_{i-1}) \end{array}\right] = 1.$$

I.e, (AS.Verifier, AS.Decider) will always accept the accumulation performed by an honest prover.

Soundness: For every polynomial-size adversary $\mathcal{A}$:

$$\Pr\left[\begin{array}{c|c} \begin{array}{c}\text{AS.}\textsc{Verifier}^{\rho}(\boldsymbol{q},\text{acc}_{i-1},\text{acc}_i)=\top\\ \text{AS.}\textsc{Decider}^{\rho}(\text{acc}_i)=\top\\ \Downarrow\\ \text{AS.}\textsc{Decider}^{\rho}(\text{acc}_{i-1})=\top\\ \forall j\in[m],\Phi^{\rho}_{\text{pp}_\Phi}(q_j)=\top\end{array} & \begin{array}{c}\rho\leftarrow\mathcal{U}(\lambda)\\ \text{pp}_\Phi\leftarrow f^{\rho}\\ \text{pp}_{\text{AS}}\leftarrow\text{AS.}\textsc{Setup}^{\rho}(1^{\lambda})\\ (\boldsymbol{q},\text{acc}_{i-1},\text{acc}_i)\leftarrow\mathcal{A}^{\rho}(\text{pp}_{\text{AS}},\text{pp}_\Phi)\end{array}\end{array}\right]\geq 1-\text{negl}(\lambda).$$

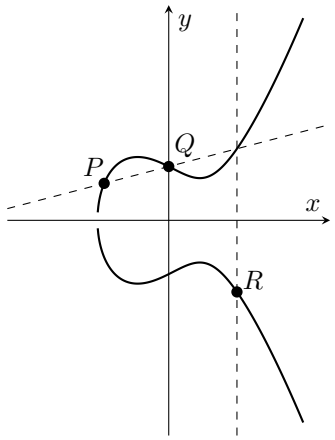
I.e, For all efficiently-generated accumulators $acc_{i-1}, acc_i \in$ **Acc** and predicate inputs $\boldsymbol{q} \in$ **Instance**^{m}, if AS.$\textsc{Decider}(acc_i) = \top$ and AS.$\textsc{Verifier}(\boldsymbol{q}_i, acc_{i-1}, acc_i) = \top$ then, with all but negligible probability, $\forall j \in [m] : \Phi(\text{pp}_\Phi, q_j) = \top$ and AS.$\textsc{Decider}(acc_i) = \top$.

2.13 Cycles of Curves

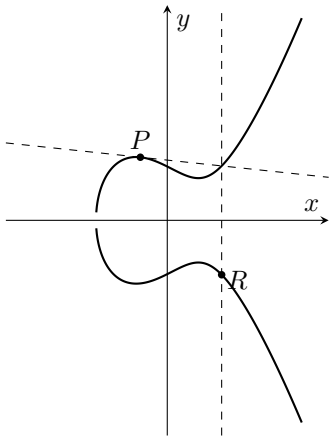
An elliptic curve over a finite field $\mathbb{F}$ is defined by an equation of the form:

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

The set of points $\mathbb{E}(\mathbb{F}) = \mathbb{F} \times \mathbb{F}$ forms an abelian group under point addition, where adding two points $R = P + Q$ is given by simple algebraic formulas in $\mathbb{F}$, and each point has an inverse given by reflection across the x-axis. There is an additional point $\mathcal{O}$, the point at infinity, for which it holds that $P + \mathcal{O} = \mathcal{O} + P = P$.



(a) Elliptic curve $y^2 = x^3 - x + 1$ with points P, Q, and $R = P + Q$.



(b) Elliptic curve $y^2 = x^3 - x + 1$ with point P point $R = 2P$.

Figure 3: Two representations of the elliptic curve $y^2 = x^3 - x + 1$ showing point addition and doubling operations.

Repeated addition defines scalar multiplication. When modelling scalar multiplication, we need two fields, a scalar-field $\mathbb{F}_{\mathcal{S}}$ and a base-field $\mathbb{F}_{\mathcal{B}}$. To see why, consider the operation aP, where $a \in \mathbb{F}_{\mathcal{S}}$ and $P \in \mathbb{E}_{\mathcal{S}}(\mathbb{F}_{\mathcal{B}})$, then the order of the scalar field must be equal to the order of the elliptic curve. Naively, one could choose the larger field of the two for a SNARK circuit and model computation over the smaller field, by using modulo operation in-circuit. With such an approach the circuit is modelling *foreign-field arithmetic* which is very expensive per foreign field operation.

To simplify elliptic curve operations, a *cycle of curves* can be used. A cycle of curves use the other's scalar field as their base field and vice-versa. This means that field operations can be handled natively in the scalar field circuit $\mathbb{F}_{\mathcal{S}}$ and elliptic curve operations are handled natively in the basefield circuit $\mathbb{F}_{\mathcal{B}}$. This improves performance drastically, since the SNARK never need to handle foreign field arithmetic. The cycle of curves used in this project is the Pasta curves[Hopwood 2021], Pallas and Vesta, both of which have the curve equation $y^2 = x^3 + 5$:

- Pallas: $a \in \mathbb{F}_p, P \in \mathbb{E}_p(\mathbb{F}_q)$
- Vesta: $a \in \mathbb{F}_q, P \in \mathbb{E}_q(\mathbb{F}_p)$

Where:

- $|\mathbb{F}_p| = p, |\mathbb{F}_q| = q, |\mathbb{E}_p(\mathbb{F}_q)| = p, |\mathbb{E}_p(\mathbb{F}_q)| = q, p > q$

- $p = 2^{254} + 45560315531419706090280762371685220353$
- $q = 2^{254} + 45560315531506369815346746415080538113$

This is useful when creating proofs. Starting in the first proof in an IVC-setting, we need a proof that verifies some relation, the simplest minimal example would be $R_{(aP)} := aP \stackrel{?}{=} \mathcal{O}$. This then creates two constraint tables and two proofs, one over $\mathbb{F}_{\mathcal{S}} = \mathbb{F}_p$ and one over $\mathbb{F}_{\mathcal{B}} = \mathbb{F}_q$. Then, in the next IVC-step, we need to verify both proofs, but the proof over $\mathbb{F}_p$ produces scalars over $\mathbb{F}_p$ and points over $\mathbb{E}_p(\mathbb{F}_q)$ and the proof over $\mathbb{F}_q$ produces scalars over $\mathbb{F}_q$ and points over $\mathbb{E}_p(\mathbb{F}_q)$. This is because a proof of $R_{(aP)}$ needs to contain both scalars and points. If we did *not* have a cycle of curves this pattern would result in a chain:

- Curve 1: $a \in \mathbb{F}_{p_1}, P \in \mathbb{E}_{p_1}(\mathbb{F}_{p_2})$
- Curve 2: $a \in \mathbb{F}_{p_2}, P \in \mathbb{E}_{p_2}(\mathbb{F}_{p_3})$
- Curve 3: $a \in \mathbb{F}_{p_3}, P \in \mathbb{E}_{p_3}(\mathbb{F}_{p_4})$
- ...

Which means that each p_i must be able to define a valid curve, and if this never cycles, we would need to support this infinite chain of curves.

2.14 Poseidon Hash

Traditionally, SNARKs are defined over somewhat large prime fields, ill-suited for bit-level operation such as XOR. As such, many modern cryptographic hash functions, particularly SHA3, are not particularly well suited for use in many SNARK circuits. The Poseidon[Grassi et al. 2019] Hash specification aims to solve this. The specification defines a cryptographic sponge construction, with the state permutation only consisting of native field operations. This makes poseidon in SNARKs much more efficient. The cryptographic sponge structure is also particularly well suited for Fiat-Shamir, as messages from the prover to the verifier can be modelled with sponge absorption and challenge messages from the verifier to the prover can be modelled with sponge squeezing. Poseidon is still a very new hash function though, and is not nearly as used and "battle-tested" as SHA3, so using it can pose a potential security risk compared to SHA3.

3 $\mathrm{PC}_{\mathrm{DL}}$: The Polynomial Commitment Scheme

3.1 Outline

The Polynomial Commitment Scheme, $\mathrm{PC}_{\mathrm{DL}}$, is based on the Discrete Log assumption, and does not require a trusted setup. Most of the functions simply works as one would expect for a PCS, but uniquely for this scheme, we have the function $\mathrm{PC}_{\mathrm{DL}}.\textsc{SuccinctCheck}$ that allows deferring the expensive part of checking PCS openings until a later point. This function is what leads to the accumulation scheme, $\mathrm{AS}_{\mathrm{DL}}$, which is also based the Discrete Log assumption. We have five main functions:

- $\mathrm{PC}_{\mathrm{DL}}.\textsc{Setup}(\lambda, D)^{\rho_0} \to \mathrm{pp}_{\mathrm{PC}}$

  The setup routine. Given security parameter λ in unary and a maximum degree bound D:

  - Runs $\mathrm{pp}_{\mathrm{CM}} \leftarrow \mathrm{CM}.\textsc{Setup}(\lambda, D+1)$,
  - Samples $H \in_R \mathbb{E}(\mathbb{F}_q)$ using the random oracle $H \leftarrow \rho_0(\mathrm{pp}_{\mathrm{CM}})$,
  - Finally, outputs $\mathrm{pp}_{\mathrm{PC}} = (\mathrm{pp}_{\mathrm{CM}}, H)$.

- $\mathrm{PC}_{\mathrm{DL}}.\textsc{Commit}(p : \mathbb{F}_q^{d'}[X], d : \mathbb{N}, \omega : \mathbf{Option}(\mathbb{F}_q)) \to \mathbb{E}(\mathbb{F}_q)$:

  Creates a commitment to the coefficients of the polynomial p of degree $d' \leq d$ with optional hiding ω, using a Pedersen commitment.

- $\mathrm{PC}_{\mathrm{DL}}.\textsc{Open}^{\rho_0}(p : \mathbb{F}_q^{d'}[X], C : \mathbb{E}(\mathbb{F}_q), d : \mathbb{N}, z : \mathbb{F}_q, \omega : \mathbf{Option}(\mathbb{F}_q)) \to \mathbf{EvalProof}$:

  Creates a proof π that states: "I know $p \in \mathbb{F}_q^{d'}[X]$ with commitment $C \in \mathbb{E}(\mathbb{F}_q)$ s.t. $p(z) = v$ and $\deg(p) = d' \leq d$" where p is private and d, z, v are public.

- $\mathrm{PC}_{\mathrm{DL}}.\textsc{SuccinctCheck}^{\rho_0}(C : \mathbb{E}(\mathbb{F}_q), d : \mathbb{N}, z : \mathbb{F}_q, v : \mathbb{F}_q, \pi : \mathbf{EvalProof}) \to \mathbf{Result}((\mathbb{F}_q^d[X], \mathbb{E}(\mathbb{F}_q)), \bot)$:

  Cheaply checks that a proof π is correct. It is not a full check however, since an expensive part of the check is deferred until a later point.

- $\mathrm{PC}_{\mathrm{DL}}.\textsc{Check}^{\rho_0}(C : \mathbb{E}(\mathbb{F}_q), d : \mathbb{N}, z : \mathbb{F}_q, v : \mathbb{F}_q, \pi : \textbf{EvalProof}) \to \textbf{Result}(\top, \bot)$:

  The full check on π.

The following subsections will describe them in pseudo-code, except for $\mathrm{PC}_{\mathrm{DL}}.\textsc{Setup}$.

3.1.1 $\mathrm{PC}_{\mathrm{DL}}.\textsc{Commit}$

Algorithm 1 $\mathrm{PC}_{\mathrm{DL}}.\textsc{Commit}$

Inputs

$p : \mathbb{F}_q^{d'}[X]$	The univariate polynomial that we wish to commit to.
$d : \mathbb{N}$	A degree bound for p.
$\omega : \textbf{Option}(\mathbb{F}_q)$	Optional hiding factor for the commitment.

Output

$C : \mathbb{E}(\mathbb{F}_q)$	The Pedersen commitment to the coefficients of polynomial p.

Require: $d \leq D$

Require: $(d+1)$ is a power of 2.

1: Let $\boldsymbol{p}^{(\text{coeffs})}$ be the coefficient vector for p.

2: Output $C := \mathrm{CM}.\textsc{Commit}(\boldsymbol{G}, \boldsymbol{p}^{(\text{coeffs})}, \omega)$.

$\mathrm{PC}_{\mathrm{DL}}.\textsc{Commit}$ is rather simple, we just take the coefficients of the polynomial and commit to them using a Pedersen commitment.

3.1.2 $\text{PC}_{\text{DL}}.\textsc{Open}$

Algorithm 2 $\text{PC}_{\text{DL}}.\textsc{Open}^{\rho_0}$

Inputs

$p : \mathbb{F}_q^{d'}[X]$	The univariate polynomial that we wish to open for.
$C : \mathbb{E}(\mathbb{F}_q)$	A commitment to the coefficients of p.
$d : \mathbb{N}$	A degree bound for p.
$z : \mathbb{F}_q$	The element that z will be evaluated on $v = p(z)$.
$\omega : \textbf{Option}(\mathbb{F}_q)$	Optional hiding factor for C. *Must* be included if C has hiding!

Output

EvalProof	Proof of: "I know $p \in \mathbb{F}_q^{d'}[X]$ with commitment C s.t. $p(z) = v$".

Require: $d \leq D$

Require: $(d+1)$ is a power of 2.

1: Let $n = d + 1$

2: Compute $v = p(z)$ and let $n = d + 1$.

3: Sample a random polynomial $\bar{p} \in_R \mathbb{F}_q^{\leq d}[X]$ such that $\bar{p}(z) = 0$.

4: Sample corresponding commitment randomness $\bar{\omega} \in_R \mathbb{F}_q$.

5: Compute a hiding commitment to $\bar{p}$: $\bar{C} \leftarrow \text{PC}_{\text{DL}}.\textsc{Commit}(\bar{p}, d, \bar{\omega}) \in \mathbb{E}(\mathbb{F}_q)$.

6: Compute the challenge $\alpha := \rho_0(C, z, v, \bar{C}) \in \mathbb{F}_q$.

7: Compute commitment randomness $\omega' := \omega + \alpha\bar{\omega} \in \mathbb{F}_q$.

8: Compute the polynomial $p' := p + \alpha\bar{p} = \sum_{i=0} c_i X_i \in \mathbb{F}_q^{\leq d}[X]$.

9: Compute a non-hiding commitment to p': $C' := C + \alpha\bar{C} - \omega' S \in \mathbb{E}(\mathbb{F}_q)$.

10: Compute the 0-th challenge field element $\xi_0 := \rho_0(C', z, v) \in \mathbb{F}_q$, then $H' := \xi_0 H \in \mathbb{E}(\mathbb{F}_q)$.

11: Initialize the vectors ($\boldsymbol{c_0}$ is defined to be coefficient vector of p'):

$$\boldsymbol{c_0} := (c_0, c_1, \ldots, c_d) \in F_q^n$$
$$\boldsymbol{z_0} := (1, z^1, \ldots, z^d) \in F_q^n$$
$$\boldsymbol{G_0} := (G_0, G_1, \ldots, G_d) \in \mathbb{E}(\mathbb{F}_q)_n$$

12: **for** $i \in [\lg(n)]$ **do**

13: Compute $L_i := \text{CM}.\textsc{Commit}(l(\boldsymbol{G_{i-1}}) + H', \; r(\boldsymbol{c_{i-1}}) + \langle r(\boldsymbol{c_{i-1}}), l(\boldsymbol{z_{i-1}})\rangle, \; \perp)$

14: Compute $R_i := \text{CM}.\textsc{Commit}(r(\boldsymbol{G_{i-1}}) + H', \; l(\boldsymbol{c_{i-1}}) + \langle l(\boldsymbol{c_{i-1}}), r(\boldsymbol{z_{i-1}})\rangle, \; \perp)$

15: Generate the i-th challenge $\xi_i := \rho_0(\xi_{i-1}, L_i, R_i) \in \mathbb{F}_q$.

16: Compress values for the next round:

$$\boldsymbol{G_i} := l(\boldsymbol{G_{i-1}}) + \xi_i \cdot r(\boldsymbol{G_{i-1}})$$
$$\boldsymbol{c_i} := l(\boldsymbol{c_{i-1}}) + \xi_i^{-1} \cdot r(\boldsymbol{c_{i-1}})$$
$$\boldsymbol{z_i} := l(\boldsymbol{z_{i-1}}) + \xi_i \cdot r(\boldsymbol{z_{i-1}})$$

17: **end for**

18: Finally output the evaluation proof $\pi := (\boldsymbol{L}, \boldsymbol{R}, U := G^{(0)}, c := c^{(0)}, \bar{C}, \omega')$

Where $l(x), r(x)$ returns the respectively left and right half of the vector given.

The $\text{PC}_{\text{DL}}.\textsc{Open}$ algorithm mostly follows the IPA algorithm from Bulletproofs. Except, in this case we are trying to prove we know polynomial p s.t. $p(z) = v = \langle \boldsymbol{c_0}, \boldsymbol{z_0} \rangle$. So because z is public, we can get away with omitting the generators, $(\boldsymbol{H})$, for $\boldsymbol{b}$ which we would otherwise need in the Bulletproofs IPA. For efficiency we also send along the curve point $U = G^{(0)}$, which the original IPA does not do. The $\text{PC}_{\text{DL}}.\textsc{SuccinctCheck}$ uses U to make its check and $\text{PC}_{\text{DL}}.\textsc{Check}$ verifies the correctness of U.

3.1.3 PC_{DL}.SuccinctCheck

Algorithm 3 $\text{PC}_{\text{DL}}.\text{SuccinctCheck}^{\rho_0}$

Inputs

$C : \mathbb{E}(\mathbb{F}_q)$	A commitment to the coefficients of p.
$d : \mathbb{N}$	A degree bound on p.
$z : \mathbb{F}_q$	The element that p is evaluated on.
$v : \mathbb{F}_q$	The claimed element $v = p(z)$.
π : **EvalProof**	The evaluation proof produced by PC_{DL}.Open.

Output

Result$((\mathbb{F}_q^d[X], \mathbb{E}(\mathbb{F}_q)), \bot)$	The algorithm will either succeed and output $(h : \mathbb{F}_q^d[X], U : \mathbb{E}(\mathbb{F}_q))$ if π is a valid proof and otherwise fail ($\bot$).

Require: $d \leq D$

Require: $(d+1)$ is a power of 2.

Parse π as $(\boldsymbol{L}, \boldsymbol{R}, U := G^{(0)}, c := c^{(0)}, \bar{C}, \omega')$ and let $n = d + 1$.
Compute the challenge $\alpha := \rho_0(C, z, v, \bar{C}) \in \mathbb{F}_q$.
Compute the non-hiding commitment $C' := C + \alpha\bar{C} - \omega' S \in \mathbb{E}(\mathbb{F}_q)$.
Compute the 0-th challenge: $\xi_0 := \rho_0(C', z, v)$, and set $H' := \xi_0 H \in \mathbb{E}(\mathbb{F}_q)$.
Compute the group element $C_0 := C' + vH' \in \mathbb{E}(\mathbb{F}_q)$.
for $i \in [\lg(n)]$ **do**
    Generate the i-th challenge: $\xi_i := \rho_0(\xi_{i-1}, L_i, R_i) \in \mathbb{F}_q$.
    Compute the i-th commitment: $C_i := \xi_i^{-1} L_i + C_{i-1} + \xi_i R_i \in \mathbb{E}(\mathbb{F}_q)$.
end for
Define the univariate polynomial $h(X) := \prod_{i=0}^{\lg(n)-1}(1 + \xi_{\lg(n)-i} X^{2^i}) \in \mathbb{F}_q[X]$.
Compute the evaluation $v' := c \cdot h(z) \in \mathbb{F}_q$.
Check that $C_{lg(n)} \stackrel{?}{=} cU + v'H'$
Output $(h(X), U)$.

The PC_{DL}.SuccinctCheck algorithm performs the same check as in the Bulletproofs protocol. With the only difference being that instead of calculating $G^{(0)}$ itself, it trusts that the verifier sent the correct $U = G^{(0)}$ in the prover protocol, and defers the verification of this claim to PC_{DL}.Check. Notice also the "magic" polynomial $h(X)$, which has a degree d, but can be evaluated in $\lg(d)$ time.

3.1.4 PC_{DL}.Check

Algorithm 4 $\text{PC}_{\text{DL}}.\text{Check}^{\rho_0}$

Inputs

$C : \mathbb{E}(\mathbb{F}_q)$	A commitment to the coefficients of p.
$d : \mathbb{N}$	A degree bound on p
$z : \mathbb{F}_q$	The element that p is evaluated on.
$v : \mathbb{F}_q$	The claimed element $v = p(z)$.
π : **EvalProof**	The evaluation proof produced by PC_{DL}.Open

Output

Result$(\top, \bot)$	The algorithm will either succeed ($\top$) if π is a valid proof and otherwise fail ($\bot$).

Require: $d \leq D$

Require: $(d+1)$ is a power of 2.

Check that $\text{PC}_{\text{DL}}.\text{SuccinctCheck}(C, d, z, v, \pi)$ accepts and outputs (h, U).
Check that $U \stackrel{?}{=} \text{CM.Commit}(\boldsymbol{G}, \boldsymbol{h}^{(\text{coeffs})}, \bot)$, where $\boldsymbol{h}^{(\text{coeffs})}$ is the coefficient vector of the polynomial h.

Since PC_{DL}.SuccinctCheck handles the verification of the IPA given that $U = G^{(0)}$, we run PC_{DL}.SuccinctCheck, then check that $U \stackrel{?}{=} (G^{(0)} = \text{CM.Commit}(\boldsymbol{G}, \boldsymbol{h}^{(\text{coeffs})}, \bot) = \langle \boldsymbol{G}, \boldsymbol{h}^{(\text{coeffs})} \rangle)$.

3.2 Completeness

Check 1 ($C_{lg(n)} \stackrel{?}{=} cU + v'H'$) **in $\mathbf{PC_{DL}}$.SuccinctCheck:**

Let's start by looking at $C_{\lg(n)}$. The verifier computes $C_{\lg(n)}$ as:

$$C_0 = C' + vH' = C + vH'$$
$$C_{\lg(n)} = C_0 + \sum_{i=0}^{\lg(n)-1} \xi_{i+1}^{-1} L_i + \xi_{i+1} R_i$$

Given that the prover is honest, the following invariant should hold:

$$\begin{aligned}
C_{i+1} &= \langle \boldsymbol{c}_{i+1}, \boldsymbol{G}_{i+1} \rangle + \langle \boldsymbol{c}_{i+1}, \boldsymbol{z}_{i+1} \rangle H' \\
&= \langle l(\boldsymbol{c}_i) + \xi_{i+1}^{-1} r(\boldsymbol{c}_i), l(\boldsymbol{G}_i) + \xi_{i+1} r(\boldsymbol{G}_i) \rangle + \langle l(\boldsymbol{c}_i) + \xi_{i+1}^{-1} r(\boldsymbol{c}_i), l(\boldsymbol{z}_i) + \xi_{i+1} r(\boldsymbol{z}_i) \rangle H' \\
&= \langle l(\boldsymbol{c}_i), l(\boldsymbol{G}_i) \rangle + \xi_{i+1} \langle l(\boldsymbol{c}_i)), r(\boldsymbol{G}_i \rangle + \xi_{i+1}^{-1} \langle r(\boldsymbol{c}_i), l(\boldsymbol{G}_i) \rangle + \langle r(\boldsymbol{c}_i), r(\boldsymbol{G}_i) \rangle \\
&+ (\langle l(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle + \xi_{i+1} \langle l(\boldsymbol{c}_i), r(\boldsymbol{z}_i) \rangle + \xi_{i+1}^{-1} \langle r(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle + \langle r(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle) H'
\end{aligned}$$

If we group these terms:

$$\begin{aligned}
C_{i+1} &= \langle l(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle \; + \langle r(\boldsymbol{c}_i), r(\boldsymbol{G}_i) \rangle \quad + \xi_{i+1} \langle l(\boldsymbol{c}_i), r(\boldsymbol{G}_i) \rangle \quad + \xi_{i+1}^{-1} \langle r(\boldsymbol{c}_i), l(\boldsymbol{G}_i) \rangle \\
&+ (\langle l(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle \; + \langle r(\boldsymbol{c}_i), r(\boldsymbol{z}_i) \rangle) H' + \xi_{i+1} \langle l(\boldsymbol{c}_i), r(\boldsymbol{z}_i) \rangle H' + \xi_{i+1}^{-1} \langle r(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle H' \\
&= C_i \qquad\qquad\qquad\qquad + \xi_{i+1} R_i \qquad\qquad + \xi_{i+1}^{-1} L_i
\end{aligned}$$

Where:

$$L_i = \langle r(\boldsymbol{c}_i), l(\boldsymbol{G}_i) \rangle + \langle r(\boldsymbol{c}_i), l(\boldsymbol{z}_i) \rangle H'$$
$$R_i = \langle l(\boldsymbol{c}_i), r(\boldsymbol{G}_i) \rangle + \langle l(\boldsymbol{c}_i), r(\boldsymbol{z}_i) \rangle H'$$

We see why $\boldsymbol{L}, \boldsymbol{R}$ is defined the way they are. They help the verifier check that the original relation hold, by showing it for the compressed form C_{i+1}. $\boldsymbol{L}, \boldsymbol{R}$ is just the minimal information needed to communicate this fact.

This leaves us with the following vectors (notice the slight difference in length):

$$\boldsymbol{L} = (L_1, \ldots, L_{\lg(n)})$$
$$\boldsymbol{R} = (R_1, \ldots, R_{\lg(n)})$$
$$\boldsymbol{C} = (C_0, \ldots, C_{\lg(n)})$$
$$\boldsymbol{\xi} = (\xi_0, \ldots, \xi_{\lg(n)})$$

This means an honest prover will indeed produce $\boldsymbol{L}, \boldsymbol{R}$ s.t. $C_{\lg(n)} = C_0 + \sum_{i=0}^{\lg(n)-1} \xi_{i+1}^{-1} L_i + \xi_{i+1} R_i$

Let's finally look at the left-hand side of the verifying check:

$$C_{\lg(n)} = C_0 + \sum_{i=0}^{\lg(n)-1} \xi_{i+1}^{-1} L_i + \xi_{i+1} R_i$$

The original definition of C_i:

$$C_{\lg(n)} = \langle \boldsymbol{c}_{\lg(n)}, \boldsymbol{G}_{\lg(n)} \rangle + \langle \boldsymbol{c}_{\lg(n)}, \boldsymbol{z}_{\lg(n)} \rangle H'$$

Vectors have length one, so we use the single elements $c^{(0)}, G^{(0)}, c^{(0)}, z^{(0)}$ of the vectors:

$$C_{\lg(n)} = c^{(0)} G^{(0)} + c^{(0)} z^{(0)} H'$$

The verifier has $c^{(0)} = c, G^{(0)} = U$ from $\pi \in$ **EvalProof**:

$$C_{\lg(n)} = cU + cz^{(0)} H'$$

Then, by construction of $h(X) \in \mathbb{F}_q^d[X]$:

$$C_{\lg(n)} = cU + ch(z) H'$$

Finally we use the definition of v':

$$C_{\lg(n)} = cU + v'H'$$

Which corresponds exactly to the check that the verifier makes.

Check 2 ($U \stackrel{?}{=} \text{CM.}\textsc{Commit}(\boldsymbol{G}, \boldsymbol{h}^{(\text{coeffs})}, \perp)$) **in PC_{DL}.Check:**

The honest prover will define $U = G^{(0)}$ as promised and the right-hand side will also become $U = G^{(0)}$ by the construction of $h(X)$.

3.3 Knowledge Soundness

This subsection will not contain a full knowledge soundness proof, but it will be briefly discussed that the *non-zero-knowledge* version of PC_{DL} should be knowledge sound. The knowledge soundness property of PC_{DL} states:

$$\Pr\left[\begin{array}{c|c} \text{PC.}\textsc{Check}^{\rho}(C, d, z, v, \pi) = 1 & \rho \leftarrow \mathcal{U}(\lambda) \\ \Downarrow & \text{pp}_{\text{PC}} \leftarrow \text{PC.}\textsc{Setup}^{\rho}(1^{\lambda}, D) \\ C = \text{PC.}\textsc{Commit}^{\rho}(p, d, \omega) & (C, d, z, v, \pi) \leftarrow \mathcal{A}^{\rho}(\text{pp}_{\text{PC}}) \\ v = p(z),\ \deg(p) \leq d \leq D & (p, \omega) \leftarrow \mathcal{E}^{\rho}(\text{pp}_{\text{PC}}) \end{array}\right] \geq 1 - \text{negl}(\lambda).$$

So, we need to show that:

1. $C = \text{PC.}\textsc{Commit}^{\rho}(p, d, \omega)$
2. $v = p(z)$
3. $\deg(p) \leq d \leq D$

The knowledge extractability of PC_{DL} is almost identical to the IPA from bulletproofs[Bünz et al. 2017], so we assume that we can use the same extractor[3], with only minor modifications. The IPA extractor extracts $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_q^n$ s.t:

$$P = \langle \boldsymbol{G}, \boldsymbol{a} \rangle + \langle \boldsymbol{H}, \boldsymbol{b} \rangle \wedge v = \langle \boldsymbol{c}, \boldsymbol{z} \rangle$$

Running the extractor for PC_{DL} should yield:

$$P = \langle \boldsymbol{G}, \boldsymbol{c} \rangle + \langle \boldsymbol{G}, \boldsymbol{z} \rangle \wedge v = \langle \boldsymbol{c}, \boldsymbol{z} \rangle$$

We should be able to remove the extraction of $\boldsymbol{z}$ since it's public:

$$C = \langle \boldsymbol{G}, \boldsymbol{c} \rangle \wedge v = \langle \boldsymbol{c}, \boldsymbol{z} \rangle$$

1. $C = \langle \boldsymbol{G}, \boldsymbol{c} \rangle = \text{PC.}\textsc{Commit}(c, G, \perp) = \text{PC.}\textsc{Commit}^{\rho}(p, d, \perp)$, $\omega = \perp$ since we don't consider zero-knowledge.
2. $v = \langle \boldsymbol{c}, \boldsymbol{z} \rangle = \langle \boldsymbol{p}^{(\text{coeffs})}, \boldsymbol{z} \rangle = p(z)$ by definition of p.
3. $\deg(p) \leq d \leq D$. The first bound holds since the vector committed to is known to have length $n = d + 1$, the second bound holds trivially, as it's checked by PC_{DL}.Check

The authors, of the paper followed[Bünz et al. 2020], note that the soundness technically breaks down when turning the IPA into a non-interactive protocol (which is the case for PC_{DL}), and that transforming the IPA into a non-interactive protocol such that the knowledge extractor does not break down is an open problem:

> **Security of the resulting non-interactive argument.** It is known from folklore that applying the Fiat–Shamir transformation to a public-coin k-round interactive argument of knowledge with negligible soundness error yields a non-interactive argument of knowledge in the random-oracle model where the extractor $\mathcal{E}$ runs in time exponential in k. In more detail, to extract from an adversary that makes t queries to the random oracle, $\mathcal{E}$ runs in time $t^{\mathcal{O}(k)}$. In our setting, the inner-product argument has $k = \mathcal{O}(\log d)$ rounds, which means that if we apply this folklore result, we would obtain an extractor that runs in superpolynomial (but sub-exponential) time $t^{\mathcal{O}(\log d)} = 2^{\mathcal{O}(log(\lambda)^2)}$. It remains an interesting open problem to construct an extractor that runs in polynomial time.

This has since been solved in a 2023 paper[Attema et al. 2023]. The abstract of the paper describes:

[3]Admittedly, this assumption is not a very solid one if the purpose was to create a proper knowledge soundness proof, but as the section is more-so devoted to give a justification for why PC_{DL} *ought to be* sound, it will do. In fact, the authors of the accumulation scheme paper[Bünz et al. 2020], use a similar argument more formally by stating (without direct proof!), that the PC_{DL} protocol is a special case of the IPA presented in another paper[Bünz et al. 2019] by mostly the same authors.

Unfortunately, the security loss for a $(2\mu+1)$-move protocol is, in general, approximately Q^μ, where Q is the number of oracle queries performed by the attacker. In general, this is the best one can hope for, as it is easy to see that this loss applies to the μ-fold sequential repetition of Σ-protocols, ..., we show that for $(k^1, \ldots, k^\mu)$-special-sound protocols (which cover a broad class of use cases), the knowledge error degrades linearly in Q, instead of Q^μ.

The IPA is exactly such a $(k^1, \ldots, k^\mu)$-special-sound protocol, they even directly state that this result applies to bulletproofs. As such we get a knowledge error that degrades linearly, instead of superpolynomially, in number of queries, t, that the adversary makes to the random oracle. Thus, the extractor runs in the required polynomial time $(\mathcal{O}(t) = \mathcal{O}(\text{poly}(\lambda)))$.

3.4 Efficiency

Given two operations $f(x), g(x)$ where $f(x)$ is more expensive than $g(x)$, we only consider $f(x)$, since $\mathcal{O}(f(n)+g(n)) = \mathcal{O}(f(n))$. For all the algorithms, the most expensive operations will be scalar multiplications. We also don't bother counting constant operations, that does not scale with the input. Also note that:

$$\mathcal{O}\left(\sum_{i=2}^{\lg(n)} \frac{n}{i^2}\right) = \mathcal{O}\left(n \sum_{i=2}^{\lg(n)} \frac{1}{i^2}\right) = \mathcal{O}(n \cdot c) = \mathcal{O}(n)$$

Remember that in the below contexts $n = d + 1$

- PC_{DL}.Commit: $n = \mathcal{O}(d)$ scalar multiplications and $n = \mathcal{O}(d)$ point additions.
- PC_{DL}.Open:
  - Step 1: 1 polynomial evaluation, i.e. $n = \mathcal{O}(d)$ field multiplications.
  - Step 13 & 14: Both commit $\lg(n)$ times, i.e. $2(\sum_{i=2}^{\lg(n)}(n+1)/i) = \mathcal{O}(2n)$ scalar multiplications. The sum appears since we halve the vector length each loop iteration.
  - Step 16: $\lg(n)$ vector dot products, i.e. $\sum_{i=2}^{\lg(n)} n/i = \mathcal{O}(n)$ scalar multiplications.

  In total, $\mathcal{O}(3d) = \mathcal{O}(d)$ scalar multiplications.
- PC_{DL}.SuccinctCheck:
  - Step 7: $\lg(n)$ hashes.
  - Step 8: $3\lg(n)$ point additions and $2\lg(n)$ scalar multiplications.
  - step 11: The evaluation of $h(X)$ which uses $\mathcal{O}(\lg(n))$ field additions.

  In total, $\mathcal{O}(2\lg(n)) = \mathcal{O}(\lg(d))$ scalar multiplications.
- PC_{DL}.Check:
  - Step 1: Running PC_{DL}.SuccinctCheck takes $\mathcal{O}(2\lg(d))$ scalar multiplications.
  - Step 2: Running CM.Commit$(\boldsymbol{G}, \boldsymbol{h}^{(\text{coeffs})}, \perp)$ takes $\mathcal{O}(d)$ scalar multiplications.

  Since step two dominates, we have $\mathcal{O}(d)$ scalar multiplications.

So PC_{DL}.Open, PC_{DL}.Check and PC_{DL}.Commit is linear and, importantly, PC_{DL}.SuccinctCheck is sub-linear.

Sidenote: The runtime of $h(X)$

Recall the structure of $h(X)$:

$$h(X) := \prod_{i=0}^{\lg(n)-1} (1 + \xi_{\lg(n)-i} X^{2^i}) \in \mathbb{F}_q[X]$$

First note that $\left(\prod_{i=0}^{\lg(n)-1} a\right)$ leads to $\lg(n)$ factors. Calculating X^{2^i} can be computed as:

$$X^{2^0}, X^{2^1} = (X^{2^0})^2, X^{2^2} = (X^{2^1})^2, \ldots$$

So that part of the evaluation boils down to the cost of squaring in the field. We therefore have $\lg(n)$ squarings (from X^{2^i}), and $\lg(n)$ field multiplications from $\xi_{\lg(n)-i} \cdot X^{2^i}$. Each squaring can naively be modelled as a field multiplication ($x^2 = x \cdot x$). We therefore end up with $2\lg(n) = \mathcal{O}(\lg(n))$ field multiplications and $\lg(n)$ field additions. The field additions are ignored as the multiplications dominate.

Thus, the evaluation of $h(X)$ requires $\mathcal{O}(\lg(n))$ field multiplications, which dominate the runtime.

4 $\mathrm{AS}_{\mathrm{DL}}$: The Accumulation Scheme

4.1 Outline

The $\mathrm{AS}_{\mathrm{DL}}$ accumulation scheme is an accumulation scheme for accumulating polynomial commitments. This means that the corresponding predicate, Φ_{AS}, that we accumulate for, represents the checking of polynomial commitment openings, $\Phi_{\mathrm{AS}}(q_i) = \mathrm{PC}_{\mathrm{DL}}.\textsc{Check}(q_i)$. The instances are assumed to have the same degree bounds. A slight deviation from the general AS specification, is that that the algorithms don't take the old accumulator acc_{i-1} as input, instead, since it has the same form as instances $((C_{\mathrm{acc}}, d_{\mathrm{acc}}, z_{\mathrm{acc}}, v_{\mathrm{acc}}), \pi_V)$, it will be prepended to the instance list $\boldsymbol{q}$. We have six main functions:

- $\mathrm{AS}_{\mathrm{DL}}.\textsc{Setup}(1^\lambda, D) \to \mathrm{pp}_{\mathrm{AS}}$

  Outputs $\mathrm{pp}_{\mathrm{AS}} = \mathrm{PC}_{\mathrm{DL}}.\textsc{Setup}(1^\lambda, D)$.

- $\mathrm{AS}_{\mathrm{DL}}.\textsc{CommonSubroutine}(\boldsymbol{q} : \mathbf{Instance}^m, \pi_V : \mathbf{AccHiding}) \to \mathbf{Result}((\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q^d[X]), \bot)$

  $\mathrm{AS}_{\mathrm{DL}}.\textsc{CommonSubroutine}$ will either succeed if the instances have consistent degree and hiding parameters and will otherwise fail. It accumulates all previous instances into a new polynomial $h(X)$, and is run by both $\mathrm{AS}_{\mathrm{DL}}.\textsc{Prover}$ and $\mathrm{AS}_{\mathrm{DL}}.\textsc{Verifier}$ in order to ensure that the accumulator, generated from $h(X)$ correctly accumulates the instances. It returns $(\bar{C}, d, z, h(X))$ representing the information needed to create the polynomial commitment represented by acc_i.

- $\mathrm{AS}_{\mathrm{DL}}.\textsc{Prover}(\boldsymbol{q} : \mathbf{Instance}^m) \to \mathbf{Result}(\mathbf{Acc}, \bot)$:

  Accumulates the instances $\boldsymbol{q}$, and an optional previous accumulator acc_{i-1}, into a new accumulator acc_i. If there is a previous accumulator acc_{i-1} then it is converted into an instance, since it has the same form, and prepended to $\boldsymbol{q}$, *before calling the prover*.

- $\mathrm{AS}_{\mathrm{DL}}.\textsc{Verifier}(\boldsymbol{q} : \mathbf{Instance}^m, \mathrm{acc}_i : \mathbf{Acc}) \to \mathbf{Result}(\top, \bot)$:

  Verifies that the instances $\boldsymbol{q}$ (as with $\mathrm{AS}_{\mathrm{DL}}.\textsc{Prover}$, including a possible acc_{i-1}) was correctly accumulated into the new accumulator acc_i.

- $\mathrm{AS}_{\mathrm{DL}}.\textsc{Decider}(\mathrm{acc}_i : \mathbf{Acc}) \to \mathbf{Result}(\top, \bot)$:

  Checks the validity of the given accumulator acc_i along with all previous accumulators that was accumulated into acc_i.

This means that accumulating m instances, $\boldsymbol{q} = [q_i]^m$, should yield acc_i, using the $\mathrm{AS}_{\mathrm{DL}}.\textsc{Prover}(\boldsymbol{q})$. If the verifier accepts $\mathrm{AS}_{\mathrm{DL}}.\textsc{Verifier}(\boldsymbol{q}, \mathrm{acc}_i) = \top$, and $\mathrm{AS}_{\mathrm{DL}}.\textsc{Decider}$ accepts the accumulator ($\mathrm{AS}_{\mathrm{DL}}.\textsc{Decider}(\mathrm{acc}_i) = \top$), then all the instances, $\boldsymbol{q}$, will be valid, by the soundness property of the accumulation scheme. This is proved for $\mathrm{AS}_{\mathrm{DL}}$ in the soundness section. Note that this also works recursively, since $q_{\mathrm{acc}_{i-1}} \in \boldsymbol{q}$ is also proven valid by the decider.

The following subsections will describe the functions in pseudo-code, except $\mathrm{AS}_{\mathrm{DL}}.\textsc{Setup}$.

4.1.1 AS_{DL}.CommonSubroutine

Algorithm 5 AS_{DL}.CommonSubroutine

Inputs

$\boldsymbol{q} : \textbf{Instance}^m$	New instances *and accumulators* to be accumulated.
π_V : **AccHiding**	Necessary parameters if hiding is desired.

Output

$\textbf{Result}((\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q^d[X]), \perp)$	The algorithm will either succeed $(\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q^d[X])$ if the instances has consistent degree and hiding parameters and will otherwise fail ($\perp$).

Require: $(D+1) = 2^k$, where $k \in \mathbb{N}$

1: Parse d from q_1.
2: Parse π_V as (h_0, U_0, ω), where $h_0(X) = aX + b \in \mathbb{F}_q^1[X], U_0 \in \mathbb{E}(\mathbb{F}_q)$ and $\omega \in \mathbb{F}_q$
3: Check that U_0 is a deterministic commitment to h_0: $U_0 = \text{PC}_{\text{DL}}.\text{Commit}(h, d, \perp)$.
4: **for** $j \in [0, m]$ **do**
5: Parse q_j as a tuple $((C_j, d_j, z_j, v_j), \pi_j)$.
6: Compute $(h_j(X), U_j) := \text{PC}_{\text{DL}}.\text{SuccinctCheck}^{\rho_0}(C_j, d_j, z_j, v_j, \pi_j)$.
7: Check that $d_j \stackrel{?}{=} d$
8: **end for**
9: Compute the challenge $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U}) \in \mathbb{F}_q$
10: Let the polynomial $h(X) := h_0 + \sum_{j=1}^m \alpha^j h_j(X) \in \mathbb{F}_q[X]$
11: Compute the accumulated commitment $C := U_0 + \sum_{j=1}^m \alpha^j U_j$
12: Compute the challenge $z := \rho_1(C, h(X)) \in \mathbb{F}_q$.
13: Randomize C: $\bar{C} := C + \omega S \in \mathbb{E}(\mathbb{F}_q)$.
14: Output $(\bar{C}, D, z, h(X))$.

The AS_{DL}.CommonSubroutine does most of the work of the AS_{DL} accumulation scheme. It takes the given instances and runs the PC_{DL}.SuccinctCheck on them to acquire $[(h_j(X), U_j)]_{i=0}^m$ for each of them. It then creates a linear combination of $h_j(X)$ using a challenge point α and computes the claimed commitment for this polynomial $C = \sum_{j=1}^m \alpha^j U_j$, possibly along with hiding information. This routine is run by both AS_{DL}.Prover and AS_{DL}.Verifier in order to ensure that the accumulator, generated from $h(X)$ correctly accumulates the instances. To see the intuition behind why this works, refer to the note in the AS_{DL}.Decider section.

4.1.2 AS_{DL}.Prover

Algorithm 6 AS_{DL}.Prover

Inputs

$\boldsymbol{q} : \textbf{Instance}^m$	New instances *and accumulators* to be accumulated.

Output

$\textbf{Result}(\textbf{Acc}, \perp)$	The algorithm will either succeed $((\bar{C}, d, z, v, \pi), \pi_V) \in \textbf{Acc}$) if the instances has consistent degree and hiding parameters and otherwise fail ($\perp$).

Require: $\forall(\_, d_i, \_, \_, \_) \in \boldsymbol{q}, \forall(\_, d_j, \_, \_, \_) \in \boldsymbol{q} : d_i = d_j \wedge d_i \leq D$
Require: $(d_i + 1) = 2^k$, where $k \in \mathbb{N}$

1: Sample a random linear polynomial $h_0(X) \in_R F_q^{\leq d}[X]$
2: Then compute a deterministic commitment to $h_0(X)$: $U_0 := \text{PC}_{\text{DL}}.\text{Commit}(h_0, d, \perp)$
3: Sample commitment randomness $\omega \in_R F_q$, and set $\pi_V := (h_0, U_0, \omega)$.
4: Then, compute the tuple $(\bar{C}, d, z, h(X)) := \text{AS}_{\text{DL}}.\text{CommonSubroutine}(\boldsymbol{q}, \pi_V)$.
5: Compute the evaluation $v := h(z) \in \mathbb{F}_q$.
6: Generate the evaluation proof $\pi := \text{PC}_{\text{DL}}.\text{Open}(h(X), \bar{C}, d, z, \omega)$.
7: Finally, output the accumulator $\text{acc}_i = ((\bar{C}, d, z, v, \pi), \pi_V)$.

Simply accumulates the the instances, $\boldsymbol{q}$, into new accumulator acc_i, using AS_{DL}.CommonSubroutine.

4.1.3 AS_{DL}.Verifier

Algorithm 7 AS_{DL}.Verifier

Inputs

$\boldsymbol{q} : \textbf{Instance}^m$	New instances *and possible accumulator* to be accumulated.
$\text{acc}_i : \textbf{Acc}$	The accumulator that accumulates $\boldsymbol{q}$. *Not* the previous accumulator acc_{i-1}.

Output

$\textbf{Result}(\top, \bot)$	The algorithm will either succeed ($\top$) if acc_i correctly accumulates $\boldsymbol{q}$ and otherwise fail ($\bot$).

Require: $(D+1) = 2^k$, where $k \in \mathbb{N}$

1: Parse acc_i as $((\bar{C}, d, z, v, \_), \pi_V)$

2: The accumulation verifier computes $(\bar{C}', d', z', h(X)) := \text{AS}_{\text{DL}}.\text{CommonSubroutine}(\boldsymbol{q}, \pi_V)$

3: Then checks that $\bar{C}' \stackrel{?}{=} \bar{C}, d' \stackrel{?}{=} d, z' \stackrel{?}{=} z$, and $h(z) \stackrel{?}{=} v$.

The verifier also runs AS_{DL}.CommonSubroutine, therefore verifying that acc_i correctly accumulates $\boldsymbol{q}$, which means:

- $\bar{C} = C + \omega S = \sum_{j=1}^{m} \alpha^j U_j + \omega S$
- $\forall (\_, d_j, \_, \_, \_) \in \boldsymbol{q} : d_j = d$
- $z = \rho_1(C, h(X))$
- $v = h(z)$
- $h(X) = \sum_{j=0}^{m} \alpha^j h_j(X)$
- $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U})$

4.1.4 AS_{DL}.Decider

Algorithm 8 AS_{DL}.Decider

Inputs

$\text{acc}_i : \textbf{Acc}$	The accumulator.

Output

$\textbf{Result}(\top, \bot)$	The algorithm will either succeed ($\top$) if the accumulator has correctly accumulated all previous instances and will otherwise fail ($\bot$).

Require: $\text{acc}_i.d \leq D$

Require: $(\text{acc}_i.d + 1) = 2^k$, where $k \in \mathbb{N}$

1: Parse acc_i as $((\bar{C}, d, z, v, \pi), \_)$

2: Check $\top \stackrel{?}{=} \text{PC}_{\text{DL}}.\text{Check}(\bar{C}, d, z, v, \pi)$

The decider fully checks the accumulator acc_i, this verifies each previous accumulator meaning that:

$$\begin{aligned}
&\forall i \in [n], \forall j \in [m] : \\
&\text{AS}_{\text{DL}}.\text{Verifier}((\text{ToInstance}(\text{acc}_{i-1}) + \boldsymbol{q}_{i-1}), \text{acc}_i) \wedge \text{AS}_{\text{DL}}.\text{Decider}(\text{acc}_n) \implies \\
&\Phi_{\text{AS}}(q_j^{(i)}) = \text{PC}_{\text{DL}}.\text{Check}(q_j^{(i)}) = \top
\end{aligned}$$

The sidenote below gives an intuition why this is the case.

Sidenote: Why does checking acc_i check all previous instances and previous accumulators?

The AS_{DL}.Prover runs the AS_{DL}.CommonSubroutine that creates an accumulated polynomial h from $[h_j(X)]^m$ that is in turn created for each instance $q_j \in \boldsymbol{q}_i$ by PC_{DL}.SuccinctCheck:

$$h_j(X) := \prod_{i=0}^{lg(n)} (1 + \xi_{\text{lg}(n)-i} \cdot X^{2^i}) \in F_q[X]$$

We don't mention the previous accumulator acc_{i-1} explicitly as it's treated as an instance in the protocol. We also only consider the case where the protocol does not have zero knowledge, meaning

that we omit the blue parts of the protocol. The AS_{DL}.Verifier shows that C is a commitment to $h(X)$ in the sense that it's a linear combination of all $h_j(X)$'s from the previous instances, by running the same AS_{DL}.CommonSubroutine algorithm as the prover to get the same output. Note that the AS_{DL}.Verifier does not guarantee that C is a valid commitment to $h(X)$ in the sense that $C = \text{PC}_{\text{DL}}.\text{Commit}(h, d, \bot)$, that's the AS_{DL}.Decider's job. Since AS_{DL}.Verifier does not verify that each U_j is valid, and therefore that $C = \text{PC}_{\text{DL}}.\text{Commit}(h, d, \bot)$, we now wish to argue that AS_{DL}.Decider verifies this for all the instances.

Showing that $C = \text{PC}_{\text{DL}}.\text{Commit}(h, d, \bot)$:

The AS_{DL}.Prover has a list of instances $(q_1, \ldots, q_m) = \boldsymbol{q}_i$, then runs PC_{DL}.SuccinctCheck on each of them, getting $(U_1, \ldots, U_m)$ and $(h_1(X), \ldots, h_m(X))$. For each element U_j in the vector $\boldsymbol{U} \in \mathbb{E}(\mathbb{F}_q)^m$ and each element $h_j(X)$ in the vector $\boldsymbol{h} \in (\mathbb{F}_q^{\leq d}[X])^m$, the AS_{DL}.Prover defines:

$$h(X) := \sum_{j=1}^{m} \alpha^j h_j(X)$$

$$C := \sum_{j=1}^{m} \alpha^j U_j$$

Since we know from the AS_{DL}.Verifier:

1. $\text{PC}_{\text{DL}}.\text{SuccinctCheck}(q_j) = \top$
2. $C_{\text{acc}_i} = \sum_{j=1}^{m} \alpha^j U_j$
3. $z_{\text{acc}_i} = \rho_1(C, h(X))$
4. $h_{\text{acc}_i}(X) = \sum_{j=0}^{m} \alpha^j h_j(X)$
5. $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U})$

Which implies that $\Phi_{\text{AS}}(q_j) = \top$ if $U = G^{(0)}$. We then argue that when the AS_{DL}.Decider checks that $C = \text{PC}_{\text{DL}}.\text{Commit}(h(X), d, \bot)$, then that implies that each U_j is a valid commitment to $h_j(X)$, $U_j = \text{PC}_{\text{DL}}.\text{Commit}(h_j(X), d, \bot) = \langle \boldsymbol{G}, \boldsymbol{h_j} \rangle$, thereby performing the second check of PC_{DL}.Check, on all q_j instances at once. We know that:

1. PC_{DL}.Check tells us that $C_{\text{acc}_i} = \sum_{j=1}^{m} \alpha^j U_j$ except with negligible probability, since,
2. The binding property of CM states that it's hard to find a different C', s.t., $C = C'$ but $h_{\text{acc}_i}(X) \neq h'(X)$. Which means that $h_{\text{acc}_i}(X) = h'(X)$.
3. Define $B_j = \langle \boldsymbol{G}, \boldsymbol{h_j}^{(\text{coeffs})} \rangle$. If $\exists j \in [m]$ $B_j \neq U_j$ then U_j is not a valid commitment to $h_j(X)$ and $\sum_{j=1}^{m} \alpha_j B_j \neq \sum_{j=1}^{m} \alpha_j U_j$. As such C_{acc_i} will not be a valid commitment to $h_{\text{acc}_i}(X)$. Unless,
4. $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U})$ or $z = \rho_1(C, h(X))$ is constructed in a malicious way, which is hard, since they're from the random oracle.

<!– TODO: This is wrong –>

To sum up, this means that running the AS_{DL}.Decider corresponds to checking all U_j's.

What about checking the previous instances, $\boldsymbol{q}_{i-1}$, accumulated into the previous accumulator, acc_{i-1}? The accumulator for $\boldsymbol{q}_{i-1}$ is represented by an instance $acc_{i-1} = (C = \text{PC}_{\text{DL}}.\text{Commit}(h_{\text{acc}_{i-1}}, d, \bot), d, z, v = h_{\text{acc}_{i-1}}(z), \pi)$, which, as mentioned, behaves like all other instances in the protocol and represents a PCS opening to $h_{\text{acc}_{i-1}}(X)$. Since acc_{i-1} is represented as an instance, and we showed that as long as each instance is checked by AS.Verifier (which acc_{i-1} also is), running $\text{PC}_{\text{DL}}.\text{Check}(\text{acc}_i)$ on the corresponding accumulation polynomial $h_{\text{acc}_i}(X)$ is equivalent to performing the second check $U_j = \text{PC}_{\text{DL}}.\text{Commit}(h_j(X), d, \bot)$ on all the $h_j(X)$ that $h_{\text{acc}_i}(X)$ consists of. Intuitively, if any of the previous accumulators were invalid, then their commitment will be invalid, and the next accumulator will also be invalid. That is, the error will propagate. Therefore, we will also check the previous set of instances $\boldsymbol{q}_{i-1}$, and by induction, all accumulated instances $\boldsymbol{q}$ and accumulators $\mathbf{acc}$.

4.2 Completeness

AS_{DL}.Verifier runs the same algorithm (AS_{DL}.CommonSubroutine) with the same inputs and, given that AS_{DL}.Prover is honest, will therefore get the same outputs, these outputs are checked to be equal to the ones received from the prover. Since these were generated honestly by the prover, also using AS_{DL}.CommonSubroutine, the AS_{DL}.Verifier will accept with probability 1, returning $\top$. Intuitively, this also makes sense. It's the job of the verifier to verify that each instance is accumulated correctly into the accumulator. This verifier does the same work as the prover and checks that the output matches.

As for the AS_{DL}.Decider, it just runs PC_{DL}.Check on the provided accumulator, which represents a evaluation proof i.e. an instance. This check will always pass, as the prover constructed it honestly.

4.3 Soundness

In order to prove soundness, we first need a helper lemma:

Lemma: Zero-Finding Game:

Let $\text{CM} = (\text{CM.Setup}, \text{CM.Commit})$ be a perfectly binding commitment scheme. Fix a maximum degree $D \in \mathbb{N}$ and a random oracle ρ that takes commitments from CM to F_{pp}. Then for every family of functions $\{f_{\text{pp}}\}_{\text{pp}}$ and fields $\{F_{\text{pp}}\}_{\text{pp}}$ where:

- $f_{\text{pp}} \in \mathcal{M} \to F_{\text{pp}}^{\leq D}[X]$
- $F \in \mathbb{N} \to \mathbb{N}$
- $|F_{\text{pp}}| \geq F(\lambda)$

That is, for all functions, f_{pp}, that takes a message, $\mathcal{M}$ as input and outputs a maximum D-degree polynomial. Also, usually $|F_{\text{pp}}| \approx F(\lambda)$. For every message format L and computationally unbounded t-query oracle algorithm $\mathcal{A}$, the following holds:

$$\Pr\left[\begin{array}{c|c} p \neq 0 \\ \wedge \\ p(z) = 0 & \begin{array}{c} \rho \leftarrow \mathcal{U}(\lambda) \\ \text{pp}_{\text{CM}} \leftarrow \text{CM.Setup}(1^\lambda, L) \\ (m, \omega) \leftarrow \mathcal{A}^\rho(\text{pp}_{\text{CM}}) \\ C \leftarrow \text{CM.Commit}(m, \omega) \\ z \in F_{\text{pp}} \leftarrow \rho(C) \\ p := f_{\text{pp}}(m) \end{array} \end{array}\right] \leq \sqrt{\frac{D(t+1)}{F(\lambda)}}$$

Intuitively, the above lemma states that for any non-zero polynomial p, that you can create using the commitment C, it will be highly improbable that a random evaluation point z be a root of the polynomial p, $p(z) = 0$. For reference, this is not too unlike the Schwartz-Zippel Lemma.

Proof:

We construct a reduction proof, showing that if an adversary $\mathcal{A}$ that wins with probability δ in the above game, then we construct an adversary $\mathcal{B}$ which breaks the binding of the commitment scheme with probability at least:

$$\frac{\delta^2}{t+1} - \frac{D}{F(\lambda)}$$

Thus, leading to a contradiction, since CM is perfectly binding. Note, that we may assume that $\mathcal{A}$ always queries $C \leftarrow \text{CM.Commit}(m, \omega)$ for its output (m, ω), by increasing the query bound from t to $t+1$.

The Adversary $\mathcal{B}(\text{pp}_{\text{CM}})$

1: Run $(m, \omega) \leftarrow \mathcal{A}^\rho(\text{pp}_{\text{CM}})$, simulating its queries to ρ.
2: Get $C \leftarrow \text{CM.Commit}(m, \omega)$.
3: Rewind $\mathcal{A}$ to the query $\rho(C)$ and run to the end, drawing fresh randomness for this and subsequent oracle queries, to obtain (p', ω').
4: Output $((m, \omega), (m', \omega'))$.

Each (m, ω)-pair represents a message where $p \neq 0 \wedge p(z) = 0$ for $z = \rho(\text{CM.Commit}(m, \omega))$ and $p = f_{\text{pp}}(m)$ with probability δ

Let:

$$\begin{aligned} C' &:= \text{CM.}\textsc{Commit}(p', \omega') \\ z &:= \rho(C) \\ z' &:= \rho(C') \\ p &:= f_{pp}(m) \\ p' &:= f_{pp}(m') \end{aligned}$$

By the Local Forking Lemma[Bellare et al. 2019], the probability that $p(z) = p'(z') = 0$ and $C = C'$ is at least $\frac{\delta^2}{t+1}$. Let's call this event E:

$$E := (p(z) = p'(z') = 0 \wedge C = C')$$

Then, by the triangle argument:

$$\Pr[E] \leq \Pr[E \wedge (p = p')] + \Pr[E \wedge (p \neq p')]$$

And, by Schwartz-Zippel:

$$\begin{aligned} \Pr[E \wedge (p = p')] &\leq \frac{D}{|F_{\text{pp}}|} \implies \\ &\leq \frac{D}{F(\lambda)} \end{aligned}$$

Thus, the probability that $\mathcal{B}$ breaks binding is:

$$\begin{aligned} \Pr[E \wedge (p = p')] + \Pr[E \wedge (p \neq p')] &\geq \Pr[E] \\ \Pr[E \wedge (p \neq p')] &\geq \Pr[E] - \Pr[E \wedge (p = p')] \\ \Pr[E \wedge (p \neq p')] &\geq \frac{\delta^2}{t+1} - \frac{D}{F(\lambda)} \end{aligned}$$

Yielding us the desired probability bound. Isolating δ will give us the probability bound for the zero-finding game:

$$\begin{aligned} 0 &= \frac{\delta^2}{t+1} - \frac{D}{F(\lambda)} \\ \frac{\delta^2}{t+1} &= \frac{D}{F(\lambda)} \\ \delta^2 &= \frac{D(t+1)}{F(\lambda)} \\ \delta &= \sqrt{\frac{D(t+1)}{F(\lambda)}} \end{aligned}$$

□

For the above Lemma to hold, the algorithms of CM must not have access to the random oracle ρ used to generate the challenge point z, but CM may use other oracles. The lemma still holds even when $\mathcal{A}$ has access to the additional oracles. This is a concrete reason why domain separation, as mentioned in the Fiat-Shamir subsection, is important.

With this lemma, we wish to show that given an adversary $\mathcal{A}$, that breaks the soundness property of AS_{DL}, we can create a reduction proof that then breaks the above zero-finding game. We fix $\mathcal{A}, D = \text{poly}(\lambda)$ from the AS soundness definition:

$$\Pr\left[\begin{array}{c|c} \begin{array}{c} \text{AS}_{\text{DL}}.\textsc{Verifier}^{\rho_1}((q_{\text{acc}_{i-1}} \mathbin{+\!\!+} \boldsymbol{q}), \text{acc}_i) = \top, \\ \text{AS}_{\text{DL}}.\textsc{Decider}^{\rho_1}(\text{acc}_i) = \top \\ \wedge \\ \exists i \in [n] : \Phi_{\text{AS}}(q_i) = \bot \end{array} & \begin{array}{r} \rho_0 \leftarrow \mathcal{U}(\lambda), \rho_1 \leftarrow \mathcal{U}(\lambda), \\ \text{pp}_{\text{PC}} \leftarrow \text{PC}_{\text{DL}}.\textsc{Setup}^{\rho_0}(1^\lambda, D), \\ \text{pp}_{\text{AS}} \leftarrow \text{AS}_{\text{DL}}.\textsc{Setup}^{\rho_1}(1^\lambda, \text{pp}_{\text{PC}}), \\ (\boldsymbol{q}, \text{acc}_{i-1}, \text{acc}_i) \leftarrow \mathcal{A}^{\rho_1}(\text{pp}_{\text{AS}}, \text{pp}_{\text{PC}}) \\ q_{acc_{i-1}} \leftarrow \textsc{ToInstance}(\text{acc}_{i-1}) \end{array} \end{array}\right] \leq \text{negl}(\lambda)$$

We call the probability that the adversary $\mathcal{A}$ wins the above game δ. We bound δ by constructing two adversaries, $\mathcal{B}_1, \mathcal{B}_2$, for the zero-finding game. Assuming:

- $\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{wins}] = \delta - \text{negl}(\lambda)$
- $\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{wins}] = 0$

These assumptions will be proved after defining the adversaries concretely. So, we claim that the probability that either of the adversaries wins is $\delta - \text{negl}(\lambda)$ and that both of the adversaries cannot win the game at the same time. With these assumptions, we can bound δ:

$$\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{ wins}] = \Pr[\mathcal{B}_1 \text{ wins}] + \Pr[\mathcal{B}_2 \text{ wins}] - \Pr[\mathcal{B}_1 \text{ wins} \wedge \mathcal{B}_2 \text{ wins}]$$
$$\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{ wins}] = \Pr[\mathcal{B}_1 \text{ wins}] + \Pr[\mathcal{B}_2 \text{ wins}] - 0$$
$$\delta - \text{negl}(\lambda) \leq \sqrt{\frac{D(t+1)}{F(\lambda)}} + \sqrt{\frac{D(t+1)}{F(\lambda)}}$$
$$\delta - \text{negl}(\lambda) \leq 2 \cdot \sqrt{\frac{D(t+1)}{|\mathbb{F}_q|}}$$
$$\delta \leq 2 \cdot \sqrt{\frac{D(t+1)}{|\mathbb{F}_q|}} + \text{negl}(\lambda)$$

Meaning that δ is negligible, since $q = |\mathbb{F}_q|$ is superpolynomial in λ. We define two perfectly binding commitment schemes to be used for the zero-finding game:

- CM_1:
  - $\text{CM}_1.\textsc{Setup}^{\rho_0}(1^\lambda, D) := \text{pp}_{\text{PC}} \leftarrow \text{PC}_{\text{DL}}.\textsc{Setup}^{\rho_0}(1^\lambda, D)$
  - $\text{CM}_1.\textsc{Commit}((p(X), h(X)), \_) := (C \leftarrow \text{PC}_{\text{DL}}.\textsc{Commit}(p(X), d, \perp), h)$
  - $\mathcal{M}_{\text{CM}_1} := \{(p(X), h(X) = \alpha^j h_j(X))\} \in \mathcal{P}((\mathbb{F}_q^{\leq D}[X])^2)$
  - $z_{\text{CM}_1} := \rho_1(\text{CM}_1.\textsc{Commit}((p(X), h(X)), \_)) = \rho_1((C \leftarrow \text{PC}_{\text{DL}}.\textsc{Commit}(p(X), d, \perp), h)) = z_{\text{acc}}$
- CM_2:
  - $\text{CM}_2.\textsc{Setup}^{\rho_0}(1^\lambda, D) := \text{pp}_{\text{PC}} \leftarrow \text{PC}_{\text{DL}}.\textsc{Setup}^{\rho_0}(1^\lambda, D)$
  - $\text{CM}_2.\textsc{Commit}([(h_j(X), U_j)]^m, \_) := [(h_j(X), U_j)]^m$:
  - $\mathcal{M}_{\text{CM}_2} := \{[(h_j(X), U_j)]^m\} \in \mathcal{P}((\mathbb{F}_q^{\leq D}[X] \times \mathbb{E}(\mathbb{F}_q))^m)$
  - $z_{\text{CM}_2} := \rho_1(\text{CM}_2.\textsc{Commit}([(h_j(X), U_j)]^m, \_)) = \rho_1([(h_j(X), U_j)]^m) = \alpha$

Note that the CM_1, CM_2 above are perfectly binding, since they either return a Pedersen commitment, without binding, or simply return their input. $\mathcal{M}_{\text{CM}_1}$ consists of pairs of polynomials of a maximum degree D, where $\forall j \in [m] : h(X) = \alpha^j h_j(X)$. $\mathcal{M}_{\text{CM}_2}$ consists of a list of pairs of a maximum degree D polynomial, $h_j(X)$, and U_j is a group element. Notice that $z_a = z_{\text{acc}}$ and $z_b = \alpha$ where z_{acc} and α are from the AS_{DL} protocol.

We define the corresponding functions $f_{\text{pp}}^{(1)}, f_{\text{pp}}^{(2)}$ for CM_1, CM_2 below:

- $f_{\text{pp}}^{(1)}(p(X), h(X) = [h_j(X)]^m) := a(X) = p(X) - \sum_{j=1}^m \alpha^j h_j(X)$,
- $f_{\text{pp}}^{(2)}(p = [(h_j(X), U_j)]^m) := b(Z) = \sum_{j=1}^m a_j Z^j$ where for each $j \in [m]$:
  - $B_j \leftarrow \text{PC}_{\text{DL}}.\textsc{Commit}(h_j, d, \perp)$
  - Compute $b_j : b_j G = U_j - B_j$

We then construct an intermediate adversary, $\mathcal{C}$, against PC_{DL}, using $\mathcal{A}$:

The Adversary $\mathcal{C}^{\rho_1}(\text{pp}_{\text{PC}})$

1: Parse pp_{PC} to get the security parameter 1^λ and set AS public parameters $\text{pp}_{\text{AS}} := 1^\lambda$.
2: Compute $(\boldsymbol{q}, \text{acc}_{i-1}, \text{acc}_i) \leftarrow \mathcal{A}^{\rho_1}(\text{pp}_{\text{AS}})$.
3: Parse pp_{PC} to get the degree bound D.
4: Output $(D, \text{acc}_i = (C_{\text{acc}}, d_{\text{acc}}, z_{\text{acc}}, v_{\text{acc}}), \boldsymbol{q})$.

The above adversary also outputs $\boldsymbol{q}$ for convenience, but the knowledge extractor simply ignores this. Running the knowledge extractor, $\mathcal{E}_{\mathcal{C}}^{\rho_1}$, on $\mathcal{C}$, meaning we extract acc_i, will give us p. Provided that $\text{AS}_{\text{DL}}.\textsc{Decider}$ accepts, the following will hold with probability $(1 - \text{negl})$:

- C_{acc} is a deterministic commitment to $p(X)$.
- $p(z_{\text{acc}}) = v_{\text{acc}}$

- $\deg(p) \leq d_{\text{acc}} \leq D$

Let's denote successful knowledge extraction s.t. the above points holds as $E_{\mathcal{E}}$. Furthermore, the $\text{AS}_{\text{DL}}.\textsc{Decider}$ (and $\text{AS}_{\text{DL}}.\textsc{Verifier}$'s) will accept with probability δ, s.t. the following holds:

- $\text{AS}_{\text{DL}}.\textsc{Verifier}^{\rho_1}((q_{\text{acc}_{i-1}} + \boldsymbol{q}), \text{acc}_i) = \top$
- $\text{AS}_{\text{DL}}.\textsc{Decider}^{\rho_1}(\text{acc}_i) = \top$
- $\exists i \in [n] : \Phi_{\text{AS}}(q_i) = \bot \implies \text{PC}_{\text{DL}}.\textsc{Check}^{\rho_0}(C_i, d_i, z_i, v_i, \pi_i) = \bot$

Let's denote this event as $E_{\mathcal{D}}$. We're interested in the probability $\Pr[E_{\mathcal{E}} \wedge E_{\mathcal{D}}]$. Using the chain rule we get:

$$\begin{aligned}\Pr[E_{\mathcal{E}} \wedge E_{\mathcal{D}}] &= \Pr[E_{\mathcal{E}} \mid E_{\mathcal{D}}] \cdot \Pr[E_{\mathcal{E}}] \\ &= \delta \cdot (1 - \text{negl}(\lambda)) \\ &= \delta - \delta \cdot \text{negl}(\lambda) \\ &= \delta - \text{negl}(\lambda)\end{aligned}$$

Now, since $\text{AS}_{\text{DL}}.\textsc{Verifier}^{\rho_1}((q_{\text{acc}_{i-1}} + \boldsymbol{q}), \text{acc}_i)$ accepts, then, by construction, all the following holds:

1. For each $j \in [m]$, $\text{PC}_{\text{DL}}.\textsc{SuccinctCheck}$ accepts.
2. Parsing $\text{acc}_i = (C_{\text{acc}}, d_{\text{acc}}, z_{\text{acc}}, v_{\text{acc}})$ and setting $\alpha := \rho_1([(h_j(X), U_j)]^m)$, we have that:
   - $z_{\text{acc}} = \rho_1(C_{\text{acc}}, [h_j(X)]^m)$
   - $C_{\text{acc}} = \sum_{j=1}^{m} \alpha^j U_j$
   - $v_{\text{acc}} = \sum_{j=1}^{m} \alpha^j h_j(z)$

Also by construction, this implies that either:

- $\text{PC}_{\text{DL}}.\textsc{SuccinctCheck}$ rejects, which we showed above is not the case, so therefore,
- The group element U_j is not a commitment to $h_j(X)$.

We utilize this fact in the next two adversaries, $\mathcal{B}_1, \mathcal{B}_2$, constructed, to win the zero-finding game for CM_1, CM_2 respectively, with non-negligible probability:

The Adversary $\mathcal{B}_k^{\rho_1}(\text{pp}_{\text{AS}})$

Compute $(D, \text{acc}_i, \boldsymbol{q}) \leftarrow C^{\rho_1}(\text{pp}_{\text{AS}})$.
Compute $p \leftarrow \mathcal{E}_C^{\rho}(\text{pp}_{\text{AS}})$.
For each $q_j \in \boldsymbol{q}$: $(h_j, U_j) \leftarrow \text{PC}_{\text{DL}}.\textsc{SuccinctCheck}(q_j)$.
Compute $\alpha := \rho_1([(h_j, U_j)]^m)$.
if $k = 1$ **then**
    Output $((n, D), (p, h := ([h_j]^m)))$
else if $k = 2$ **then**
    Output $((n, D), ([(h_j, U_j)]^m))$
end if

Remember, the goal is to find an evaluation point, s.t. $a(X) \neq 0 \wedge a(z_a) = 0$ for CM_1 and $b(X) \neq 0 \wedge b(z_b) = 0$ for CM_2. We set $z_a = z_{\text{acc}}$ and $z_b = \alpha$. Now, there are then two cases:

1. $C_{\text{acc}} \neq \sum_{j=1}^{m} \alpha^j B_j$: This means that for some $j \in [m]$, $U_j \neq B_j$. Since C_{acc} is a commitment to $p(X)$, $p(X) - h(X)$ is not identically zero, but $p(z_{\text{acc}}) = h(z_{\text{acc}})$. Thusly, $a(X) \neq 0$ and $a(z_{\text{acc}}) = 0$. Because $z_{\text{acc}} = z_a$ is sampled using the random oracle ρ_1, $\mathcal{B}_1$ wins the zero-finding game against $(\text{CM}_1, \{f_{\text{pp}}^{(1)}\}_{\text{pp}})$.
2. $C = \sum_{j=1}^{n} \alpha^j B_j$. Which means that for all $j \in [m]$, $U_j = B_j$. Since $C = \sum_{j=1}^{n} \alpha^j U_j$, α is a root of the polynomial $a(Z)$, $a(\alpha) = 0$. Because α is sampled using the random oracle ρ_1, $\mathcal{B}_2$ wins the zero-finding game against $(CM_2, \{f_{\text{pp}}^{(2)}\}_{\text{pp}})$.

So, since one of these adversaries always win if $E_{\mathcal{E}} \wedge E_{\mathcal{D}}$, the probability that $\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{wins}]$ is indeed $\delta - \text{negl}(\lambda)$. And since the above cases are mutually exclusive we also have $\Pr[\mathcal{B}_1 \text{ wins} \vee \mathcal{B}_2 \text{wins}]$. Thus, we have proved that, given the zero-finding game Lemma, the probability that an adversary can break the soundness property of the AS_{DL} accumulation scheme is negligible.

□

4.4 Efficiency

- AS_{DL}.CommonSubroutine:
  - Step 6: m calls to PC_{DL}.SuccinctCheck, $m \cdot \mathcal{O}(2\lg(d)) = \mathcal{O}(2m\lg(d))$ scalar multiplications.
  - Step 11: m scalar multiplications.

  Step 6 dominates with $\mathcal{O}(2m\lg(d)) = \mathcal{O}(m\lg(d))$ scalar multiplications.
- AS_{DL}.Prover:
  - Step 4: 1 call to AS_{DL}.CommonSubroutine, $\mathcal{O}(md)$ scalar multiplications.
  - Step 5: 1 evaluation of $h(X)$, $\mathcal{O}(\lg(d))$ scalar multiplications.
  - Step 6: 1 call to PC_{DL}.Open, $\mathcal{O}(3d)$ scalar multiplications.

  Step 6 dominates with $\mathcal{O}(3d) = \mathcal{O}(d)$ scalar multiplications.
- AS_{DL}.Verifier:
  - Step 2: 1 call to AS_{DL}.CommonSubroutine, $\mathcal{O}(2m\lg(d))$ scalar multiplications.

  So $\mathcal{O}(2m\lg(d)) = \mathcal{O}(m\lg(d))$ scalar multiplications.
- AS_{DL}.Decider:
  - Step 2: 1 call to PC_{DL}.Check, with $\mathcal{O}(d)$ scalar multiplications.

  $\mathcal{O}(d)$ scalar multiplications.

So AS_{DL}.Prover and AS_{DL}.Decider are linear and AS_{DL}.Decider is sub-linear.

5 IVC-friendly Plonk Scheme

We construct a NARK with support for recursive proofs, heavily inspired by the Z-Cash's Halo2 and Mina's Kimchi proof systems. As with both of these protocols, we instantiate them over a Bulletproofs-style PCS and corresponding accumulation scheme[Jakobsen 2025]. We have taken liberties to try to simplify the protocol at the cost of performance, but have taken an effort to ensure we only affect constant-time factors. Meaning that the following still hold for our protocol:

1. The prover is bounded by $\mathcal{O}(n\lg(n))$
2. The verifier time is linear ($\mathcal{O}(n)$)
3. The proof size is bounded by $\mathcal{O}(\lg(n))$

The below sections will describe this protocol in detail.

5.1 Arguments

We now describe the arguments used in Plonk. We can safely assume that the degree bound of any polynomial is always much smaller than the size of the field, $d \ll |\mathbb{F}|$.

5.1.1 Vanishing Argument

The checks that the verifier makes in Plonk boils down to checking identities of the following form:

$$\forall s \in S : f(s) \stackrel{?}{=} 0$$

For some polynomial $f(X) \in \mathbb{F}_{\leq d}$ and some set $S \subset \mathbb{F}$. The subset, S, may be much smaller than $\mathbb{F}$ as is the case for Plonk where S is set to be the set of roots of unity ($S = H = \{\omega^1, \omega^2, \ldots, \omega^n\}$). Since we ultimately model the above check with challenge scalars, using just S, might not be sound. For this purpose, we can construct the **Single Polynomial Vanishing Argument Protocol**:

Prover$(f \in \mathbb{F}_{\leq d}[X])$		**Verifier**
$C_f = \text{PC.}\textsc{Commit}(f(X), d, \bot)$		
$z_S(X) = \prod_{s \in S}(X - s)$		
$t(X) = \frac{f(X)}{z_S}$		
$C_t = \text{PC.}\textsc{Commit}(t(X), d, \bot)$	$\xrightarrow{C_f, C_t}$	$\xi \in_R \mathbb{F}$
$v_f = f(\xi)$	$\xleftarrow{\xi}$	
$\pi_f = \text{PC.}\textsc{Open}(f(X), C_f, d, \xi, \bot)$		
$v_t = t(\xi)$		
$\pi_t = \text{PC.}\textsc{Open}(t(X), C_f, d, \xi, \bot)$	$\xrightarrow{v_f, \pi_f, v_t, \pi_t}$	$v_f \stackrel{?}{=} v_t \cdot z_S(\xi)$
		$\text{PC.}\textsc{Check}(C_f, d, \xi, v_f, \pi_f) \stackrel{?}{=} \top$
		$\text{PC.}\textsc{Check}(C_t, d, \xi, v_t, \pi_t) \stackrel{?}{=} \top$

Correctness

Define $p(X) = f_i(\xi) - t(\xi)z_S(\xi)$. For any $\xi \in \mathbb{F} \setminus S$, the following holds:

$$\begin{aligned} p(\xi) &= f_i(\xi) - t(\xi)z_S(\xi) \\ &= f_i(\xi) - \left(\frac{f_i(\xi)}{z_S(\xi)}\right) z_S(\xi) \\ &= 0 \end{aligned}$$

□

Soundness

The factor theorem states that if $f(X)$ is a univariate polynomial, then $x - a$ is a factor of $f(X)$ if and only if $f(a) = 0$. This means $z_S(X)$ only divides $f(X)$ if and only if all of $s \in S : f(s) = 0$. The Schwartz-Zippel Lemma states that evaluating a non-zero polynomial on an input chosen randomly from a large enough set is extremely unlikely to evaluate to zero. Specifically, it ensures that $Pr[p(\xi) = 0 \wedge p(X) \neq 0] \leq \frac{deg(p(X))}{|\mathbb{F}|}$. Clearly $\xi \in_R \mathbb{F}$ is sampled from a large enough set as $|\mathbb{F}| \gg d \geq \deg(p(X))$ and therefore $Pr[p(\xi) = 0 \mid P \neq 0]$ is negligible. Lastly, the evaluation checked depends on the soundness of the underlying PCS scheme used, but we assume that it has knowledge soundness and binding. From all this, we conclude that the above vanishing argument is sound.

Extending to multiple f's

We can use a linear combination of α to generalize the **Single Polynomial Vanishing Argument**, creating a **Vanishing Argument Protocol**:

Prover($\boldsymbol{f} \in \mathbb{F}^k_{\leq d}[X]$)		**Verifier**
$C_{f_i} = \text{PC.Commit}(f_i(X), d, \perp)$	$\xrightarrow{\boldsymbol{C_f}}$	$\alpha \in_R \mathbb{F}$
$z_S(X) = \prod_{s \in S}(X - s)$	$\xleftarrow{\alpha}$	
$t(X) = \frac{\sum_{i=1}^{k-1} \alpha^i f_i(X)}{z_S}$		
$C_t = \text{PC.Commit}(t(X), d, \perp)$	$\xrightarrow{C_t}$	$\xi \in_R \mathbb{F}$
$v_{f_i} = f_i(\xi)$	$\xleftarrow{\xi}$	
$\pi_{f_i} = \text{PC.Open}(f_i(X), C_{f_i}, d, \xi, \perp)$		
$v_t = t(\xi)$		
$\pi_t = \text{PC.Open}(t(X), C_f, d, \xi, \perp)$	$\xrightarrow{\boldsymbol{v_f}, \boldsymbol{\pi_f}, v_t, \pi_t}$	$\sum_{i=0}^{k-1} \alpha^i v_{f_i} = v_t \cdot z_S(\xi)$
		$\forall i \in [k] : \text{PC.Check}(C_{f_i}, d, \xi, v_{f_i}, \pi_{f_i}) \stackrel{?}{=} \top$
		$\text{PC.Check}(C_t, d, \xi, v_t, \pi_t) \stackrel{?}{=} \top$

You can compress the $k+1$ evaluations proofs into a single evaluation proof using the **Batched Evaluation Proofs Protocol**.

5.1.2 Batched Evaluation Proofs

If we have m polynomials, $\boldsymbol{f}$, that all need to evaluate to zero at the same challenge ξ, normally, we could construct m opening proofs, and verify these. We can, however, use the following protocol to only create the **Batched Evaluation Proofs Protocol**:

Prover($\boldsymbol{f} \in \mathbb{F}^k_{\leq d}[X]$)		**Verifier**
$C_{f_i} = \text{PC.Commit}(f_i(X), d, \perp)$	$\xrightarrow{\boldsymbol{C_f}}$	$\alpha, \xi \in_R \mathbb{F}$
$w(X) = \sum_{i=0}^{k-1} \alpha^i f_i(X)$	$\xleftarrow{\alpha, \xi}$	
$C_w(X) = \text{PC.Commit}(w(X), d, \perp)$		
$v_{f_i} = f_i(\xi)$		
$\pi_w = \text{PC.Open}(w(X), C_w, d, \xi, \perp)$	$\xrightarrow{\pi_w, \boldsymbol{v_f}}$	$C_w = \sum_{i=0}^{k-1} \alpha^i C_{f_i}$
		$v_w = \sum_{i=0}^{k-1} \alpha^i v_{f_i}$
		$\text{PC.Check}(C_w, d, \xi, v_w, \pi_w) \stackrel{?}{=} \top$

Correctness:

Since:

- $C_w = \sum_{i=0}^{k-1} \alpha^i C_{f_i} = \text{PC.Commit}(f(X), d, \perp)$ (Assuming a homomorphic commitment scheme)
- $v_w = \sum_{i=0}^{k-1} \alpha^i v_{f_i} = w(\xi)$ (By definition of $w(X)$)

The correctness of the protocol is derived from the correctness of the underlying PCS.

Soundness:

Recall that:

$$\sum_{i=0}^{k-1} \alpha^i f_i(\xi) = \sum_{i=0}^{k-1} \alpha^i v_i$$

This can be recontextualized as a polynomial:

$$p(\alpha) = \sum_{i=0}^{k-1} \alpha^i (f_i(\xi) - v_i) = 0$$

Then from Schwartz-Zippel, we achieve soundness, since the probability that this polynomial evaluates to zero given that it's not a zero-polynomial is $\frac{k}{|\mathbb{F}|}$.

5.1.3 Grand Product Argument

Suppose a prover, given polynomials $f(X), g(x)$ wanted to prove that these polynomials when viewed as sets are equal to each other. This is called multi-set equality, i.e, $\forall \omega \in H : f(X) = g(X)$. This relation can be modelled with a grand product:

$$\begin{aligned} f'(X) &= f(X) + \gamma \\ g'(X) &= g(X) + \gamma \\ \prod_{i \in [n]} f'(\omega^i) &= \prod_{i \in [n]} g'(\omega^i) \end{aligned}$$

Completeness is trivial. As for soundness. We can interpret each side of the equality as a polynomial variable in γ:

$$\begin{aligned} p(X) &= \prod_{i \in [n]} f(\omega^i) + X \\ q(X) &= \prod_{i \in [n]} g(\omega^i) + X \end{aligned}$$

Then by Schwarz-Zippel, if we consider $r(X) = p(X) - q(X)$, if $r(\gamma) = 0$ and $r(\gamma) : \gamma \in_R \mathbb{F}$ then $p(X) = q(X)$. Now, we just need to prove that $p(X) = q(X) \implies \{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$.

Consider the roots of $p(X)$ and $q(X)$, starting with $p(X)$:

$$p(X) = \prod_{i \in [n]} f(\omega^i) + X$$

This polynomial evaluates to zero only if one of the factors equals $f(\omega^i)$. The same argument for $q(X)$ can also be applied:

$$\begin{aligned} \text{roots}(p(X)) &= \{-f(\omega^1), \ldots, -f(\omega^n)\} \\ &= \{-a_1, \ldots, -a_n\} \\ \text{roots}(q(X)) &= \{-g(\omega^1), \ldots, -g(\omega^n)\} \\ &= \{-b_1, \ldots, -b_n\} \end{aligned}$$

Since the two polynomials are equal, they must have the same roots. Thus:

$$\begin{aligned} \text{roots}(p(X)) = \text{roots}(q(X)) &\implies \\ \{-a_1, \ldots, -a_n\} = \{-b_1, \ldots, -b_n\} &\implies \\ \{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\} & \end{aligned}$$

□

We still need to convert $\prod_{i\in n} f'(\omega^i) = \prod_{i\in n} g'(\omega^i)$ to a polynomial that can be checked by the verifier. The prover can create a polynomial, $z(X)$, to check the relation:

$$z(\omega^1) = 1$$
$$z(\omega^i) = \prod_{1\leq j<i} \frac{f'(\omega^j)}{g'(\omega^j)}$$

The prover needs to convince the verifier that $z(X)$ has the expected form:

$$\begin{aligned}
z(\omega^i) &= \prod_{1\leq j<i} \frac{f'(\omega^j)}{g'(\omega^j)} \\
z(\omega^i) &= z(\omega^{i-1})\frac{f'(\omega^{i-1})}{g'(\omega^{i-1})} \\
z(\omega^{i+1}) &= z(\omega^i)\frac{f'(\omega^i)}{g'(\omega^i)} \\
z(\omega^{i+1})g'(\omega^i) &= z(\omega^i)f'(\omega^i) \\
z(\omega^i)f'(\omega^i) &= z(\omega\cdot\omega^i)g'(\omega^i)
\end{aligned}$$

While also proving that $z(\omega^1) = 1$. This leads to the following polynomials:

$$f_{CC_1}(X) = l_1(X)(z(X) - 1)$$
$$f_{CC_2}(X) = z(X)f'(X) - z(\omega X)g'(X)$$

That should be zero for all $\omega \in H$, which can be checked using the **Vanishing Argument Protocol**. Finally, it needs to be argued that checking these constraints lead to the desired goal of checking whether $\prod_{\omega\in H} f'(\omega) \stackrel{?}{=} \prod_{\omega\in H} g'(\omega)$. Notice that in the last case, $i = n$:

$$\begin{aligned}
z(\omega^n)f'(\omega^n) &= z(\omega^{n+1})g'(\omega) \\
\prod_{1\leq j<i} \frac{f'(\omega^j)}{g'(\omega^j)} f'(\omega^n) &= g'(\omega) \\
\prod_{1\leq j<i} \frac{f'(\omega^j)f'(\omega^n)}{g'(\omega^j)g'(\omega^n)} &= 1 \\
\frac{\prod_{i\in[n]} f'(\omega^i)}{\prod_{i\in[n]} g'(\omega^i)} &= 1 \\
\prod_{i\in[n]} f'(\omega^i) &= \prod_{i\in[n]} g'(\omega^i)
\end{aligned}$$

And since, by the Vanishing Argument, $f_{CC_1}(X)$ and $f_{CC_2}(X)$ holds for all $\omega \in H$, it also holds for ω^n.

Prover$(f, g \in \mathbb{F}_{\leq d}[X])$		**Verifier**
$C_f = \text{PC.}\textsc{Commit}(f(X), d, \perp)$		
$C_g = \text{PC.}\textsc{Commit}(g(X), d, \perp)$	$\xrightarrow{C_f, C_g}$	$\alpha, \beta \in_R \mathbb{F}$
$z(\omega^1) = 1$		
$z(\omega^i) = \prod_{1\leq j<i} \frac{f(\omega^j)+\gamma}{g(\omega^j)+\gamma}$	$\longleftrightarrow$	$\forall h \in H : f_{CC_1}(h) \stackrel{?}{=} l_1(h)(z(h) - 1)$
	$\longleftrightarrow$	$\forall h \in H : f_{CC_2}(h) \stackrel{?}{=} z(h)f'(h) - z(\omega h)g'(h)$

5.2 Protocol Components

5.2.1 Gate Constraints

Imagine we want to prove that we have a witness for $3x_1^2 + 5x_2 = 47$, meaning we want to show that we know x_1, x_2 such that the equation equals 47. We can represent that equation as a simple circuit.

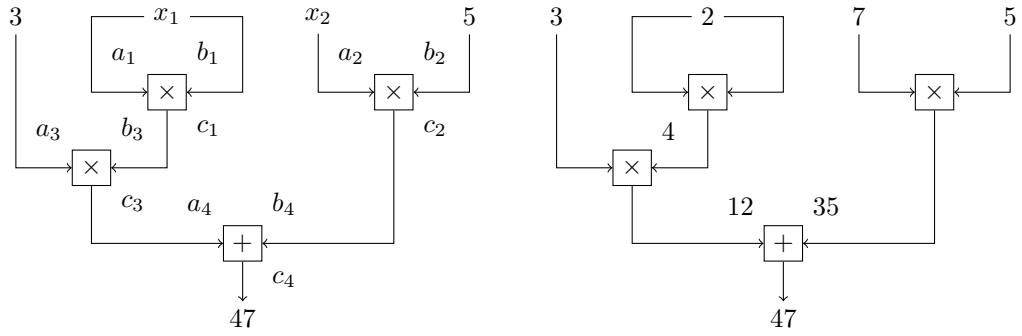


Figure 4: Two ways of viewing the circuit representing $3x_1^2 + 5x_2 = 47$. The left circuit is also instantiated with the witness x_1, x_2.

This is a trivial problem, so we deduce that $x_1 = 2, x_2 = 7$. From the graphs above, we can construct vectors representing the wire values of our circuit:

$$\begin{aligned}\boldsymbol{w} &= [2, 7, 4, 3, 12, 5, 35, 47] \\ \boldsymbol{a} &= [2, 7, 3, 12] \\ \boldsymbol{b} &= [2, 5, 4, 35] \\ \boldsymbol{c} &= [4, 35, 12, 42]\end{aligned}$$

We can then create polynomials $a(X), b(X), c(X)$ corresponding to the left-input wire, the right-input wire and the output wire respectively:

$$\begin{aligned}a(X) &= \text{lagrange}(\boldsymbol{a}) \\ b(X) &= \text{lagrange}(\boldsymbol{b}) \\ c(X) &= \text{lagrange}(\boldsymbol{c})\end{aligned}$$

Now, we can use selector polynomials, $q_l(X), q_r(X), q_o(X), q_m(X), q_c(X)$, to show that the constructed polynomials $a(X), b(X), c(X)$ satisfy the circuit relations by proving that a constructed polynomial $f_{GC}(X) = 0$ at $i = [1, 8]$:

$$f_{GC}(X) = a(X)q_l(X) + b(X)q_r(X) + c(X)q_o(X) + a(X)b(X)q_m(X) + q_c(X)$$

Where $a(X), b(X), c(X)$ are private and the selector polynomials are public. Notice that we can represent this as a table:

i	$a(i)$	$b(i)$	$c(i)$	$q_l(i)$	$q_r(i)$	$q_o(i)$	$q_m(i)$	$q_c(i)$
1	3	0	0	1	0	0	0	-3
2	5	0	0	1	0	0	0	-5
3	47	0	0	1	0	0	0	-47
4	2	2	4	0	0	-1	1	0
5	5	7	35	0	0	-1	1	0
6	4	3	12	0	0	-1	1	0
7	35	12	47	1	1	-1	0	0
8	0	0	0	0	0	0	0	0

Lagrange interpolation is slow, with a runtime of $\mathcal{O}(n^2)$, we can instead use FFT to construct our polynomials, which has a runtime of $\mathcal{O}(n\log(n))$. For this, we construct the polynomials over the roots of unity ($\omega^1, \omega^2, \ldots, \omega^8$ where ω is the 8'th root of unity), meaning that our table becomes:

ω^i	$a(\omega^i)$	$b(\omega^i)$	$c(\omega^i)$	$q_l(\omega^i)$	$q_r(\omega^i)$	$q_o(\omega^i)$	$q_m(\omega^i)$	$q_c(\omega^i)$
ω^1	3	0	0	1	0	0	0	-3
ω^2	5	0	0	1	0	0	0	-5
ω^3	47	0	0	1	0	0	0	-47
ω^4	2	2	4	0	0	-1	1	0
ω^5	5	7	35	0	0	-1	1	0
ω^6	4	3	12	0	0	-1	1	0
ω^7	35	12	47	1	1	-1	0	0
ω^8	0	0	0	0	0	0	0	0

Now we wish to prove that:

$$\forall \omega \in H = \{\omega^1, ..., \omega^6\} : f_{GC}(X) = 0$$

And for this, we can use the **Vanishing Argument Protocol**. And in order for the verifier to know that $f_{GC}(X)$ is constructed honestly, i.e. it respects the public selector polynomials, we can use the **Batched Evaluations Proofs Protocol** over each witness polynomial instead of $f_{GC}(X)$. This securely gives the verifier $v_{f_{GC}} = f_{GC}(\xi), v_a = a(\xi), v_b = b(\xi), v_c = c(\xi)$ and the verifier can then check:

$$v_f = v_a q_l(\xi) + v_b q_r(\xi) + v_c q_o(\xi) + v_a v_b q_m(\xi) + q_c(\xi)$$

We still need to handle copy constraints, because as can be seen in the table, we need to verify identities between wires (like $a(\omega^1) = b(\omega^1)$). For this we need *Copy Constraints.*

5.2.2 Copy Constraints

For example we want to show that $a(\omega^1) = b(\omega^1)$, first we concatenate the lists $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$:

$$\boldsymbol{f} = [2, 7, 3, 12] +\!\!+ [2, 5, 4, 35] +\!\!+ [4, 35, 12, 42] = [2, 7, 3, 12, 2, 5, 4, 35, 4, 35, 12, 42]$$

Now, we wish to show, that for some permutation $\sigma : \mathbb{F}^k \to \mathbb{F}^k$, the list remains unchanged once permuted:

$$\boldsymbol{f} = \sigma(\boldsymbol{f})$$

This permutation permutes the list according to what wires we wish to show are equal:

$$\boldsymbol{f} = [2, 7, 3, 12] +\!\!+ [2, 5, 4, 35] +\!\!+ [4, 35, 12, 42]$$

From the circuit in Figure 4 we gather that the following wires must be equal:

$$a_1 = b_1, \quad c_1 = b_3, \quad c_3 = a_4, \quad c_2 = b_4$$

To highlight the values of $\boldsymbol{f}$ and $\sigma(\boldsymbol{f})$, the specific values have been subbed out for variables below:

$$\begin{aligned} \boldsymbol{f} &= [a_1, a_2, a_3, a_4] +\!\!+ [b_1, b_2, b_3, b_4] +\!\!+ [c_1, c_2, c_3, c_4] \\ \sigma(\boldsymbol{f}) &= [b_1, a_2, a_3, c_3] +\!\!+ [a_1, b_2, c_1, c_2] +\!\!+ [b_3, b_4, a_4, c_4] \end{aligned}$$

If the prover is honest, it's easy to see that these lists will match, in fact, that's why we have to use variables in the above list, otherwise the permutation *seems* to do nothing. But as can also be seen above, if the prover tries to

cheat by violating $a_1 = b_1$ then the permuted $\sigma(\boldsymbol{f})$ will not be equal to the original $\boldsymbol{f}$. As in the above section we can model the vectors as polynomials using FFT, such that $f(\omega^1) = f_1, f(\omega^2) = f_2 \ldots$.

Then, given the polynomial $f(X)$ we want to check whether:

$$\forall i \in [n] : f(\omega^i) \stackrel{?}{=} f(\omega^{\sigma(i)})$$

Where $n = |H|$. One approach is to use the **Grand Product Argument**, defined earlier, which would show:

$$\prod_{i=1}^{n} f(\omega^i) = \prod_{i=1}^{n} f(\omega^{\sigma(i)})$$

But this only proves there exists *some* permutation between $f(X)$ and itself, not necessarily σ. We can start by trying to model sets of pairs, rather than just sets:

$$\{(a_i, b_i) \mid i \in [1, n]\} = \{(c_i, d_i) \mid i \in [1, n]\}$$

Which can be modelled with:

$$f(X) = a_i(X) + \beta b_i(X), \quad g(X) = c_i(X) + \beta d_i(X)$$

Correctness:

We need to prove that:

$$(a(\omega^i), b(\omega^i)) = (c(\omega^i), d(\omega^i)) \implies a(\omega^i) + \beta b(\omega^i) = c(\omega^i) + \beta d(\omega^i)$$

Which holds trivially, since the LHS implies $a(\omega^i) = c(\omega^i), b(\omega^i) = d(\omega^i)$, meaning we can rewrite:

$$a(\omega^i) + \beta b(\omega^i) = a(\omega^i) + \beta b(\omega^i)$$

□

Soundness:

We need to prove that for a uniformly random β:

$$a(\omega^i) + \beta b(\omega^i) = c(\omega^i) + \beta d(\omega^i) \implies a(\omega^i) = c(\omega^i) \wedge b(\omega^i) = d(\omega^i)$$

Except with negligible probability. Assuming $a(\omega^i) + \beta b(\omega^i) \neq c(\omega^i) + \beta d(\omega^i) \wedge g(\omega^i)$:

$$\begin{aligned} a(\omega^i) + \beta b(\omega^i) = c(\omega^i) + \beta d(\omega^i) &\implies \\ a(\omega^i) - c(\omega^i) = \beta d(\omega^i) - \beta b(\omega^i) &\implies \\ a(\omega^i) - c(\omega^i) = \beta \cdot (d(\omega^i) - b(\omega^i)) &\implies \\ \beta = \frac{a(\omega^i) - c(\omega^i)}{d(\omega^i) - b(\omega^i)} & \end{aligned}$$

Since all the left-hand sides are constant and β is uniformly random in $\mathbb{F}$, there is a $1/|\mathbb{F}|$ probability that the claim doesn't hold. Thus, we have soundness.

□

The prover then wants to prove that for $i \in [n] : f_i = f_{\sigma(i)}$, for a specific permutation σ:

$$\{(f_i, i) \mid i \in [1, n]\} = \{(f_i, \sigma(i)) \mid i \in [1, n]\} \implies f_i = f_{\sigma_i} \implies \boldsymbol{f} = \sigma(\boldsymbol{f})$$

Which for polynomials:

$$\{(f(\omega^i), \text{id}(\omega^i)) \mid i \in [1,n]\} = \{(f(\omega^i), \sigma(\omega^i)) \mid i \in [1,n]\} \implies f(\omega^i) = f(\omega^{\sigma(i)})$$

So *now* we can use the **Grand Product Argument** to prove that $\forall i \in [n] : f(\omega^i) \stackrel{?}{=} f(\omega^{\sigma(i)})$, with:

$$f'(X) = f(X) + \beta \text{id}(X), \quad g'(X) = f(X) + \beta\sigma(X)$$

Example

An example, without soundness values β, γ, for why this approach to proving $\sigma(\boldsymbol{f}) = \boldsymbol{f}$ is sensible:

$$\begin{gathered}
\text{id}(1) = 1 \quad \text{id}(2) = 2 \quad \text{id}(3) = 3 \quad \text{id}(4) = 4 \quad \text{id}(5) = 5 \quad \text{id}(6) = 6 \\
\sigma(1) = 1 \quad \sigma(2) = 4 \quad \sigma(3) = 5 \quad \sigma(4) = 6 \quad \sigma(5) = 3 \quad \sigma(6) = 2
\end{gathered}$$

$$\begin{aligned}
\prod_{\omega \in H} (f(\omega) + \text{id}(\omega)) &= (f(\omega^1)+1)(f(\omega^2)+2)(f(\omega^3)+3)(f(\omega^4)+4)(f(\omega^5)+5)(f(\omega^6)+6) \\
\prod_{\omega \in H} (f(\omega) + \sigma(\omega)) &= (f(\omega^1)+1)(f(\omega^2)+4)(f(\omega^3)+5)(f(\omega^4)+6)(f(\omega^5)+3)(f(\omega^6)+2) \\
&= (f(\omega^1)+1)(f(\omega^4)+4)(f(\omega^5)+5)(f(\omega^6)+6)(f(\omega^3)+3)(f(\omega^2)+2) \\
&= (f(\omega^1)+1)(f(\omega^2)+2)(f(\omega^3)+3)(f(\omega^4)+4)(f(\omega^5)+5)(f(\omega^6)+6)
\end{aligned}$$

5.2.2.1 Permutation Argument Over Multiple Polynomials In Plonk, we don't have a single polynomial spanning over each $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$. Since the Grand Product Argument operates over products, we can define:

$$\begin{aligned}
f_a(X) &= a(X) + \text{id}(X)\beta + \gamma \\
f_b(X) &= b(X) + \text{id}(n+X)\beta + \gamma \\
f_c(X) &= c(X) + \text{id}(2n+X)\beta + \gamma \\
g_a(X) &= a(X) + \sigma(X)\beta + \gamma \\
g_b(X) &= b(X) + \sigma(n+X)\beta + \gamma \\
g_c(X) &= c(X) + \sigma(2n+X)\beta + \gamma \\
f(X) &= f_a(X) \cdot f_b(X) \cdot f_c(X) \\
g(X) &= g_a(X) \cdot g_b(X) \cdot g_c(X)
\end{aligned}$$

Which means that for our example circuit in Figure 4, we now get the table:

ω^i	a	b	c	q_l	q_r	q_o	q_m	q_c	id_a	id_b	id_c	σ_a	σ_b	σ_c
ω^1	3	0	0	1	0	0	0	-3	1	9	17	14	9	17
ω^2	5	0	0	1	0	0	0	-5	2	10	18	5	10	18
ω^3	47	0	0	1	0	0	0	-47	3	11	19	23	11	19
ω^4	2	2	4	0	0	-1	1	0	4	12	20	12	4	6
ω^5	5	7	35	0	0	-1	1	0	5	13	21	2	13	7
ω^6	4	3	12	0	0	-1	1	0	6	14	22	20	1	15
ω^7	35	12	47	1	1	-1	0	0	7	15	23	21	22	3
ω^8	0	0	0	0	0	0	0	0	8	16	24	8	16	24

5.2.3 Public Inputs

It might be useful to have public inputs for a circuit. This is not to be confused with constants in the circuits:

- *A constant* is always the value set by the circuit, and is public, known by both the prover and verifier.
- *A witness* is an input value to the circuit, and is private, known only by the prover.
- *A public input* is an input value to the circuit, and is public, known by both the prover and verifier.

We have a vector of public inputs:

$$\boldsymbol{x} : |\boldsymbol{x}| = \ell_2$$

Naturally leading to a polynomial:

$$x(X) = \text{ifft}(\boldsymbol{x})$$

The number of public inputs, ℓ_2, is embedded in the circuit specification. The first ℓ_2 rows of the witness table is reserved for public inputs. For each $x_i \in \boldsymbol{x}$, we set $q_l(\omega^i) = 1, a(\omega^i) = x_i$ and the rest of the witness and selector polynomials to zero. F_{GC} must then also be updated:

$$f_{GC}(X) = a(X)q_l(X) + b(X)q_r(X) + c(X)q_o(X) + a(X)b(X)q_m(X) + q_c(X) + x(X)$$

5.2.4 Input Passing

Since we use a cycle of curves, each language instruction is mapped to one of two circuits, verifying both circuits should convince the verifier that the program $f(w, x)$ is satisfied. However, for Elliptic Curve Multiplication and the Poseidon Hashes, we need to pass inputs from one circuit to another.

Passing $v^{(q)} \to v^{(p)}$:

We start with the simple case. We have a circuit over $\mathbb{F}_p$, $R^{(p)}$, and a circuit over $\mathbb{F}_q$, $R^{(q)}$, with $p > q$. We wish to pass a value, $v^{(q)} \in \mathbb{F}_q$, from $R^{(q)}$ to $R^{(p)}$. We can add $v^{(p)}$ to the public inputs to $R^{(q)}$, but then we still need to convince the verifier that $v^{(q)} = v^{(q)}$. Naively, the verifier could add the check that $v^{(q)} \stackrel{?}{=} v^{(p)}$. But this won't work for IVC, since we can't check equality across circuits, in-circuit. Instead we compute the commitment to $v^{(q)}$ on the $R^{(q)}$-side.

$$C_{\text{IP}}^{(q)} := v^{(q)} \cdot G_1^{(q)} \in \mathbb{E}_p(\mathbb{F}_q)$$

The scalar operation may seem invalid, but since we know that $v^{(q)} \leq q - 1 < p - 1$, it can logically be computed by the usual double and add, since the bits of $v^{(q)}$ will correspond to the bits of $v^{(p)}$ if $\text{lift}(v^{(q)}) = \text{lift}(v^{(p)})$, where $\text{lift} \in \mathbb{F} \to \mathbb{Z}$ is a function that returns the integer value of a finite field. If $C_{\text{IP}}^{(q)}$ is emitted in the public inputs of the circuit, then the verifier will know that $C_{\text{IP}}^{(q)}$ is a commitment to $v^{(q)}$. To convince the verifier of the desired relation that $\text{lift}(v^{(q)}) = \text{lift}(v^{(p)})$, it will now suffice to convince them that $v^{(p)}$ is a valid opening of $C_{\text{IP}}^{(q)}$. So the verifier checks manually that:

$$C_{\text{IP}}^{(q)} \stackrel{?}{=} v^{(p)} \cdot G_1^{(q)}$$

Which, given that the rest of the proof verifies correctly, will then imply that $v^{(q)} = v^{(p)}$. If the verifier is encoded as a circuit, then we need to input pass when performing this additional check, since scalar multiplication itself requires input passing to work. However this is no problem, since that circuit-verifier will be verified by another verifier! At some point, this deferral will end with a regular verifier, that can compute the commitment outside the circuit.

Passing $v^{(p)} \to v^{(q)}$:

What if we reverse the flow? We now have a value $v^{(p)}$, in $R^{(p)}$, that we want to pass to $R^{(q)}$. Here the problem is that since $p > q$, the value might be too large to represent in the $\mathbb{F}_q$-field. The solution is to decompose the value:

$$v_p^{(p)} = 2h^{(p)} + l^{(p)}$$

Where $h^{(p)}$ represents the high-bits of $v^{(p)}$ ($h^{(p)} \in [0, 2^{\lfloor \log p \rfloor}]$) and $l^{(p)}$ represents the low-bit ($h^{(p)} \in \mathbb{B}$). The value $v^{(p)}$ can now be represented with $h^{(p)}, l^{(p)}$, both of which are less than q. Which means we can pass the value to $R^{(q)}$.

The constraints added to $R^{(p)}$ then becomes:

- $C_{\text{IP}}^{(p)} \stackrel{?}{=} h \cdot G_1^{(p)} + l \cdot G_2^{(p)}$
- $v = 2h^{(p)} + l^{(p)}$
- $h \in [0, 2^{\lfloor \log p \rfloor}]$ (Using the rangecheck gate, corresponding to a range proof)

- $l \in \mathbb{B}$ (A Simple boolean constraint)

Combining Commitments:

We of course don't need to commit each time we pass inputs, we can create a standard vector Pedersen commit, containing all the passed values:

$$C_{\text{IP}}^{(p)} = h_{v_1}^{(p)} \cdot G_1^{(p)} + l_{v_1}^{(p)} \cdot G_2^{(p)} + h_{v_2}^{(p)} \cdot G_3^{(p)} + l_{v_2}^{(p)} \cdot G_4^{(p)} + \dots$$

Now, the R_q-verifier and R_p-verifier, would each also take in a single input pass vector, in addition to the standard public input vector:

$$\text{InputPass}^{(q \to p)} \in \mathbb{F}_p^k, \qquad \text{InputPass}^{(p \to q)} \in \mathbb{F}_q^k$$

Each passed input is of course public, so the public input vector is then defined as:

$$\text{PublicInputs}_{\text{new}}^{(p)} := \text{PublicInputs}^{(p)} \mathbin{+\!\!+} \text{InputPass}^{(p)}$$

For both the verifier and prover of course. Each of the $R^{(p)}$ and $R^{(q)}$ verifier can then use $\text{InputPass}^{(q \to p)}, \text{InputPass}^{(p \to q)}$ to verify $C_{\text{IP}}^{(p)}, C_{\text{IP}}^{(q)}$:

Example

Take the following example circuit:

Example Circuit

Inputs

$x, y \in \mathbb{F}_p$
$P \in \mathbb{E}_p(\mathbb{F}_q)$

1: $z := x + y \in \mathbb{F}_p$
2: $Q_1 := z \cdot P \in \mathbb{E}_p(\mathbb{F}_q)$
3: $Q_2 := x \cdot P \in \mathbb{E}_p(\mathbb{F}_q)$
4: $\alpha := \mathcal{H}(Q_1, Q_2) \in \mathbb{F}_p$

Which means that we pass z, x from $R^{(p)}$ to $R^{(q)}$ and α from $R^{(q)}$ to $R^{(p)}$. Thus, we need to split z, x but not α. We add the constraints:

- $R^{(p)}$:
  - $C_{\text{IP}}^{(p)} := h_z^{(p)} \cdot G_1^{(p)} + l_z^{(p)} \cdot G_2^{(p)} + h_x^{(p)} \cdot G_3^{(p)} + l_x^{(p)} \cdot G_4^{(p)}$
  - $z := 2 \cdot h_z^{(p)} + l_z^{(p)}$ (Decomposition correctness check)
  - $h_z^{(p)} \in [0, 2^{\lfloor \log p \rfloor}]$ (Range check)
  - $l_z^{(p)} \in \mathbb{B}$ (Boolean check)
- $R^{(q)}$:
  - $C_{\text{IP}}^{(q)} := \alpha^{(q)} \cdot G_1^{(q)}$

Now the R_q-verifier and R_p-verifier, would each also take in the input pass vectors:

$$\text{InputPass}^{(p \to q)} = [h_z^{(q)}, l_z^{(q)}, h_x^{(q)}, l_x^{(q)}]$$
$$\text{InputPass}^{(q \to p)} = [\alpha^{(p)}]$$

Each passed input is of course public, so the public input vector is then defined as:

$$\text{PublicInputs}_{\text{new}}^{(p)} := \text{PublicInputs}^{(p)} \mathbin{+\!\!+} \text{InputPass}^{(p)}$$

For both the verifier and prover of course. Now the verifier needs to verify what it otherwise would, but also that:

$$C_{\text{IP}}^{(p)} \stackrel{?}{=} h_z^{(q)} \cdot G_1^{(p)} + l_z^{(q)} \cdot G_2^{(p)} + h_x^{(q)} \cdot G_3^{(p)} + l_x^{(q)} \cdot G_4^{(p)}$$
$$C_{\text{IP}}^{(q)} \stackrel{?}{=} \alpha^{(p)} \cdot G_1^{(q)}$$

5.3 Custom Gates

For each gate, we have a *Witness Row*, *Selector Row* and a *Coefficient Row*. These rows describe the form of the constraints. Take the addition gate as an example:

Example - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
a	b	c	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Here the first row describes the 16 witness inputs associated for the gate. In this case only three witnesses are needed, so the other columns are set to 0. The Second row describes the copy constraints. In this case that means that slot 1 (corresponding to w_1), is copy constrained to the left input wire, slot 2 (corresponding to w_2) is copy constrained to the right input wire and slot 3 (corresponding to w_3) is copy constrained to the first, and for this gate; only, output wire.

Example - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
1	1	-1	0	0	0	0	0	0	0	0

This means that the $q_l = 1, q_r = 1, q_o = -1$ is set for this row. So that for this row f_{GC} becomes:

$$f_{GC} = q_l w_1 + q_r w_2 + q_o w_3 + \cdots = 0 \implies w_1 + w_2 - w_3 = 0$$

We also have a coefficient row for each gate, that can store constants for each gate. This is used in the scalar multiplication, Poseidon and range-check gates. For all other gates they are set to zero. If they are not listed in a gate specification below, then all row-values are set to zero. This is also the case for our example add gate.

Example - Coefficient Row

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Some of the more complicated gates will have their own designated selector polynomial and a table of constraints. The simplest one is the equals gate:

Example - Custom Constraints

Degree	Constraint	Meaning
3	$(x - y) \cdot b$	$x \neq y \implies b = 0$
3	$(x - y) \cdot \alpha + b - 1$	$x = y \implies b = 1$

It's implicit that each constraint in the constraint tables should always be equal to zero. To translate this into the form expected by $f_{GC}(X)$, we start by multiplying each row by the relevant designated selector polynomial, in this case it's $q_{(=)}(X)$. This also explains why the degree is three, not two, in the table. The values x, y, b, α can be read from the witness table from the equality gate, which in this case leads to $x = w_1(X), y = w_2(X), b = w_3(X), \alpha = w_4(X)$. Finally, a challenge (in this case ζ) multiplied to each row, creating a geometric sum. Taken together, this leads to adding the following terms to $f_{GC}(X)$ for the equality gate:

$$f_{GC}(X) = \cdots + q_{(=)}(X) \cdot (((w_1(X) - w_2(X)) \cdot w_3(X)) + \zeta((w_1(X) - w_2(X)) \cdot w_4(X) + w_3(X) - 1)) + \ldots$$
$$f_{GC}(X) = \ldots$$
$$+ \zeta^0 \cdot q_{(=)}(X) \cdot ((w_1(X) - w_2(X)) \cdot w_3(X))$$
$$+ \zeta^1 \cdot q_{(=)}(X) \cdot ((w_1(X) - w_2(X)) \cdot w_4(X) + w_3(X) - 1))$$
$$+ \ldots$$

Of course, if this gate had more rows in the constraint table, the next term would be multiplied with ζ^2 and so on. If a constraint uses exponentiation, for example, if a row states x^2 and $x = w_1(X)$ then that simply means $w_1(X) \cdot w_1(X)$ as one would expect.

5.3.1 Field

5.3.1.1 Addition, Subtraction, Multiplication, Negation For completeness, we include the witness tables for field addition, subtraction, multiplication and negation even though they are part of vanilla Plonk as originally defined in the Plonk paper[Gabizon et al. 2019]:

Addition:

Field Addition - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
a	b	c	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Field Addition - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
1	1	-1	0	0	0	0	0	0	0	0

Subtraction:

Field Subtraction - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
a	b	c	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Field Subtraction - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
1	-1	-1	0	0	0	0	0	0	0	0

Negation can be modelled as $-a = 0 - a$, a dedicated negation gate would also be one row anyways.

Multiplication:

Field Multiplication - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
a	b	c	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Field Multiplication - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	-1	1	0	0	0	0	0	0	0

Inverse:

To model inverses, we can witness the inverse of x, x^{-1}, and constrain that $x \cdot x^{-1} = 1$:

Field Inverse - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x	x^{-1}	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Field Inverse - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	1	1	0	0	0	0	0	0

Assert Equals:

We can also create a gate that asserts equality between two values x and y:

Field Inverse - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x	y	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Field Inverse - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
1	-1	0	0	0	0	0	0	0	0	0

5.3.2 Booleans

5.3.2.1 Witness Boolean To witness a boolean, we need to constrain that the witnessed value indeed is a bit:

$$(b \cdot b) - b = 0$$

This can be modelled using the vanilla Plonk selector polynomials.

Witness Boolean - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
b	b	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_1	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Witness Boolean - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
-1	0	0	1	0	0	0	0	0	0	0

5.3.2.2 Equality To check whether two values are equal, $b = x \stackrel{?}{=} y$, we need to witness b and $\text{inv0}(x - y)$:

$$b = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{inv0}(x) = \begin{cases} x^{-1} & \text{if } x \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\alpha = \text{inv0}(x - y)$$

Now, we want that $x = y \implies b = 1$ and $a \neq b \implies \alpha = 0$.

Equality - Custom Constraints

Degree	Constraint	Meaning
3	$(x - y) \cdot b$	$x \neq y \implies b = 0$
3	$(x - y) \cdot \alpha + b - 1$	$x = y \implies b = 1$

Equality - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x	y	b	α	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Equality - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	0	0	0	1	0

Completeness:

Case $x \neq y \wedge b = 0$:

$$(x - y) \cdot 0 = 0$$

$$(x - y) \cdot (x - y)^{-1} + 0 - 1 = 1 - 1 = 0$$

Case $x = y \wedge b = 1$:

$$(x - y) \cdot 1 = 0 \cdot 1 = 0$$

$$(x - y) \cdot 0 + 1 - 1 = 1 - 1 = 0$$

Soundness:

The first constraint is trivial. For the second constraint:

Case $x \neq y$:

The first constraint ensures that $b = 0$ in this case:

$$\begin{aligned} (x - y) \cdot \alpha + 0 - 1 = 0 &\implies \\ (x - y) \cdot \alpha = 1 & \end{aligned}$$

Case $x = y$:

$$\begin{aligned} (x - y) \cdot \alpha + b - 1 = 0 &\implies \\ 0 \cdot \alpha + b - 1 = 0 &\implies \\ b = 1 & \end{aligned}$$

5.3.2.3 And, Or To implement "And" for two booleans, x, y, we can simply multiply them, costing a single row. Because x, y are constrained to be bits, when they are input, the output is also guaranteed to be a bit. To implement "Or", we can compose the following constraint that $c = x \vee y = x + y - (x \cdot y)$. To see why it works, given that x, y are already constrained to be bits:

x	y	Out
0	0	$0 + 0 - (0 \cdot 0) = 0$
0	1	$0 + 1 - (0 \cdot 1) = 1$
1	0	$1 + 0 - (1 \cdot 0) = 1$
1	1	$1 + 1 - (1 \cdot 1) = 1$

This can naively be done in three rows, but we can compress it to a single row as $0 = x + y - c - (x \cdot y)$:

Or - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
a	b	c	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	I_2	O_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Or - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
1	1	-1	-1	0	0	0	0	0	0	0

5.3.3 Rangecheck

We want to constrain $x \in [0, 2^{254})$. We decompose x into 254 bits and check that:

$$x = \sum_{i=0}^{253} b_i \cdot 2^i$$

The entire range-check then consists of $254/15 = 17$ rows:

Rangecheck - Witness Table

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
acc_0	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}
acc_1	b_{15}	b_{16}	b_{17}	b_{18}	b_{19}	b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}	b_{26}	b_{27}	b_{28}	b_{29}
acc_2	b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}	b_{36}	b_{37}	b_{38}	b_{39}	b_{40}	b_{41}	b_{42}	b_{43}	b_{44}
acc_3	b_{45}	b_{46}	b_{47}	b_{48}	b_{49}	b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}	b_{56}	b_{57}	b_{58}	b_{59}
acc_4	b_{60}	b_{61}	b_{62}	b_{63}	b_{64}	b_{65}	b_{66}	b_{67}	b_{68}	b_{69}	b_{70}	b_{71}	b_{72}	b_{73}	b_{74}
acc_5	b_{75}	b_{76}	b_{77}	b_{78}	b_{79}	b_{80}	b_{81}	b_{82}	b_{83}	b_{84}	b_{85}	b_{86}	b_{87}	b_{88}	b_{89}
acc_6	b_{90}	b_{91}	b_{92}	b_{93}	b_{94}	b_{95}	b_{96}	b_{97}	b_{98}	b_{99}	b_{100}	b_{101}	b_{102}	b_{103}	b_{104}
acc_7	b_{105}	b_{106}	b_{107}	b_{108}	b_{109}	b_{110}	b_{111}	b_{112}	b_{113}	b_{114}	b_{115}	b_{116}	b_{117}	b_{118}	b_{119}
acc_8	b_{120}	b_{121}	b_{122}	b_{123}	b_{124}	b_{125}	b_{126}	b_{127}	b_{128}	b_{129}	b_{130}	b_{131}	b_{132}	b_{133}	b_{134}
acc_9	b_{135}	b_{136}	b_{137}	b_{138}	b_{139}	b_{140}	b_{141}	b_{142}	b_{143}	b_{144}	b_{145}	b_{146}	b_{147}	b_{148}	b_{149}
acc_{10}	b_{150}	b_{151}	b_{152}	b_{153}	b_{154}	b_{155}	b_{156}	b_{157}	b_{158}	b_{159}	b_{160}	b_{161}	b_{162}	b_{163}	b_{164}
acc_{11}	b_{165}	b_{166}	b_{167}	b_{168}	b_{169}	b_{170}	b_{171}	b_{172}	b_{173}	b_{174}	b_{175}	b_{176}	b_{177}	b_{178}	b_{179}
acc_{12}	b_{180}	b_{181}	b_{182}	b_{183}	b_{184}	b_{185}	b_{186}	b_{187}	b_{188}	b_{189}	b_{190}	b_{191}	b_{192}	b_{193}	b_{194}
acc_{13}	b_{195}	b_{196}	b_{197}	b_{198}	b_{199}	b_{200}	b_{201}	b_{202}	b_{203}	b_{204}	b_{205}	b_{206}	b_{207}	b_{208}	b_{209}
acc_{14}	b_{210}	b_{211}	b_{212}	b_{213}	b_{214}	b_{215}	b_{216}	b_{217}	b_{218}	b_{219}	b_{220}	b_{221}	b_{222}	b_{223}	b_{224}
acc_{15}	b_{225}	b_{226}	b_{227}	b_{228}	b_{229}	b_{230}	b_{231}	b_{232}	b_{233}	b_{234}	b_{235}	b_{236}	b_{237}	b_{238}	b_{239}
acc_{16}	b_{240}	b_{241}	b_{242}	b_{243}	b_{244}	b_{245}	b_{246}	b_{247}	b_{248}	b_{249}	b_{250}	b_{251}	b_{252}	b_{253}	0
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

The last row still indicates the copy constraints[4]. Each acc_i is the accumulation of all previously witnessed bits, so:

$$
\begin{aligned}
acc_0 &= 0 \\
acc_1 &= \sum_{i=0}^{14} b_i \cdot 2^i \\
acc_2 &= \sum_{i=0}^{29} b_i \cdot 2^i \\
acc_3 &= \dots
\end{aligned}
$$

However, for this, we still need to witness each power of two. Luckily, these are constant and fixed in the circuit specification, so we can use the coefficient table for this:

Rangecheck - Coefficient Table

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}
2^{30}	2^{31}	2^{32}	2^{33}	2^{34}	2^{35}	2^{36}	2^{37}	2^{38}	2^{39}	2^{40}	2^{41}	2^{42}	2^{43}	2^{44}
2^{45}	2^{46}	2^{47}	2^{48}	2^{49}	2^{50}	2^{51}	2^{52}	2^{53}	2^{54}	2^{55}	2^{56}	2^{57}	2^{58}	2^{59}
2^{60}	2^{61}	2^{62}	2^{63}	2^{64}	2^{65}	2^{66}	2^{67}	2^{68}	2^{69}	2^{70}	2^{71}	2^{72}	2^{73}	2^{74}
2^{75}	2^{76}	2^{77}	2^{78}	2^{79}	2^{80}	2^{81}	2^{82}	2^{83}	2^{84}	2^{85}	2^{86}	2^{87}	2^{88}	2^{89}
2^{90}	2^{91}	2^{92}	2^{93}	2^{94}	2^{95}	2^{96}	2^{97}	2^{98}	2^{99}	2^{100}	2^{101}	2^{102}	2^{103}	2^{104}
2^{105}	2^{106}	2^{107}	2^{108}	2^{109}	2^{110}	2^{111}	2^{112}	2^{113}	2^{114}	2^{115}	2^{116}	2^{117}	2^{118}	2^{119}
2^{120}	2^{121}	2^{122}	2^{123}	2^{124}	2^{125}	2^{126}	2^{127}	2^{128}	2^{129}	2^{130}	2^{131}	2^{132}	2^{133}	2^{134}
2^{135}	2^{136}	2^{137}	2^{138}	2^{139}	2^{140}	2^{141}	2^{142}	2^{143}	2^{144}	2^{145}	2^{146}	2^{147}	2^{148}	2^{149}
2^{150}	2^{151}	2^{152}	2^{153}	2^{154}	2^{155}	2^{156}	2^{157}	2^{158}	2^{159}	2^{160}	2^{161}	2^{162}	2^{163}	2^{164}
2^{165}	2^{166}	2^{167}	2^{168}	2^{169}	2^{170}	2^{171}	2^{172}	2^{173}	2^{174}	2^{175}	2^{176}	2^{177}	2^{178}	2^{179}
2^{180}	2^{181}	2^{182}	2^{183}	2^{184}	2^{185}	2^{186}	2^{187}	2^{188}	2^{189}	2^{190}	2^{191}	2^{192}	2^{193}	2^{194}
2^{195}	2^{196}	2^{197}	2^{198}	2^{199}	2^{200}	2^{201}	2^{202}	2^{203}	2^{204}	2^{205}	2^{206}	2^{207}	2^{208}	2^{209}
2^{210}	2^{211}	2^{212}	2^{213}	2^{214}	2^{215}	2^{216}	2^{217}	2^{218}	2^{219}	2^{220}	2^{221}	2^{222}	2^{223}	2^{224}
2^{225}	2^{226}	2^{227}	2^{228}	2^{229}	2^{230}	2^{231}	2^{232}	2^{233}	2^{234}	2^{235}	2^{236}	2^{237}	2^{238}	2^{239}
2^{240}	2^{241}	2^{242}	2^{243}	2^{244}	2^{245}	2^{246}	2^{247}	2^{248}	2^{249}	2^{250}	2^{251}	2^{252}	2^{253}	0

Now for the constraints, for each *row* in the tables above:

[4]Except, this table doesn't capture the fact that acc_0 needs to be constrained to a zero constant value in the circuit! All other rows does not have any copy constraints associated with it.

Range Check - Custom Constraints

Degree	Constraint	Meaning
3	$b_{(i\cdot 15+0)} \cdot (b_{(i\cdot 15+0)} - 1)$	$b_{(i\cdot 15+0)}, \dots b_{(i\cdot 15+14)} \in \mathbb{B}$
3	$b_{(i\cdot 15+1)} \cdot (b_{(i\cdot 15+1)} - 1)$	
3	$b_{(i\cdot 15+2)} \cdot (b_{(i\cdot 15+2)} - 1)$	
3	$b_{(i\cdot 15+3)} \cdot (b_{(i\cdot 15+3)} - 1)$	
3	$b_{(i\cdot 15+4)} \cdot (b_{(i\cdot 15+4)} - 1)$	
3	$b_{(i\cdot 15+5)} \cdot (b_{(i\cdot 15+5)} - 1)$	
3	$b_{(i\cdot 15+6)} \cdot (b_{(i\cdot 15+6)} - 1)$	
3	$b_{(i\cdot 15+7)} \cdot (b_{(i\cdot 15+7)} - 1)$	
3	$b_{(i\cdot 15+8)} \cdot (b_{(i\cdot 15+8)} - 1)$	
3	$b_{(i\cdot 15+9)} \cdot (b_{(i\cdot 15+9)} - 1)$	
3	$b_{(i\cdot 15+11)} \cdot (b_{(i\cdot 15+11)} - 1)$	
3	$b_{(i\cdot 15+12)} \cdot (b_{(i\cdot 15+12)} - 1)$	
3	$b_{(i\cdot 15+13)} \cdot (b_{(i\cdot 15+13)} - 1)$	
3	$b_{(i\cdot 15+14)} \cdot (b_{(i\cdot 15+14)} - 1)$	
2	acc_{i+1}	$acc_{i+1} = acc_i + \sum_{j=0}^{14} b_{(i\cdot 15+j)}$
2	$-acc_i$	
3	$-(b_{(i\cdot 15+0)} \cdot 2^{(i\cdot 15+0)})$	
3	$-(b_{(i\cdot 15+1)} \cdot 2^{(i\cdot 15+1)})$	
3	$-(b_{(i\cdot 15+2)} \cdot 2^{(i\cdot 15+2)})$	
3	$-(b_{(i\cdot 15+3)} \cdot 2^{(i\cdot 15+3)})$	
3	$-(b_{(i\cdot 15+4)} \cdot 2^{(i\cdot 15+4)})$	
3	$-(b_{(i\cdot 15+5)} \cdot 2^{(i\cdot 15+5)})$	
3	$-(b_{(i\cdot 15+6)} \cdot 2^{(i\cdot 15+6)})$	
3	$-(b_{(i\cdot 15+7)} \cdot 2^{(i\cdot 15+7)})$	
3	$-(b_{(i\cdot 15+8)} \cdot 2^{(i\cdot 15+8)})$	
3	$-(b_{(i\cdot 15+9)} \cdot 2^{(i\cdot 15+9)})$	
3	$-(b_{(i\cdot 15+10)} \cdot 2^{(i\cdot 15+10)})$	
3	$-(b_{(i\cdot 15+11)} \cdot 2^{(i\cdot 15+11)})$	
3	$-(b_{(i\cdot 15+12)} \cdot 2^{(i\cdot 15+12)})$	
3	$-(b_{(i\cdot 15+13)} \cdot 2^{(i\cdot 15+13)})$	
3	$-(b_{(i\cdot 15+14)} \cdot 2^{(i\cdot 15+14)})$	

However, notice that we reference the next *acc* (acc_{i+1}) in the constraints, but this can be modelled as $w_1(\omega X)$. Each of the powers of two are available to the prover and verifier in the coefficient table. Note that this fact also means that in the constraints like $-(b_{(i\cdot 15+0)} \cdot 2^{(i\cdot 15+0)})$, $2^{(i\cdot 15+0)}$ is from the coefficient table, meaning that $2^{(i\cdot 15+0)} = r_1$, which also increases the degree from two to three.

Analysis:

- Bit constraints: The bit constraints follow the previously defined bit constraint. So from the first series we get: $b_{(i\cdot 15+0)}, \dots b_{(i\cdot 15+14)} \in \mathbb{B}$
- acc_{i+1} constraints: We want to capture that $x = \sum_{i=0}^{253} b_i \cdot 2^i$. We can make this hold if $acc_0 = 0$ (We can just copy-constrain acc_0 to a zero-constant in the circuit) and $acc_{i+1} = acc_i + \sum_{j=0}^{14} b_{(i\cdot 15+j)}$. But taken together, this is exactly what these bottom-half constraints state, giving us:
  $acc_{i+1} = acc_i + \sum_{j=0}^{14} b_{(i\cdot 15+j)} \implies acc_{17} = x = \sum_{i=0}^{253} b_i \cdot 2^i$

Finally, due to the fact that we store the sum in the *next* row, we need a single zero row to capture the result of the sum:

Rangecheck (Zero Row) - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
acc_{17}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Rangecheck (Zero Row) - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	0	0	0	0	0

We copy constrain acc_{17} to I_1 to indicate that $x = acc_{17}$ must hold. Since we only copy constrain I_1, a rangecheck can be viewed as a gate with one input and zero outputs.

5.3.4 Poseidon

We also create a special gate type for performing Poseidon[Grassi et al. 2019] hashing. This gate type is inspired from an equivalent gate in Mina's Kimchi proof system. At the heart of the Poseidon hashing algorithm lies a cryptographic sponge construction, like the one seen in SHA3. This is very convenient for Fiat-Shamir transformations, since information sent to the verifier can cleanly be modelled as sponge absorption, and queries made to the verifier can be modelled as sponge squeezing. Squeezing and absorbing from the sponge a certain number of times, triggers a permutation of the sponge state. The original Poseidon paper provide several small variations on how this permutation can be performed, with a variable number of partial and full rounds of permutation. Kimchi's approach to this is to only perform the expensive full rounds, but conversely make a highly specialized gate for only these full rounds.

A complete permutation of the Poseidon sponge state of size 3, then consists of 55 full rounds of the following computation:

$$\begin{aligned} \boldsymbol{s_i} &= [s_{i,0}, s_{i,1}, s_{i,2}]^\top \\ \text{sbox}(x) &= x^7 \\ \boldsymbol{s_{i+1}} &= \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_i})) + [r_{i,0}, r_{i,1}, r_{i,2}]^\top \end{aligned}$$

$\boldsymbol{M} \in \mathbb{F}^{(3,3)}$ represents the constant MDS matrix, and r_i represents the 55 round constants. Both of these were extracted from Kimchi's code, as we wanted our hash to have the same behaviour and therefore security. If we split this computation:

$$\begin{aligned} s'_0 &= M_{0,0} \cdot s_0^7 + M_{0,1} \cdot s_0^7 + M_{0,2} \cdot s_0^7 + r_{i,0} \\ s'_1 &= M_{1,0} \cdot s_1^7 + M_{1,1} \cdot s_1^7 + M_{1,2} \cdot s_1^7 + r_{i,1} \\ s'_2 &= M_{2,0} \cdot s_2^7 + M_{2,1} \cdot s_2^7 + M_{2,2} \cdot s_2^7 + r_{i,2} \end{aligned}$$

Leading us to the constraints:

Poseidon - Custom Constraints

Degree	Constraint	Meaning
8	$s_{1,0} - M_{0,0} \cdot s_{0,0}^7 + M_{0,1} \cdot s_{0,0}^7 + M_{0,2} \cdot s_{0,0}^7 + r_{0,0}$	
8	$s_{1,1} - M_{1,0} \cdot s_{0,1}^7 + M_{1,1} \cdot s_{0,1}^7 + M_{1,2} \cdot s_{0,1}^7 + r_{0,1}$	$\boldsymbol{s_1} = \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_0})) + [r_{0,0}, r_{0,1}, r_{0,2}]^\top$
8	$s_{1,2} - M_{2,0} \cdot s_{0,2}^7 + M_{2,1} \cdot s_{0,2}^7 + M_{2,2} \cdot s_{0,2}^7 + r_{0,2}$	
8	$s_{2,0} - M_{0,0} \cdot s_{1,0}^7 + M_{0,1} \cdot s_{1,0}^7 + M_{0,2} \cdot s_{1,0}^7 + r_{1,0}$	
8	$s_{2,1} - M_{1,0} \cdot s_{1,1}^7 + M_{1,1} \cdot s_{1,1}^7 + M_{1,2} \cdot s_{1,1}^7 + r_{1,1}$	$\boldsymbol{s_2} = \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_1})) + [r_{1,0}, r_{1,1}, r_{1,2}]^\top$
8	$s_{2,2} - M_{2,0} \cdot s_{1,2}^7 + M_{2,1} \cdot s_{1,2}^7 + M_{2,2} \cdot s_{1,2}^7 + r_{1,2}$	
8	$s_{3,0} - M_{0,0} \cdot s_{2,0}^7 + M_{0,1} \cdot s_{2,0}^7 + M_{0,2} \cdot s_{2,0}^7 + r_{2,0}$	
8	$s_{3,1} - M_{1,0} \cdot s_{2,1}^7 + M_{1,1} \cdot s_{2,1}^7 + M_{1,2} \cdot s_{2,1}^7 + r_{2,1}$	$\boldsymbol{s_3} = \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_2})) + [r_{2,0}, r_{2,1}, r_{2,2}]^\top$
8	$s_{3,2} - M_{2,0} \cdot s_{2,2}^7 + M_{2,1} \cdot s_{2,2}^7 + M_{2,2} \cdot s_{2,2}^7 + r_{2,2}$	
8	$s_{4,0} - M_{0,0} \cdot s_{3,0}^7 + M_{0,1} \cdot s_{3,0}^7 + M_{0,2} \cdot s_{3,0}^7 + r_{3,0}$	
8	$s_{4,1} - M_{1,0} \cdot s_{3,1}^7 + M_{1,1} \cdot s_{3,1}^7 + M_{1,2} \cdot s_{3,1}^7 + r_{3,1}$	$\boldsymbol{s_4} = \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_3})) + [r_{3,0}, r_{3,1}, r_{3,2}]^\top$
8	$s_{4,2} - M_{2,0} \cdot s_{3,2}^7 + M_{2,1} \cdot s_{3,2}^7 + M_{2,2} \cdot s_{3,2}^7 + r_{3,2}$	
8	$s_{5,0} - M_{0,0} \cdot s_{4,0}^7 + M_{0,1} \cdot s_{4,0}^7 + M_{0,2} \cdot s_{4,0}^7 + r_{4,0}$	
8	$s_{5,1} - M_{1,0} \cdot s_{4,1}^7 + M_{1,1} \cdot s_{4,1}^7 + M_{1,2} \cdot s_{4,1}^7 + r_{4,1}$	$\boldsymbol{s_5} = \boldsymbol{M} \cdot (\text{sbox}(\boldsymbol{s_4})) + [r_{4,0}, r_{4,1}, r_{4,2}]^\top$
8	$s_{5,2} - M_{2,0} \cdot s_{4,2}^7 + M_{2,1} \cdot s_{4,2}^7 + M_{2,2} \cdot s_{4,2}^7 + r_{4,2}$	

For the first row:

Poseidon - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{4,0}$	$s_{4,1}$	$s_{4,2}$	0
I_1	I_2	I_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Poseidon - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	1	0	0	0	0	0

This is missing the last 3 states, but these are used in the next five rounds of the permutation, so you can add these constraints, witnesses and selector polynomials 10 more times to complete the permutation[5]. Finally, a zero-row can be added to store the final state after 55 rounds (11 times the above gates):

Poseidon (Zero Row) - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
$s_{55,0}$	$s_{55,1}$	$s_{55,2}$	0	0	0	0	0	0	0	0	0	0	0	0	0
O_1	O_2	O_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Poseidon (Zero Row) - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	1	0	0	0	0	0

5.3.5 Elliptic Curves

5.3.5.1 Witness Point Points are represented in Affine Form, and the identity point is represented as $\mathcal{O} = (0, 0)$. 0 is not a valid x-coordinate of a valid point, because we need the curve equation to hold ($y^2 = x^3 + 5$), this is not possible since 5 is not square in $\mathbb{F}_p$ and 0 is not a y-coordinate in a valid point since -5 is not a cube in $\mathbb{F}_q$.

[5]Obviously, for the next 10 rounds there is no copy constraints.

To witness a point, we have to constrain that the witnessed point is on the curve. For the Pallas/Vesta curves used we have the curve equation. So we need constraints that encodes that $x \neq 0 \wedge y \neq 0 \implies y^2 - x^3 - 5 = 0$:

Witness Point - Custom Constraints

Degree	Constraint	Meaning
5	$x \cdot (y^2 - x^3 - 5) = 0$	$x \neq 0 \implies (y^2 - x^3 - 5) = 0$
5	$y \cdot (y^2 - x^3 - 5) = 0$	$y \neq 0 \implies (y^2 - x^3 - 5) = 0$

Witness Point - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x	y	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O_1	O_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Witness Point - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	1	0	0	0	0

Soundness and completeness hold trivially.

5.3.5.2 Point Addition In the constraints, we use a trick similar to the one used in the equality gate, where we model the condition $x = 0 \implies y = z$ by using the constraint $(1 - x \cdot \text{inv0}(x)) \cdot y - z = 0$. When $x = 0$:

$$\begin{aligned} 0 &= (1 - x \cdot \text{inv0}(x)) \cdot y - z \\ 0 &= (1 - 0) \cdot y - z \\ 0 &= y - z \end{aligned}$$

The inverse is there for correctness, so we don't constrain the $y - z$ when $x \neq 0$.

We witness:

$$\begin{aligned} \alpha &= \text{inv0}(x_q - x_p) \\ \beta &= \text{inv0}(x_p) \\ \gamma &= \text{inv0}(x_q) \\ \delta &= \begin{cases} \text{inv0}(y_q + y_p), & \text{if } x_q = x_p, \\ 0, & \text{otherwise.} \end{cases} \\ \lambda &= \begin{cases} \frac{y_q - y_p}{x_q - x_p}, & \text{if } x_q \neq x_p, \\ \frac{3x_p^2}{2y_p}, & \text{if } x_q \neq x_p, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Where:

$$\text{inv0}(x) = \begin{cases} 0, & \text{if } x = 0, \\ 1/x, & \text{otherwise.} \end{cases}$$

Point Addition - Custom Constraints

Degree	Constraint	Meaning
4	$(x_q - x_p) \cdot ((x_q - x_p) \cdot \lambda - (y_q - y_p))$	$x_q \neq x_p \implies \lambda = \frac{y_q - y_p}{x_q - x_p}$
5	$(1 - (x_q - x_p) \cdot \alpha) \cdot (2y_p \cdot \lambda - 3x_p^2)$	$x_q = x_p \wedge y_p \neq 0 \implies \lambda = \frac{3x_p^2}{2y_p}$ $x_q = x_p \wedge y_p = 0 \implies x_p = 0$
6	$(x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda^2 - x_p - x_q - x_r)$	$x_p \neq 0 \wedge x_q \neq 0 \wedge x_q \neq x_p \implies x_r = \lambda^2 - x_p - x_q$
6	$(x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r)$	$x_p \neq 0 \wedge x_q \neq 0 \wedge x_q \neq x_p \implies y_r = \lambda \cdot (x_p - x_q) - y_p$
6	$(x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda^2 - x_p - x_q - x_r)$	$x_p \neq 0 \wedge x_q \neq 0 \wedge y_q \neq -y_p \implies x_r = \lambda^2 - x_p - x_q$
6	$(x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r)$	$x_p \neq 0 \wedge x_q \neq 0 \wedge y_q \neq -y_p \implies y_r = \lambda \cdot (x_p - x_q) - y_p$
4	$(1 - x_p \cdot \beta) \cdot (x_r - x_q)$	$x_p = 0 \implies x_r = x_q$
4	$(1 - x_p \cdot \beta) \cdot (y_r - y_q)$	$x_p = 0 \implies y_r = y_q$
4	$(1 - x_q \cdot \beta) \cdot (x_r - x_p)$	$x_q = 0 \implies x_r = x_p$
4	$(1 - x_q \cdot \beta) \cdot (y_r - y_p)$	$x_q = 0 \implies y_r = y_p$
4	$(1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot x_r$	$x_q = x_p \wedge y_q = -y_p \implies x_r = 0$
4	$(1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot y_r$	$x_q = x_p \wedge y_q = -y_p \implies y_r = 0$

Point Addition - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_p	y_p	x_q	y_q	x_r	y_r	α	β	γ	δ	λ	0	0	0	0	0
I_1	I_2	I_3	I_4	O_1	O_2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Point Addition - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	0	1	0	0	0

We prove soundness and completeness of these constraints in the appendix.

5.3.5.3 Scalar Multiplication We follow a standard double-and-add scalar multiplication algorithm. It's specified below. It may seem a bit odd, but it's just to better relate to the eventual constraints.

Double-and-Add Scalar Multiplication

Inputs

x	The scalar.
P	The point to scale.

Output

A	$A = x \cdot P$

1: Let $\boldsymbol{b}$ be the bits of x, from LSB to MSB.
2: Let $A = \mathcal{O}$.
3: Let $acc = 0$.
4: **for** $i \in (255, 0]$ **do**
5: $\quad acc \mathrel{+}= b_i \cdot 2^i$
6: $\quad Q := 2A$
7: $\quad R := P + Q$
8: $\quad S :=$ **if** $b_i = 1$ **then** R **else** Q
9: $\quad A := S$
10: **end for**
11: **assert** $acc \stackrel{?}{=} x$
12: **return** A

There is three points to compute, Q, R, S, we start by constraining the doubling.

- $A = \mathcal{O} \implies Q = \mathcal{O}$:
- $A \neq \mathcal{O} \implies Q = 2A$:

Standard doubling dictates that to compute the doubling of $A = (x_a, y_a)$, $2A = Q = (x_q, y_q)$:

$$\begin{aligned} \lambda &= \frac{3x_a^2}{2y_a} \\ x_q &= \lambda_q^2 - 2 \cdot x_a \\ y_q &= \lambda_q \cdot (x_a - x_q) - y_a \end{aligned}$$

Except when $A = \mathcal{O}$, then becomes $Q = \mathcal{O}$. From this we can derive the constraints. Witness:

$$\begin{aligned} \gamma_q &= \text{inv0}(x_a) \\ \lambda_q &= \begin{cases} \frac{3x_a^2}{2y_a}, & \text{if } A \neq \mathcal{O}, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

And add the following constraints:

Point Doubling - Custom Constraints

Degree	Constraint	Meaning
4	$(1 - x_a \cdot \gamma_q) \cdot x_q$	$x_a = 0 \implies x_q = 0$
4	$(1 - x_a \cdot \gamma_q) \cdot y_q$	$x_a = 0 \implies y_q = 0$
3	$(2 \cdot y_a \cdot \lambda_q - 3 \cdot x_a^2)$	$\lambda_q = \frac{3x_a^2}{2y_a}$
3	$(\lambda_q^2 - 2 \cdot x_a - x_q)$	$x_q = \lambda_q^2 - 2 \cdot x_a$
3	$(\lambda_q \cdot (x_a - x_q) - y_a - y_q)$	$y_q = \lambda_q \cdot (x_a - x_q) - y_a$

Propositions:

1. $x_a = 0 \implies (x_q, y_q) = (0, 0)$
2. $\lambda_q = \frac{3x_a^2}{2y_a}$
3. $x_q = \lambda_q^2 - 2 \cdot x_a$

4. $y_q = \lambda_q \cdot (x_a - x_q) - y_a$

Cases:

- $A = (0,0) = \mathcal{O}$:
  - Completeness:
    1. Holds because $(x_a, y_a) = (x_q, y_q) = (0,0)$
    2. Holds because $0 = \lambda_q = x_a = y_a$
    3. Holds because $0 = x_q = \lambda_q = x_a$
    4. Holds because $0 = y_q = \lambda_q = x_a = x_q = y_a$
  - Soundness: $(x_r, y_r) = (0,0)$ is the only solution to 1.
- $A = (x_a, y_a) \neq \mathcal{O}$:
  - Completeness:
    (1) Holds because $x_a \neq 0$
    (2) Holds because $\lambda_q = 3x_p^2/2y_p$
    (3) Holds because $x_q = \lambda_q^2 - 2 \cdot x_a$
    (4) Holds because $y_q = \lambda_q \cdot (x_a - x_q) - y_a$
  - Soundness:
    Firstly, (2) states that λ_q is computed correctly, (3) states that x_q is computed correctly, (4) states that y_q is computed correctly. Thus, $Q = 2A$

From here, we simply add the previous point addition constraints, with one small change; we already have γ_q, which is used to check whether A is the identity point. However, since $A = \mathcal{O} \implies Q = \mathcal{O}$, we can replace the γ from the point addition constraints with the already witnessed γ_q. Then, soundness and completeness follow trivially from the previous section. Finally, we need to create constraints for $S =$ **if** $b_1 = 1$ **then** R **else** Q.

Ternary Point - Custom Constraints

Degree	Constraint	Meaning
3	$b_i \cdot (b_i - 1)$	$b_i \in \mathbb{B}$
3	$x_s - (b_i \cdot x_r + (1 - b_i) \cdot x_q)$	$x_s =$ **if** $b_i = 1$ **then** x_r **else** x_q
3	$y_s - (b_i \cdot y_r + (1 - b_i) \cdot y_q)$	$y_s =$ **if** $b_i = 1$ **then** y_r **else** y_q
3	$acc_{i+1} - (acc_i + b_i \cdot 2^i)$	$acc_{i+1} - (acc_i + b_i \cdot 2^i)$

Propositions:

(1) $b_i \in \mathbb{B}$ Standard bit constraint
(2) $S =$ **if** $b_i = 1$ **then** R **else** Q $x_s - (b_i \cdot x_r + (1 - b_i) \cdot x_q) \implies x_s = (b_i \cdot x_r + (1 - b_i) \cdot x_q)$
- $b_i = 0$: $x_s = (0 \cdot x_r + (1 - 0) \cdot x_q) \implies x_s = x_r$
- $b_i = 1$: $x_s = (1 \cdot x_r + (1 - 1) \cdot x_q) \implies x_s = x_r$

Since this also holds for y_s, y_r, y_q: $S =$ **if** $b_i = 1$ **then** R **else** Q
(3) $acc_{i+1} - (acc_i + b_i \cdot 2^i)$ $acc_{i+1} = (acc_i + b_i \cdot 2^i)$

So, for each iteration of the loop, we witness:

$$\gamma_q = \text{inv0}(x_a)$$
$$\lambda_q = \begin{cases} \frac{3x_a^2}{2y_a}, & \text{if } A \neq \mathcal{O}, \\ 0, & \text{otherwise.} \end{cases}$$
$$\alpha_r = \text{inv0}(x_q - x_p)$$
$$\beta_r = \text{inv0}(x_p)$$
$$\delta_r = \begin{cases} \text{inv0}(y_q + y_p), & \text{if } x_q = x_p, \\ 0, & \text{otherwise.} \end{cases}$$
$$\lambda_r = \begin{cases} \frac{y_q - y_p}{x_q - x_p}, & \text{if } x_q \neq x_p, \\ \frac{3x_p^2}{2y_p}, & \text{if } x_q \neq x_p, \\ 0, & \text{otherwise.} \end{cases}$$

Scalar Multiplication - Custom Constraints

Degree	Constraint	Meaning
4	$(1 - x_a \cdot \beta_q) \cdot x_q$	$x_a = 0 \implies x_q = 0$
4	$(1 - x_a \cdot \beta_q) \cdot y_q$	$x_a = 0 \implies y_q = 0$
3	$(2 \cdot y_a \cdot \lambda_q - 3 \cdot x_a^2)$	$\lambda_q = \frac{3x_a^2}{2y_a}$
3	$(\lambda_q^2 - 2 \cdot x_a - x_q)$	$x_q = \lambda_q^2 - 2 \cdot x_a$
3	$(\lambda_q \cdot (x_a - x_q) - y_a - y_q)$	$y_q = \lambda_q \cdot (x_a - x_q) - y_a$
4	$(x_q - x_p) \cdot ((x_q - x_p) \cdot \lambda_r - (y_q - y_p))$	$x_q \neq x_p \implies \lambda_r = \frac{y_q - y_p}{x_q - x_p}$
5	$(1 - (x_q - x_p) \cdot \alpha_r) \cdot (2y_p \cdot \lambda_r - 3x_p^2)$	$x_q = x_p \wedge y_p \neq 0 \implies \lambda_r = \frac{3x_p^2}{2y_p}$ $x_q = x_p \wedge y_p = 0 \implies x_p = 0$
6	$x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda_r^2 - x_p - x_q - x_r)$	$x_p, x_q \neq 0 \wedge x_q \neq x_p \implies x_r = \lambda_r^2 - x_p - x_q$
6	$x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda_r \cdot (x_p - x_r) - y_p - y_r)$	$x_p, x_q \neq 0 \wedge x_q \neq x_p \implies y_r = \lambda_r \cdot (x_p - x_q) - y_p$
6	$x_p \cdot x_q \cdot (y_q - y_p) \cdot (\lambda_r^2 - x_p - x_q - x_r)$	$x_p, x_q \neq 0 \wedge y_q \neq y_p \implies x_r = \lambda_r^2 - x_p - x_q$
6	$x_p \cdot x_q \cdot (y_q - y_p) \cdot (\lambda_r \cdot (x_p - x_r) - y_p - y_r)$	$x_p, x_q \neq 0 \wedge y_q \neq y_p \implies y_r = \lambda_r \cdot (x_p - x_q) - y_p$
4	$(1 - x_p \cdot \beta_r) \cdot (x_r - x_q))$	$x_p = 0 \implies x_r = x_q$
4	$(1 - x_p \cdot \beta_r) \cdot (y_r - y_q))$	$x_p = 0 \implies y_r = y_q$
4	$(1 - x_q \cdot \gamma_q) \cdot (x_r - x_p))$	$x_q = 0 \implies x_r = x_p$
4	$(1 - x_q \cdot \gamma_q) \cdot (y_r - y_p))$	$x_q = 0 \implies y_r = y_p$
4	$(1 - (x_q - x_p) \cdot \alpha_r - (y_q + y_p) \cdot \delta_r) \cdot x_r)$	$x_q = x_p \wedge y_q = -y_p \implies x_r = 0$
4	$(1 - (x_q - x_p) \cdot \alpha_r - (y_q + y_p) \cdot \delta_r) \cdot y_r)$	$x_q = x_p \wedge y_q = -y_p \implies y_r = 0$
3	$b_i \cdot (b_i - 1)$	$b_i \in \mathbb{B}$
3	$(x_s - (b_i \cdot x_r + (1 - b_i) \cdot x_q))$	$x_s =$ **if** $b_i = 1$ **then** x_r **else** x_q
3	$(y_s - (b_i \cdot y_r + (1 - b_i) \cdot y_q))$	$y_s =$ **if** $b_i = 1$ **then** y_r **else** y_q
3	$(acc_{i+1} - (acc_i + b_i \cdot 2^i))$	$acc_{i+1} - (acc_i + b_i \cdot 2^i)$

Scalar Multiplication - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_a	y_a	acc	x_p	y_p	x_q	y_q	x_r	y_r	b_i	γ_q	λ_q	α_r	β_r	δ_r	λ_r
$\perp$	$\perp$	$\perp$	I_2	I_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Each next S is stored in the next row, so in the constraints, one would define $x_s(X) = w_1(\omega X), y_s = w_2(\omega X), acc_{i+1} = w_3(\omega X)$.

Scalar Multiplication - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	0	0	1	0	0

Scalar Multiplication - Coefficient Row

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}
2^i	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Meaning that in the constraints $2^i = r_1(X)$

Last Row

We need one last row since the next $A =$ if $b_i \stackrel{?}{=} 1$ then R else Q is stored in the next row in each iteration.

Scalar Multiplication (Zero Row) - Witness Row

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_a	y_a	acc	0	0	0	0	0	0	0	0	0	0	0	0	0
O_1	O_2	I_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Note: Copy constraining acc to input 1 ensures that $x = \sum_{i=0}^{254} b_i \cdot 2^i$.

Since this row is just to store the result and copy constrain acc, all selector polynomials are set to zero.

Scalar Multiplication (Zero Row) - Selector Row

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R
0	0	0	0	0	0	0	0	0	0	0

5.4 Full Plonk Protocol

Combining all the previously discussed subprotocols we get the full Plonk protocol. Soundness and completeness should follow from the subprotocols, so we won't discuss those here, but we will do a short analysis and justification for the worst-case runtimes of both algorithms.

But first, let's more precisely define the inputs to the protocols, the circuit, witness and public inputs:

- PlonkCircuit:
  - $n \in \mathbb{F}$: The number of rows in the trace table, must be a power of two.
  - $d = n - 1$: The degree bound for all committed polynomials.
  - $\ell_1 \in \mathbb{F}$: The number of input passes, from *the other circuit* to *this circuit*.
  - $\ell_2 \in \mathbb{F}$: The number of public inputs in the circuit.
  - $\omega \in \mathbb{F}$: The base element for the set of roots of unity $H = \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$
  - Commitments:
    - $\boldsymbol{C_q} \in \mathbb{E}(\mathbb{F})^{10}$: Commitments to the selector polynomials
    - $\boldsymbol{C_r} \in \mathbb{E}(\mathbb{F})^{15}$: Commitments to the coefficient polynomials.
    - $\boldsymbol{C_{id}} \in \mathbb{E}(\mathbb{F})^4$: Commitments to the identity polynomials.
    - $\boldsymbol{C_\sigma} \in \mathbb{E}(\mathbb{F})^4$: Commitments to the σ polynomials.

  All of these are static for the circuit, meaning that they do not depend on the prover's private input or public input.
- PlonkWitness:
  - $\boldsymbol{q^{(e)}} \in (\mathbb{F}^n)^{10}$
  - $\boldsymbol{r^{(e)}} \in (\mathbb{F}^n)^{15}$
  - $\mathbf{id}^{(e)} \in (\mathbb{F}^n)^4$
  - $\boldsymbol{\sigma^{(e)}} \in (\mathbb{F}^n)^4$
  - $\boldsymbol{w^{(e)}} \in (\mathbb{F}^n)^{16}$

  Here, we use e to denote that these are evaluations of the polynomials, which are computed by the arithmetizer. To get the actual polynomials, the prover runs ifft on each of these in round 0.
- PlonkPublicInputs:
  - $\boldsymbol{x} \in \mathbb{F}^{\ell_1+2+\ell_2}$: The public inputs for the trace table, containing both the vanilla public inputs (ℓ_2), the inputs passed from the other circuit (ℓ_1) a commitment to the passed inputs which has size 2.
- PlonkProof:
  - Evaluation Proofs:
    - $\pi_s \in$ **EvalProof**
    - $\pi_{s_\omega} \in$ **EvalProof**
  - Evaluations:
    - $\boldsymbol{q^{(\xi)}} \in \mathbb{F}^{10}$
    - $\boldsymbol{r^{(\xi)}} \in \mathbb{F}^{15}$
    - $\mathbf{id}^{(\xi)} \in \mathbb{F}^4$
    - $\boldsymbol{\sigma^{(\xi)}} \in \mathbb{F}^4$
    - $\boldsymbol{w^{(\xi)}} \in \mathbb{F}^{16}$

    * $\boldsymbol{w^{(\xi\omega)}} \in \mathbb{F}^3$
    * $\boldsymbol{t^{(\xi)}} \in \mathbb{F}^{16}$
    * $z^{(\xi)} \in \mathbb{F}$
    * $z^{(\xi\omega)} \in \mathbb{F}$
- Commitments:
    * $\boldsymbol{C_w} \in \mathbb{E}(\mathbb{F})^{16}$
    * $\boldsymbol{C_t} \in \mathbb{E}(\mathbb{F})^{16}$
    * $C_z \in \mathbb{E}(\mathbb{F})$

All of this PlonkProof is constant size, except the two evaluation proofs which have size $\mathcal{O}(\lg(n))$. Thus the Plonk proof size is also $\mathcal{O}(\lg(n))$.

5.4.1 Prover

Plonk Non-Interactive Prover:

Inputs: PlonkCircuit, PlonkPublicInput, PlonkWitness

Output: PlonkProof

Round 0:

$$\boldsymbol{q} := [\text{ifft}(q_i^{(e)})]_{i=1}^{10}, \quad \boldsymbol{r} := [\text{ifft}(r_i^{(e)})]_{i=1}^{15}, \quad \mathbf{id} := [\text{ifft}(\text{id}_i^{(e)})]_{i=1}^{4}, \quad \boldsymbol{\sigma} := [\text{ifft}(\sigma_i^{(e)})]_{i=1}^{4},$$
$$\boldsymbol{w} := [\text{ifft}(w_i^{(e)})]_{i=1}^{16}, \quad x(X) := \text{ifft}(-\boldsymbol{x})$$

Round 1:

1: $\boldsymbol{C_w} := [\text{PC}_{\text{DL}}.\textsc{Commit}(w_1, d, \perp)]_{i=1}^{16}$
2: $\mathcal{T} \leftarrow \boldsymbol{C_w}$

Round 2:

3: $\mathcal{T} \rightarrow \beta, \gamma$
4: $f'(X) := \prod_{i=1}^{4} w_i(X) + \beta \text{id}_i(X) + \gamma$
5: $g'(X) := \prod_{i=1}^{4} w_i(X) + \beta \sigma_i(X) + \gamma$
6: Define z:
$z(\omega^1) := 1, \quad z(\omega^i) := \prod_{1 \le j < i} \frac{f'(\omega^j)}{g'(\omega^j)}$
7: $C_z := \text{PC}_{\text{DL}}.\textsc{Commit}(z, d, \perp)$
8: $\mathcal{T} \leftarrow C_z$

Round 3:

9: $\mathcal{T} \rightarrow \alpha, \zeta$
10: Define $f_{CG}(X)$ to be all the constraints listed in the custom constraint section, using $[1, \zeta, \zeta^2, \dots]$ as the challenges required for the custom constraints. Define $f(X), t(X)$:
$f_{GC}(X) := w_1(X)q_1(X) + w_2(X)w_2(X) + w_3(X)q_3(X) + w_1(X)w_2(X)q_4(X) + q_5(X) + x(X) + f_{CG}(X)$
$f_{CC_1}(X) := l_1(X) \cdot z(X) - 1$
$f_{CC_2}(X) := z(X) \cdot f'(X) - z(X \cdot \omega) \cdot g'(X)$
$f(X) := f_{GC}(X) + \alpha f_{CC_1}(X) + \alpha^2 f_{CC_2}(X)$
$t(X) := f(X)/z_H(X)$
11: Split $t(X)$ into $\boldsymbol{t} \in \mathbb{F}_{\le d}^{16}$, s.t:
$t(X) = \sum_{i=1}^{16} (X^n)^{i-1} \cdot t_i(X)$
12: Commit to each of them $\boldsymbol{C_t} := [\text{PC}_{\text{DL}}.\textsc{Commit}(t_1(X), d, \perp)]_{i=1}^{16}$
13: $\mathcal{T} \leftarrow \boldsymbol{C_t}$

Round 4:

14: $\mathcal{T} \leftarrow \eta$
15: $s(X) = \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i(X) \quad \text{where} \quad \boldsymbol{\tau} = \boldsymbol{q} + \boldsymbol{w} + \boldsymbol{t} + [z(X)]$
16: $s_\omega(X) = \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i(X) \quad \text{where} \quad \boldsymbol{\tau} = [w_1(X), w_2(X), w_3(X), z(X)]$

Round 5:

17: $\mathcal{T} \rightarrow \xi$
18: $C_s := \text{PC}_{\text{DL}}.\textsc{Commit}(s(X), d, \perp), \quad C_{s_\omega} = \text{PC}_{\text{DL}}.\textsc{Commit}(s_\omega(X), d, \perp)$
19: $\pi_s = \text{PC}_{\text{DL}}.\textsc{Open}(s(X), C_s, d, \xi, \perp)$
20: $\pi_{s_\omega} = \text{PC}_{\text{DL}}.\textsc{Open}(s_\omega(X), C_{s_\omega}, d, \xi \cdot \omega, \perp)$

$$\boldsymbol{q^{(\xi)}} = [q_i(\xi)]_{i=1}^{10}, \quad \boldsymbol{r^{(\xi)}} = [r_i(\xi)]_{i=1}^{15}, \quad \mathbf{id^{(\xi)}} = [\text{id}_i(\xi)]_{i=1}^{4}, \quad \boldsymbol{\sigma^{(\xi)}} = [\sigma_i(\xi)]_{i=1}^{4},$$
$$\boldsymbol{w^{(\xi)}} = [w_i(\xi)]_{i=1}^{16}, \quad \boldsymbol{w^{(\xi\omega)}} = [w_i(\xi \cdot \omega)]_{i=1}^{3}, \quad \boldsymbol{t^{(\xi)}} = [t_i(\xi)]_{i=1}^{16},$$

return $\pi_s, \pi_{s_\omega}, \boldsymbol{q^{(\xi)}}, \boldsymbol{r^{(\xi)}}, \mathbf{id^{(\xi)}}, \boldsymbol{\sigma^{(\xi)}}, \boldsymbol{w^{(\xi)}}, \boldsymbol{w^{(\xi\omega)}}, \boldsymbol{t^{(\xi)}}, z(\xi), z(\xi \cdot \omega), \boldsymbol{C_w}, \boldsymbol{C_t}, C_z$

Runtime:

First note that *all* polynomial multiplications can be modelled to run in $\mathcal{O}(n \lg(n))$ because they can be modelled as $c(X) = a(X) \cdot b(X) = \text{ifft}(\text{fft}(a(X)) \cdot \text{fft}(b(X)))$. The runtime of multiplication over the evaluation domain is $\mathcal{O}(n)$ and the runtime of the fft and ifft is $\mathcal{O}(n \lg(n))$. Polynomial addition is $\mathcal{O}(n)$. The PC_{DL} functions $\text{PC}_{\text{DL}}.\textsc{Commit}, \text{PC}_{\text{DL}}.\textsc{Open}$ have a runtime of $\mathcal{O}(n)$. Therefore the worst-case runtime of the prover is $\mathcal{O}(n \lg(n))$.

5.4.2 Verifier

Plonk Non-Interactive Verifier:

Inputs: PlonkCircuit, PlonkPublicInput, PlonkProof

Output: **Result**$(\top, \bot)$

Round 1:

1: $\mathcal{T} \leftarrow \boldsymbol{C_w}$

Round 2:

2: $\mathcal{T} \rightarrow \beta, \gamma$

3: $\mathcal{T} \leftarrow C_z$

Round 3:

4: $\mathcal{T} \rightarrow \alpha, \zeta$

5: $\mathcal{T} \leftarrow \boldsymbol{C_t}$

Round 4:

6: $\mathcal{T} \rightarrow \eta$

7: $\mathcal{T} \leftarrow C_s, C_{s_\omega}$

Round 5:

8: $\mathcal{T} \rightarrow \xi$

9: Compute:

ξ^n using iterative squaring, since n is a power of 2 ($\lg(n)$ multiplications).

$l_1(\xi) = \frac{\omega \cdot (\xi^n - 1)}{n \cdot (\xi - \omega)}$

$z_H(\xi) = \xi^n - 1$

$x(\xi) = \sum_{i=1}^{\ell_1 + 2 + \ell_2} l_i(\xi) \cdot (-x_i) \quad \text{where} \quad l_i(\xi) = \frac{\omega^i \cdot (\xi^n - 1)}{n \cdot (\xi - \omega^i)}$

10: Define $f_{CG}(\xi)$ to be all the constraints listed in the custom constraint section, using $[1, \zeta, \zeta^2, \dots]$ as the challenges required for the custom constraints, and the evaluations $(\boldsymbol{q^{(v)}}, \boldsymbol{r^{(v)}}, \boldsymbol{w^{(v)}}, \boldsymbol{w_\omega^{(v)}})$ provided by the prover. Then, define $f(\xi), t(\xi)$:

$f'(\xi) = \prod_{i=1}^{4} w_i^{(v)} + \beta \cdot \mathrm{id}_i^{(v)} + \gamma$

$g'(\xi) = \prod_{i=1}^{4} w_i^{(v)} + \beta \cdot \sigma_i^{(v)} + \gamma$

$f_{GC}(\xi) = w_1^{(v)} q_1^{(v)} + w_2^{(v)} q_2^{(v)} + w_3^{(v)} q_3^{(v)} + w_1^{(v)} w_2^{(v)} q_4^{(v)} + q_5^{(v)} + x(\xi) + f_{CG}(\xi)$

$f_{CC_1}(\xi) = l_1(\xi) \cdot (z^{(v)} - 1)$

$f_{CC_2}(\xi) = z^{(v)} \cdot f'(\xi) - z_\omega^{(v)} \cdot g'(\xi)$

$f(\xi) = f_{GC}(\xi) + \alpha \cdot f_{CC_1}(\xi) + \alpha^2 \cdot f_{CC_2}(\xi)$

$t(\xi) = \sum_{i=1}^{16} (\xi^n)^{i-1} \cdot t_i^{(v)}$

11: Compute the evaluations and commitments of $s(X), s_\omega(X)$:

$s(\xi) := \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i \quad \text{where} \quad \boldsymbol{\tau} = \boldsymbol{q^{(v)}} + \boldsymbol{w^{(v)}} + \boldsymbol{t^{(v)}} + [z(\xi)]$

$s_\omega(\xi) := \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i \quad \text{where} \quad \boldsymbol{\tau} = [w_1(\xi \cdot \omega), w_2(\xi \cdot \omega), w_3(\xi \cdot \omega), z(\xi \cdot \omega)]$

$C_s := \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i \quad \text{where} \quad \boldsymbol{\tau} = \boldsymbol{C_q} + \boldsymbol{C_w} + \boldsymbol{C_t} + [C_z]$

$C_{s_\omega} := \sum_{i=1}^{|\boldsymbol{\tau}|} \eta^{i-1} \cdot \tau_i \quad \text{where} \quad \boldsymbol{\tau} = [C_{w_1}, C_{w_2}, C_{w_3}, C_z]$

12: Compute the commitment to the passed inputs:

$C_{\text{IP}} := \sum_{i=1}^{\ell_1} x_i \cdot G_i$

$C'_{\text{IP}} := (x_{\ell_1 + 1}, x_{\ell_1 + 2})$

Checks:

13: $C_{\text{IP}} \stackrel{?}{=} C'_{\text{IP}}$

14: $f(\xi) \stackrel{?}{=} t(\xi) \cdot z_H(\xi)$

15: $\text{PC}_{\text{DL}}.\textsc{Check}(C_s, d, \xi, s(\xi), \pi_s)$

16: $\text{PC}_{\text{DL}}.\textsc{Check}(C_{s_\omega}, d, \xi \cdot \omega, s_\omega(\xi), \pi_{s_\omega})$

Runtime:

- ξ^n is $\mathcal{O}(\lg(n))$ multiplications, due to iterative squaring.
- $x(\xi)$ is $\mathcal{O}(\ell)$
- The computation of $f(\xi), t(\xi)$ doesn't depend on n, so that part is constant. Same is true for $s(X), s_\omega(X)$.
- The computation of C_{IP} is ℓ_1 scalar multiplications.
- All transcript hash interactions are also constant.

Overall, the worst-case runtime with fixed ℓ_1, ℓ_2 and variable n is $\mathcal{O}(\lg(n))$.

6 IVC Scheme

We start by sketching out a novel use-case for IVC, a chain of signatures for use in modern blockchains for fast catchup. Then we formally define the IVC scheme required to achieve this.

6.1 Chain of Signatures

BFT-style blockchains with committees[6], such as Concordium and Partisia, have elected committees sign blocks. The highest block signed by the current committee is deemed the latest block. During catch-up, when a node has to sync with the blockchain, it has to download all previous blocks from the first block, the genesis block. This is also the case for light nodes that require less resources, with slightly inferior security guarantees. We want to enable near-instant catchup for light clients in blockchains based on BFT-style blockchains with committees with only minimal security slackening compared to traditional light client catchup.

We specify a recursive SNARK construction and instantiate it over a chain of signatures, which would allow safe catchup for light clients on the forementioned blockchains. Taking Concordium as the main example; they elect a committee once a day and that committee is responsible for signing valid blocks. Concordium is a proof of stake blockchain so the committee is elected according to the size of their staked tokens. They could create a parallel *IVC blockchain*, one where each block contains:

$$B_i = \{\sigma_i^{(pk)}, j_i = i, pk_i, ptr_i \in \mathbb{B}^{256}, \sigma_i^{(ptr)}\}$$

- $\sigma_i^{(pk)}$: A signature on the public key of the current committee (pk_i), signed by the previous committee identified by the public key pk_{i-1}.
- j_i: A sequential block-id. This must be present for the soundness of the IVC circuit.
- pk_i: The public key of the current committee.
- ptr_i: A hash of the most recent block on the main blockchain.
- $\sigma_i^{(ptr)}$: A signature on ptr_i, signed by the current public key.

Traditionally, a blockchain would need the hash of the previous block to tie together blocks. We can omit that since we already have the signature $\sigma_i^{(pk)}$, linking together pk_{i-1} and pk_i, thus also linking together B_{i-1} and B_i. To verify this IVC blockchain, one would need all blocks from the genesis block B_0, until the most recent block B_n. Then they may verify the relation:

$$\text{Verify}_{pk_{i-1}}(\sigma^{(pk)}, pk_i) \land \text{Verify}_{pk_i}(\sigma^{(ptr)}, ptr_i) \land j_i \stackrel{?}{=} j_{i-1} + 1$$

Now we have a chain of signatures from the first genesis committee, all the way to the final committee at block n. Assuming that the first committee is honest, it should only sign the next honestly elected committee, which by the security of the blockchain should also be majority-honest. That committee will then also only sign the next honest committee. We can continue this argument until reaching committee n, which contains a pointer to the most recent block on the main blockchain. We can now trust that block, and trust the blockchain, given that we trust the genesis committee, and that the subsequent committees have been honest.

This is of course not much of an improvement, to catch up on the main blockchain you need to catch up on some *other* blockchain. The second blockchain is however constructed to be SNARK-friendly. There is only a single public key, representing each committee, the signature scheme can be Schnorr's using poseidon for the hashing of messages, which works well in SNARK constructions. Importantly, this secondary blockchain can use Poseidon hashes, while the main blockchain may prefer Sha3 for security benefits, and the secondary blockchain may use Schnorr signatures, while the main blockchain doesn't have to change their signature scheme.

The main committee still needs to generate and sign using the Schnorr signature scheme, but for this they can use a multisignature scheme like FROST[Komlo and Goldberg 2021]. In the next section we define the IVC scheme that's able to support this.

[6] An example is any blockchain based on the HotStuff[Yin et al. 2019] consensus, which includes Concordium and Partisia.

6.2 IVC Construction

We build the IVC construction using the defined Plonk NARK:

- PLONK.$\textsc{Prover}(R : \textbf{Circuit}, x : \textbf{PublicInputs}, w : \textbf{Witness}) \to \textbf{Proof}$
- PLONK.$\textsc{Verifier}(R : \textbf{Circuit}, x : \textbf{PublicInputs}, \pi : \textbf{Proof}) \to \textbf{Result}(\top, \bot)$
- PLONK.$\textsc{VerifierFast}(R : \textbf{Circuit}, x : \textbf{PublicInputs}) \to \textbf{Result}(\top, \bot)$

The (PLONK.$\textsc{Prover}$, PLONK.$\textsc{Verifier}$) pair are the same as those defined in the previous section. The PLONK.$\textsc{VerifierFast}$, however, is almost the same as PLONK.$\textsc{Verifier}$, but without the PC_{DL}.$\textsc{Check}$ performed on the instances! Instead, the instances can be checked separately by the AS_{DL}.$\textsc{Verifier}$, which lets us define the IVC-circuit, using only sub-linear operations.

Each step in the IVC protocol built from accumulation schemes, consists of the triple $(s_{i-1}, \pi_{i-1}, \text{acc}_{i-1})$, representing the previous proof, accumulator and value. We also operate over two curves now, with two accumulators, two proofs and a single state:

$$(s_i, \text{acc}_i = (\text{acc}_i^{(p)}, \text{acc}_i^{(q)}), \pi_i = (\pi_i^{(p)}, \pi_i^{(q)}))$$

In the base-case π_0 are invalid proofs, and acc_0 are valid accumulation of some dummy instances. This gives us the following chain:

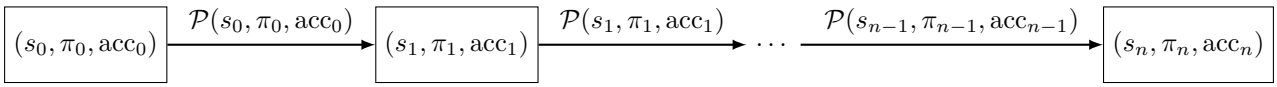


Figure 5: A visualization of the relationship between $F, \boldsymbol{s}, \boldsymbol{\pi}$ and $\textbf{acc}$ in an IVC setting using Accumulation Schemes. Where $\mathcal{P}$ is defined to be $\mathcal{P}(s_{i-1}, \pi_{i-1}, \text{acc}_{i-1}) = \text{IVC.}\textsc{Prover}(s_{i-1}, \pi_{i-1}, \text{acc}_{i-1}) = \pi_i,\ s_i = F(s_{i-1}),$ $\text{acc}_i = \text{AS.}\textsc{Prover}(\boldsymbol{q}, \text{acc}_{i-1})$.

Before describing the IVC protocol, we first describe the circuit for the IVC relation as it's more complex than for the naive SNARK-based approach. Let:

$$\begin{aligned}
c_{\text{VF}_p} &= \text{PLONK.}\textsc{VerifierFast}(R_{IVC}^{(q)}, x_{i-1}^{(p)}, \pi_{i-1}^{(p)}) \stackrel{?}{=} \top \\
c_{\text{AS}_p} &= \text{AS.}\textsc{Verifier}(\boldsymbol{q}^{(p)}, \text{acc}_{i-1}^{(p)}, \text{acc}_i^{(p)}) \stackrel{?}{=} \top \\
c_{\text{VF}_q} &= \text{PLONK.}\textsc{VerifierFast}(R_{IVC}^{(q)}, x_{i-1}^{(q)}, \pi_{i-1}^{(q)}) \stackrel{?}{=} \top \\
c_{\text{AS}_q} &= \text{AS.}\textsc{Verifier}(\boldsymbol{q}^{(q)}, \text{acc}_{i-1}^{(q)}, \text{acc}_i^{(q)}) \stackrel{?}{=} \top \\
c_V &= c_{\text{VF}_p} \wedge c_{\text{AS}_p} \wedge c_{\text{VF}_q} \wedge c_{\text{AS}_q} \\
c_0 &= s_{i-1} \stackrel{?}{=} s_0 \\
c_F &= F'(s_{i-1}, s_i) \\
c_{\text{IVC}} &= (c_0 \vee c_V) \wedge c_F
\end{aligned}$$

We also need to check that the next i is equals to the previous i incremented once, but this can be modelled as part of the state transition check function F'.

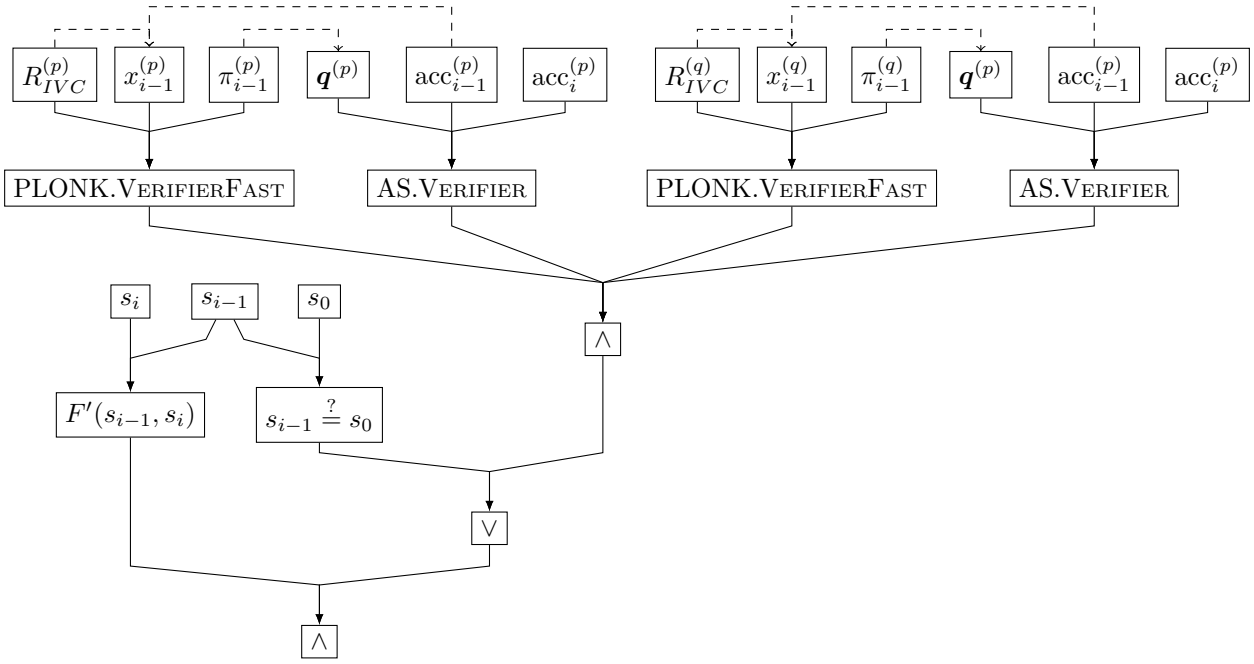


Figure 6: A visualization of R_{IVC}.

For the purpose of creating the chain of signatures we can define:

$$
\begin{aligned}
s_0 &= (\sigma_0, j_0 = 0, pk_0) \\
s_i &= (\sigma_i, j_i, pk_i) \\
F'(s_{i-1}, s_i) &= \textsc{Schnorr.Verify}_{pk_{i-1}}(\sigma_i, pk_i) \wedge j_i \stackrel{?}{=} j_{i-1} + 1
\end{aligned}
$$

The first signature, s_0, can be invalid, since it's never checked. The $j_i \stackrel{?}{=} j_{i-1}$ is required for soundness, it means that each iteration will terminate. The $s_{i-1} \stackrel{?}{=} s_0$ will thus also check whether we are in the base-state with $j = 0$ and that pk_0 is the genesis public-key.

The verifier and prover for the IVC scheme can be seen below:

Algorithm IVC.Prover

Constants

$R_{\text{IVC}} = \left(R^{(p)}_{\text{IVC}}, R^{(q)}_{\text{IVC}}\right)$	The IVC circuit as defined above.
$s_0 = (\sigma_0, 0, pk_0)$	The base IVC-state.

Inputs

$s_{i-1} = (\sigma_{i-1}, j_{i-1}, pk_{i-1})$	The previous IVC-state.
$\pi_{i-1} = \left(\pi^{(p)}_{i-1}, \pi^{(q)}_{i-1}\right)$	The previous IVC-proof.
$\text{acc}_{i-1} = \left(\text{acc}^{(p)}_{i-1}, \text{acc}^{(q)}_{i-1}\right)$	The previous IVC-accumulator.
$s_i = (\sigma_i, j_i, pk_i)$	The next IVC-state

Output

$(S, \textbf{Proof}, \textbf{Acc})$	The values for the next IVC iteration.

Require: $F'(s_{i-1}, s_i) = \top$

Require: $j_i = j_{i-1} + 1$

1: Compute the next IVC-proof, π_i:

2: Define the witness for the IVC-circuit:

$x^{(p)}_{i-1} := \{R^{(p)}_{\text{IVC}}, s_0, s_{i-1}, acc^{(p)}_{i-1}\}$

$x^{(q)}_{i-1} := \{R^{(q)}_{\text{IVC}}, acc^{(q)}_{i-1}\}$

$w^{(p)}_i := \{x^{(p)}_{i-1}, \pi^{(p)}_{i-1}, \text{acc}^{(p)}_{i-1}, s_{i-1}\}$

$w^{(q)}_i := \{x^{(q)}_{i-1}, \pi^{(q)}_{i-1}, \text{acc}^{(q)}_{i-1}\}$

3: Define the public inputs for the IVC-circuit:

$x^{(p)}_i := \{R^{(p)}_{\text{IVC}}, s_0, s_i, acc^{(p)}_i\}$

$x^{(q)}_i := \{R^{(q)}_{\text{IVC}}, acc^{(q)}_i\}$

4: Compute the proofs:

$\pi^{(p)}_i := \text{PLONK.Prover}\left(R^{(p)}_{\text{IVC}}, x^{(p)}_i, w^{(p)}_i\right)$

$\pi^{(q)}_i := \text{PLONK.Prover}\left(R^{(q)}_{\text{IVC}}, x^{(q)}_i, w^{(q)}_i\right)$

$\pi_i := \left(\pi^{(p)}_i, \pi^{(q)}_i\right)$

5: Compute the next accumulator, acc_i:

6: Parse $\boldsymbol{q}^{(p)}$ from $\pi^{(p)}_{i-1}$, and $\boldsymbol{q}^{(q)}$ from $\pi^{(q)}_{i-1}$.

7: Run the AS.Prover.

$\text{acc}^{(p)}_i = \text{AS.Prover}\left(\boldsymbol{q}^{(p)}, \text{acc}^{(p)}_{i-1}\right)$

$\text{acc}^{(q)}_i = \text{AS.Prover}\left(\boldsymbol{q}^{(q)}, \text{acc}^{(q)}_{i-1}\right)$

$\text{acc}_i = \left(\text{acc}^{(p)}_i, \text{acc}^{(q)}_i\right)$

8: Output $(s_i, \pi_i, \text{acc}_i)$

Algorithm IVC.Verifier

Constants

$R_{\text{IVC}} = \left(R^{(p)}_{\text{IVC}}, R^{(q)}_{\text{IVC}}\right)$ The IVC circuit as defined above.

$s_0 = (\sigma_0, 0, pk_0)$ The base IVC-state.

Inputs

$s_i = (\sigma_i, j_i, pk_i)$ The current IVC-state.

$\pi_i = \left(\pi^{(p)}_i, \pi^{(q)}_i\right)$ The current IVC-proof.

$\text{acc}_i = \left(\text{acc}^{(p)}_i, \text{acc}^{(q)}_i\right)$ The current IVC-accumulator.

Output

Result$(\top, \bot)$ Returns $\top$ if the verifier accepts and $\bot$ if the verifier rejects.

1: **if** $s_i \stackrel{?}{=} s_0$ **then** ▷ If this is true, then the proofs will be invalid and unnecessary.

2: **return** $\top$.

3: **end if**

4: Verify the accumulators using the accumulation scheme decider:

$\text{AS.Decider}\left(\text{acc}^{(p)}_i\right) \stackrel{?}{=} \text{AS.Decider}\left(\text{acc}^{(q)}_i\right) \stackrel{?}{=} \top$

5: Verify the Plonk-proofs:

$x^{(p)}_i := \{R^{(p)}_{\text{IVC}}, s_0, s_i, acc^{(p)}_i\}$

$x^{(q)}_i := \{R^{(q)}_{\text{IVC}}, acc^{(q)}_i\}$

$\text{PLONK.Verifier}\left(R^{(p)}_{\text{IVC}}, x^{(p)}_i, \pi^{(p)}_i\right) \stackrel{?}{=} \text{PLONK.Verifier}\left(R^{(q)}_{\text{IVC}}, x^{(q)}_i, \pi^{(q)}_i\right) \stackrel{?}{=} \top$

6: If the above two checks pass, then output $\top$, else output $\bot$.

Consider the IVC-chain from Figure 5 run n times. As in the "simple" SNARK IVC construction, if IVC.Verifier accepts at the end, then we get a chain of implications:

$$\begin{aligned}
&\text{IVC.Verifier}(R_{IVC}, x_n = \{s_0, s_n, \text{acc}_i\}, \pi_n) = \top \implies \\
&\left(c^{(i)}_0 \vee c^{(i)}_V\right) \wedge c^{(i)}_F \implies \\
&c^{(i)}_{\text{VF}_p} \wedge c^{(i)}_{\text{AS}_p} \wedge c^{(i)}_{\text{VF}_q} \wedge c^{(i)}_{\text{AS}_q} \wedge F'(s_{n-1}, s_n) \implies \\
&c^{(i-1)}_{\text{VF}_p} \wedge c^{(i-1)}_{\text{AS}_p} \wedge c^{(i-1)}_{\text{VF}_q} \wedge c^{(i-1)}_{\text{AS}_q} \wedge F'(s_{n-2}, s_{n-1}) \implies \dots \\
&s_{1-1} = s_0 \wedge F'(s_0, s_1)
\end{aligned}$$

Since IVC.Verifier runs AS.Decider, the previous accumulator is valid, and by recursion, all previous accumulators are valid, given that each AS.Verifier accepts. Therefore, if a AS.Verifier accepts, that means that the evaluation proofs are valid. We defined PLONK.VerifierFast, s.t. it verifies correctly provided the $\boldsymbol{q}$'s are valid evaluation proofs. This allows us to recurse through this chain of implications.

From this we learn:

1. $\forall i \in [2, n]$: $\text{AS.Verifier}(\pi^{(p)}_{i-1}, \text{acc}^{(p)}_{i-1}, \text{acc}^{(p)}_i) = \text{AS.Verifier}(\pi^{(q)}_{i-1}, \text{acc}^{(q)}_{i-1}, \text{acc}^{(q)}_i) = \top$, i.e, all accumulators are accumulated correctly.
2. $\forall i \in [2, n]$: $\text{PLONK.VerifierFast}(R^{(p)}_{IVC}, x^{(p)}_{i-1}, \pi^{(p)}_{i-1}) = \text{PLONK.VerifierFast}(R^{(p)}_{IVC}, x^{(p)}_{i-1}, \pi^{(p)}_{i-1}) = \top$, i.e, all the proofs are valid.

These points in turn imply that $\forall i \in [n] : F(s_{i-1}) = s_i$, therefore, $s_n = F^n(s_0)$. From this discussion it should be clear that an honest prover will convince an honest verifier, i.e. completeness holds. As for soundness, it should mostly depend on the soundness of the underlying PCS, accumulation scheme and Plonk[7].

As for efficiency:

- The runtime of IVC.Prover is:

[7] A more thorough soundness discussion would reveal that running the extractor on a proof-chain of length n actually fails, as argued by Valiant in his original 2008 paper. Instead he constructs a proof-tree of size $\mathcal{O}(\lg(n))$ size, to circumvent this. However, practical applications conjecture that the failure of the extractor does not lead to any real-world attack, thus still achieving constant proof sizes, but with an additional security assumption added.

- Step 4: The cost of running AS_{DL}.Prover, $\mathcal{O}(d)$.
- Step 7: The cost of running PLONK.Prover, $\mathcal{O}(d)$.

Totalling $\mathcal{O}(d)$. If the prover is required to compute the next state s_i, it's assumed to be at most linear in d. For a chain of signatures, it's well within bounds.

- The runtime of IVC.Verifier is:
  - Step 4: The cost of running AS_{DL}.Decider, $\mathcal{O}(d)$.
  - Step 5: The cost of running PLONK.Verifier, $\mathcal{O}(d)$.

  Totalling $\mathcal{O}(d)$.

Notice that although the runtime of IVC.Verifier is linear, it scales with d, *not* n. So the cost of verifying does not scale with the number of iterations.

6.3 Arithmetization

In the implementation circuits are written in a custom-made eDSL - an embedded Domain Specific Language - that lets a circuit-creator specify circuits as regular rust code. The circuit is modelled as a DAG - a Directed Acyclic Graph - with the following kinds of nodes:

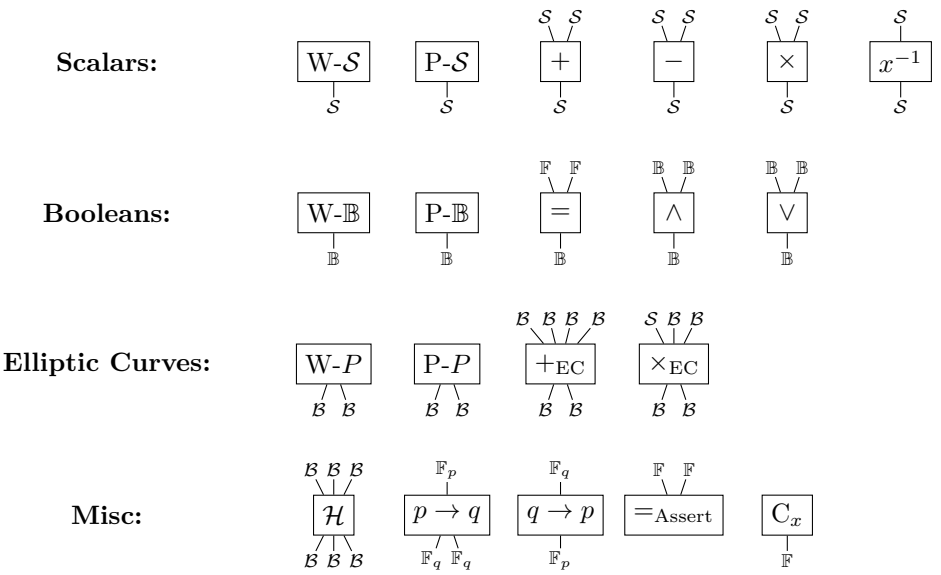


Figure 7: The gates available as DAG nodes.

Wires can have two types: $\mathbb{F}_p$ or $\mathbb{F}_q$. The other symbols denote:

- $\mathbb{B}$: Either $\mathbb{F}_p$ or $\mathbb{F}_q$, but whatever the concrete field element, it is constrained to be either 0, or 1, a bit.
- $\mathcal{S}$: A scalar-field element, either $\mathbb{F}_p$ or $\mathbb{F}_q$, depending on if the whether you model the Pallas ($\mathcal{S} = \mathbb{F}_p$) or Vesta ($\mathcal{S} = \mathbb{F}_q$) curve.
- $\mathcal{B}$: A base-field element, either $\mathbb{F}_p$ or $\mathbb{F}_q$, depending on if the whether you model the Pallas ($\mathcal{B} = \mathbb{F}_q$) or Vesta ($\mathcal{B} = \mathbb{F}_p$) curve.
- $\mathbb{F}$: The gate works when instantiated with either $\mathbb{F}_p$ or $\mathbb{F}_q$.

For both $\mathbb{F}$ and $\mathbb{B}$ it is invalid to provide $\mathbb{B} = \mathbb{F}_p$ as one of the inputs and $\mathbb{B} = \mathbb{F}_q$ as the other.

- **Scalars:**
  - W-$\mathcal{S}$: Witness scalar.
  - P-$\mathcal{S}$: Public input scalar.
  - $(+)$: Add two scalars.
  - $(-)$: Subtract two scalars.
  - $(\times)$: Multiply two scalars.

  - (x^{-1}): Compute the inverse of x.
- **Booleans:**
  - W-$\mathbb{B}$: Witness Boolean, of either $\mathbb{F}_p$ or $\mathbb{F}_q$.
  - P-$\mathbb{B}$: Public input Boolean, of either $\mathbb{F}_p$ or $\mathbb{F}_q$.
  - $(=)$: Equality gate, taking two inputs of the same type, of either $\mathbb{F}_p$ or $\mathbb{F}_q$. If two $\mathbb{F}_p$ elements are inputted, the resulting Boolean element will be an $\mathbb{F}_p$ element constrained to be either 0 or 1, with the converse being true if both inputs are $\mathbb{F}_q$ elements.
  - $(\wedge)$: AND gate, taking two Boolean-constrained inputs of the same type, of either $\mathbb{F}_p$ or $\mathbb{F}_q$. As with the equality gate, the output has the same type as the input.
  - $(\vee)$: OR gate, taking two Boolean-constrained inputs of the same type, of either $\mathbb{F}_p$ or $\mathbb{F}_q$. As with the equality gate, the output has the same type as the input.
- **Elliptic Curves:**
  - W-P: Witness curve point, in affine form, constrained to fit the curve equation.
  - P-P: Public input curve point, in affine form, constrained to fit the curve equation.
  - $(+_{\text{EC}})$: Add two elliptic curve points.
  - $(\times_{\text{EC}})$: Scale a point with a scalar, it is implicit that the scalar is input passed from the scalar-field to the base-field.
- **Miscellaneous:**
  - $\mathcal{H}$: Performs five rounds of the Poseidon hashing on the three given base-field elements, representing the Poseidon sponge state.
  - $p \to q$: Message passes an $\mathbb{F}_p$ element to the $\mathbb{F}_q$ circuit.
  - $q \to p$: Message passes an $\mathbb{F}_q$ element to the $\mathbb{F}_p$ circuit.
  - $(=_{\text{Assert}})$: Asserts that the two field elements of the same type are equal.
  - C_x: A constant gate, outputting a fixed value x of either $\mathbb{F}_p$ or $\mathbb{F}_q$.

We can represent our previous example circuit from Figure 4 using these nodes:

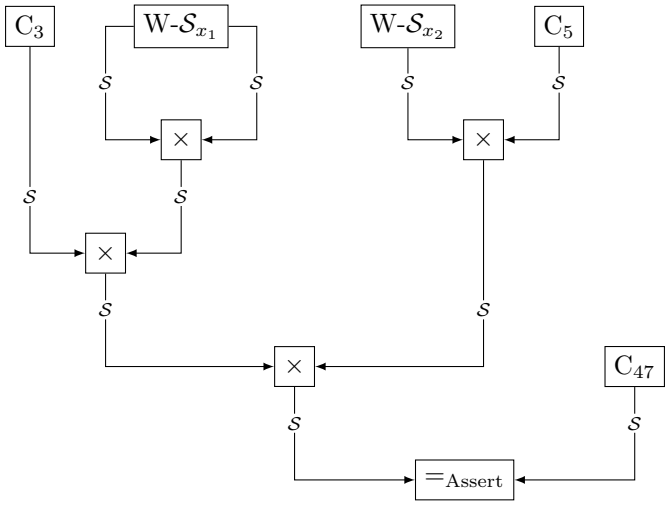


Figure 8: The example circuit from the Figure 4, as a DAG, using the defined DAG nodes.

To arithmetize our program, yielding the polynomials required by the Plonk prover and verifier, we need to extract the necessary constraint table from the circuit and interpolate the columns to get the polynomials. We first define some useful objects:

WireId	$= \mathbb{N}$	(A unique sequential id for each wire),
SlotId	$= \mathbb{N} \times \mathbb{N}$	(An entry in the constraint table, e.g. (4,3) refers to the fourth row, third column),
WireType	$= \{\mathbb{F}_p, \mathbb{F}_q\}$	(Wires have a type, because a wire value can either be $\mathbb{F}_p$ or $\mathbb{F}_q$),

And a **GateType**, describing what kind of gate a node is:

$$
\begin{aligned}
&\textbf{GateType} = \{ \\
&\qquad \text{"W-}\mathcal{S}\text{"}, \text{"P-}\mathcal{S}\text{"}, \text{"}(-)\text{"}, \text{"}(+)\text{"}, \text{"}(\times)\text{"}, \text{"}(x^{-1})\text{"}, \\
&\qquad \text{"W-}\mathbb{B}\text{"}, \text{"P-}\mathcal{S}\text{"}, \text{"}(=)\text{"}, \text{"}(\wedge)\text{"}, \text{"}(\vee)\text{"}, \\
&\qquad \text{"W-}P\text{"}, \text{"P-}P\text{"}, \text{"}(+_{\mathrm{EC}})\text{"}, \text{"}(\times_{\mathrm{EC}})\text{"}, \\
&\qquad \text{"}\mathcal{H}\text{"}, \text{"}p \to q\text{"}, \text{"}q \to p\text{"} \\
&\}
\end{aligned}
$$

The DAG can then be defined, with each vertex containing a **GateType**, n **WireId**'s representing the id's of the incoming wires and m **WireId**'s representing the id's of the outgoing wires. The edges have no associated data:

$$G = (V \in \textbf{GateType} \times \textbf{WireId}^n \times \textbf{WireId}^m, E \in \{\})$$

We describe an algorithm, trace, that generally takes a circuit and processes it into the constraint table. We iterate through the DAG in topological order, processing each node such that all its predecessors, the nodes with edges pointing to it, are processed before it. Throughout the iteration we store and get values from two maps, ev, mapping each wire to a value, and cc, mapping each wire to a set of slot-ids:

$$ev \in \textbf{WireId} \to \{\mathbb{F}_p, \mathbb{F}_q\},\ cc \in \textbf{WireId} \to \{\textbf{SlotId}\}$$

The map ev represents the evaluated values of each wire, and cc represents the slot-ids, the entries of the constraint table, that should be copy constrained to be equal. Then for each node in this iteration we have a node:

$$\forall v \in V : v = (t \in \textbf{GateType},\ \boldsymbol{i} \in \textbf{WireId}^n,\ \boldsymbol{o} \in \textbf{WireId}^m)$$

1. We look up all inputs for the current node v, in the ev map. These lookups will always yield a value since the value has been written in a previous iteration, due to the topological iteration order:

$$\boldsymbol{ev^{(i)}} = [ev(i_1), \ldots, ev(i_n)]$$

2. We compute the evaluation of the operation applied to the input values. The gate type t decides what operation, $op \in \mathbb{F}^n \to \mathbb{F}^m$, shall be performed:

$$\boldsymbol{ev^{(o)}} = op(\boldsymbol{ev^{(i)}}), \quad \forall k \in [n] : ev(o_k) = ev_k^{(i)}$$

   For example, for the addition gate, op would be defined as: $op_{(+)}([ev_1^{(i)}, ev_2^{(i)}]) = ev_1^{(i)} + ev_2^{(i)}$
3. Now we can append a row to the constraint table with the computed values according to the specification in the custom gates section. We also add any relevant coefficient rows.
4. Finally, we add the input and output wires to the copy constraint map:

$$\forall k \in [n] : cc(i_k) = i_k^{(\mathrm{SlotId})},$$
$$\forall k \in [m] : cc(o_k) = o_k^{(\mathrm{SlotId})}$$

It is important that we designate the first ℓ_2 rows for public inputs, but other than that, the above loop describes how to construct the constraint table as defined in the custom gate section. From here we just interpolate each column to get the witness, selector, coefficient and copy constraint polynomials. To run this trace algorithm as a verifier, that doesn't have access to a valid witness, the verifier can run it with a fake witness and omit the private witness table, getting the same selector, coefficient and copy constraint polynomials as the prover.

Example

Consider the following small example circuit:

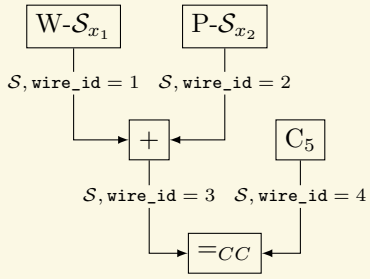


Figure 9: A smaller example circuit representing the claim that I know a private scalar x_1 and a public scalar x_2, s.t. $x_1 + x_2 = 5$.

We iterate through this circuit in topological order, so we can start with either of the two inputs. We instantiate the arithmetization with $x_1 = 2, x_2 = 3$, but we leave them as variables in the following description:

- Node$_1$ = ("W-$\mathcal{S}_{x_1}$", $\boldsymbol{i} = [\,], \boldsymbol{o} = [1]$):
  1. There are no inputs.
  2. There is no computation so: $op_{\text{W-}\mathcal{S}_{x_1}}([\,]) = \boldsymbol{ev^{(o)}} = [x_1], \quad ev(o_1) = ev_1^{(o)} = x_1$.
  3. For private inputs, there is no rows added to the constraint table.
  4. Since there is no row, there is no **SlotId** to add to the copy constraints.
- Node$_2$ = ("P-$\mathcal{S}_{x_2}$", $\boldsymbol{i} = [\,], \boldsymbol{o} = [2]$):
  1. There are no inputs.
  2. There is no computation so: $op_{\text{P-}\mathcal{S}_{x_2}}([\,]) = \boldsymbol{ev^{(o)}} = [x_2], \quad ev(o_1) = ev_1^{(o)} = x_2$.
  3. For public inputs, we add the following witnesses and selector polynomials:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R					
1	0	0	0	0	0	0	0	0	0	0					

  4. We add the slot-id of x_2 $(1, 1)$ to the copy constraints of the output wire with id 2:

$$cc(o_1 = 2) = cc(o_1) \cup \{\, o_1^{\text{SlotId}} = (1, 1) \,\}$$

- Node$_3$ = ("(+)", $\boldsymbol{i} = [1, 2], \boldsymbol{o} = [3]$):
  1. We lookup the two inputs: $\boldsymbol{ev^{(i)}} = [ev(i_1 = 1) = x_1, ev(i_2 = 2) = x_2]$.
  2. Perform the computation:

$$op_{(+)}(\boldsymbol{ev^{(i)}}) = \boldsymbol{ev^{(o)}} = [ev_1^{(i)} + ev_2^{(i)}] = [x_1 + x_2] = [x_3], \quad ev(o_1) = ev_1^{(i)} = x_3$$

  3. For addition, we add the following witnesses and selector polynomials:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_1	x_2	x_3	0	0	0	0	0	0	0	0	0	0	0	0	0

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R					
1	1	-1	0	0	0	0	0	0	0	0					

  4. We add the slot-ids to the copy constraints:

$$cc(i_1 = 1) = cc(i_1) \cup \{\, i_1^{\text{SlotId}} = (2, 1) \,\}$$
$$cc(i_2 = 2) = cc(i_2) \cup \{\, i_2^{\text{SlotId}} = (2, 2) \,\}$$
$$cc(o_1 = 3) = cc(o_1) \cup \{\, o_1^{\text{SlotId}} = (2, 3) \,\}$$

- Node$_4$ = ("C_5", $\boldsymbol{i} = [\,], \boldsymbol{o} = [4]$):

1. There are no inputs.
2. There is no computation so:

$$op_{\mathrm{C}_5}([\,]) = 5, \quad \boldsymbol{ev} = [ev_1^{(i)} = 5], \quad ev(o_1) = ev_1^{(i)} = 5$$

3. For a constant gate, we add the following witnesses and selector polynomials:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R					
1	0	0	0	-5	0	0	0	0	0	0					

4. We add the slot-id to the copy constraints:

$$cc(o_1 = 4) = cc(o_1) \cup \{\, o_1^{\mathrm{SlotId}} = (3, 1) \,\}$$

- $\mathrm{Node}_5 = (\text{"}(=_{CC})\text{"}, \boldsymbol{i} = [3, 4], \boldsymbol{o} = [\,])$:
  1. We lookup the two inputs: $\boldsymbol{ev^{(i)}} = [ev(i_1 = 3) = x_3, ev(i_2 = 4) = 5]$
  2. There is no output, so no computation.
  3. For a constant gate, we add the following witnesses and selector polynomials:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
x_3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R					
1	-1	0	0	0	0	0	0	0	0	0					

4. We add the slot-id to the copy constraints:

$$cc(i_1 = 3) = cc(i_1) \cup \{\, i_1^{\mathrm{SlotId}} = (4, 1) \,\}$$
$$cc(i_2 = 4) = cc(i_2) \cup \{\, i_2^{\mathrm{SlotId}} = (4, 2) \,\}$$

Which finishes the arithmetization loop. If we arithmetized with $x_1 = 2, x_2 = 3$ we would get the following table:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	w_{11}	w_{12}	w_{13}	w_{14}	w_{15}	w_{16}
$x_2 = 3$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_1 = 2$	$x_2 = 3$	$x_3 = 5$	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_4 = 5$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_3 = 5$	$x_4 = 5$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

q_l	q_r	q_o	q_m	q_c	$q_{\mathcal{H}}$	q_P	$q_{(+)}$	$q_{(\cdot)}$	$q_{(=)}$	q_R					
1	0	0	0	0	0	0	0	0	0	0					
1	1	-1	0	0	0	0	0	0	0	0					
1	0	0	0	-5	0	0	0	0	0	0					
1	-1	0	0	0	0	0	0	0	0	0					

The copy constraints were defined as:

$$cc(1) = \{\, (2, 1) \,\}$$
$$cc(2) = \{\, (1, 1), (2, 2) \,\}$$
$$cc(3) = \{\, (2, 3), (4, 1) \,\}$$
$$cc(4) = \{\, (3, 1), (4, 2) \,\}$$

Which gives us the following copy-constraint table:

ω^i	id_1	id_2	id_3	id_4	id_5	id_6	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
ω^1	1	5	9	13	17	21	6	5	9	13	17	21
ω^2	2	6	10	14	18	22	2	1	4	14	18	22
ω^3	3	7	11	15	19	23	8	7	11	15	19	23
ω^4	4	8	12	16	20	24	10	3	12	16	20	24

From here, we just need to interpolate each column into a polynomial.

7 Implementation and Benchmarks

We implemented the Plonk prover and verifier in Rust, using the previous implementations[Jakobsen 2025] of AS_{DL} and PC_{DL} as submodules. Both submodules still needed pretty significant changes however. Neither submodule supported generic curves, which would be needed for Plonk instantiated over a cycle of curves. A new infrastructure for setup parameters had to be implemented, that efficiently supported much higher degree polynomials, since the IVC circuit is still quite large. The Fiat-Shamir hashing also needed to be changed over to a Poseidon sponge-based construction, rather than Sha3, which we implemented ourselves. The Poseidon implementation was inspired by Mina's work, so we used the same parameters (Since we also use the fields from Pallas and Vesta in the hashing, just like Mina's Kimchi) and unit-tested that the hash-behaviour of our implementation was identical to theirs.

After this, the Plonk arithmetizer, prover, and verifier could be implemented and parametrized by a given curve, either Pallas or Vesta. The implemented arithmetizer supports standard elliptic curve operations, Fiat-Shamir oriented sponge-based hashing using Poseidon, regular scalar operations and Boolean operations. This is implemented as a circuit (modelled as a directed acyclic graph) with wrappers around it, effectively creating an embedded domain specific language for writing circuits in Rust. The frontend is so similar that the code to implement the in-circuit verifiers for IVC looks almost identical to the Rust/Arkworks implementations. This made it much easier to implement the relevant verifying circuits. Here's an overview of the Rust crates[8]:

- Plonk: The Plonk Prover/Verifier, including arithmetization and IVC-circuit. This also includes all the subcircuits needed for IVC (Poseidon, PC_{DL}.SuccinctCheck, AS_{DL}.Verifier, PLONK.Verifier).
- Accumulation: Compromising of the PCS, PC_{DL}, and the accumulation scheme, AS_{DL}. This was already implemented.
- Group: Code relating to evaluation domains, public parameters for PC_{DL} (including caching them to binary files), and wrapper traits and struct for the cycle of curves.
- Poseidon: The Poseidon hash function, implemented in Rust, not in-circuit.
- Schnorr: A simple Schnorr signature implementation, using Poseidon for the message hash function.

As the purpose of the code is to prototype the ideas presented, and get some benchmarks on the performance of the scheme, there might be soundness bugs in the implementation. Obviously, the code should not be used in production. However, any soundness bugs should not affect performance to any significant degree.

Before presenting the benchmarks, we first briefly discuss what performance is needed for our IVC approach to be preferred. If Concordium created a light node implementation, there are several available ways to catch-up to the current block and trust that the block is correct:

- Catch up as a full node would, validating each block, which would take days.
- Simply trust a full node, which is very insecure.
- Ask a lot of full nodes and if a quorum agrees that a given block is the current one, then use it. This is more secure, but requires a lot of network traffic from previously unconnected peers.
- Verify only the blocks where the committee changes, which is one block per day.

The last option is definitely the preferred one, and thus our chain of signatures SNARK should compete with that solution. Here we would need to verify 2 signatures per day (the other signature arises from Concordium's Finality layer) and some hashes. For this comparison we just focus on the signatures. We can now present the benchmarks which ran on a 20 thread Thinkpad P50:

[8]The Plonk crate is 17,116 LOC, the accumulation crate is 2,940 LOC, the group crate is 4,240 LOC, the Poseidon crate is 948 LOC and the Schnorr crate is 169 LOC

- **IVC-Prover:** Parallel: ~300 s. Single Threaded: ~900 s.
- **IVC-Verifier:** Parallel: ~3 s. Single Threaded: ~9 s.
- **Naive Signature Verification:** Parallel: ~1300 signatures per second. Single Threaded: ~310 signatures per second.

Assuming that the only bottleneck in this process is processing power would mean that for a multithreaded verifier, it would take $1300 \cdot 3 \ / \ 2 = 1950$ days before the IVC solution was faster. Which is not ideal given the complexity of the IVC construction, but it's not far from being viable. If we instead look at the size of each "proof" involved, starting with the IVC proof:

- Signature: 1 point, 1 scalar.
- EvalProof: $1 + 2\lg(n) = 1 + 16 = 17$ points, 1 scalar.
- PlonkProof: 2 EvalProofs, 33 commitments (points), 74 polynomial evaluations (scalars). 67 points, 76 scalars.
- Accumulator: 1 EvalProof, 1 point, 3 scalars. 18 points, 4 scalars.
- IVC-Proof: 2 Accumulators, 2 PlonkProofs, 2 signatures, 2 public-keys (2 points), 2 j scalars. 174 points, 164 scalars,

Modelling each scalar as 256 bits and each point as 256 bits (255 bit field element and 1 additional sign bit), gives us ~10 kB for a single IVC proof. Comparing to just verifying the signatures, after 87 days the IVC proof will be smaller than the 174 signatures needed to verify the same claim. Obviously, if the committee changes more ofter (say once an hour), the IVC approach will much more quickly become economical.

If the use-case is to create a single proof for a new blockchain committee once a day, ~5 minutes on a modern laptop is not at all unreasonable. As for the verifier, it takes ~3 s, which isn't ideal, but will be better than the naive solution after 1950 days. The proof size is okay comparatively though, as the IVC proof will be smaller than the naive solution after only 87 days. These results are pretty promising, especially considering that further optimization should be possible.

8 Conclusion

The core goal for this thesis was to implement, benchmark, analyze and understand Incrementally Verifiable Computation in its entirety, using the ideas put forward in the Halo paper[Bowe et al. 2019]. We also wished to show whether an IVC chain of signatures could be practically useful in the blockchain industry with currently known recursive SNARK technology. The benchmarks show that IVC may be a decent solution, but they do not definitively show that the IVC solution is markedly better than the naive solution. Given that our results indicate that it's viable using our simplified recursive SNARK, then it should definitely be feasible for the more optimized Kimchi and Halo2 protocols.

There are plenty of remaining optimizations and improvements. This SNARK is not quantum-safe, but if instantiated with FRI and a corresponding accumulation scheme it should be able to be adapted with only minor modifications. Of course, the omitted optimizations, like the Maller optimization, could be added back for smaller proof sizes and a faster verifier. Zero-knowledge might not be particularly useful for IVC, but adding it would be useful if the Plonk construction is also used as a general-purpose ZK-SNARK. Lookups can also be very useful, for certain operations like XOR, which would be especially important if the SNARK should be able to model SHA3 efficiently. We investigated Plonkup[Luke Pearson 2020] for this, but ultimately deemed it unnecessary to achieve primary goal of IVC.

A Appendix

A.1 Security of Elliptic Curve Addition Constraints

Analysis

1. $q_{(+)} \cdot (x_q - x_p) \cdot ((x_q - x_p) \cdot \lambda - (y_q - y_p)) = 0$

   Which ensures:
   $x_q \neq x_p \implies \lambda = \frac{y_q - y_p}{x_q - x_p}$
   $(P \neq Q \land P \neq -Q) \implies \lambda = \frac{y_q - y_p}{x_q - x_p}$

2. $q_{(+)} \cdot (1 - (x_q - x_p) \cdot \alpha) \cdot (2y_p \cdot \lambda - 3x_p^2) = 0$

   Meaning that $x_q = x_p \implies \lambda = \frac{3x_p^2}{2y_p}$, except if $y_p = 0$, then:

   $0 = (1 - (x_q - x_p) \cdot \alpha) \cdot (2y_p \cdot \lambda - 3x_p^2) = (2y_p \cdot \lambda - 3x_p^2) = -3x_p^2$

   Which is only satisfied if $x_p = 0$. So this means that the constraint ensures:

   $x_q = x_p \land y_p \neq 0 \implies \lambda = \frac{3x_p^2}{2y_p}$
   $x_q = x_p \land y_p = 0 \implies x_p = 0$
   $(P = Q \lor Q = -P) \land P \neq \mathcal{O} \land Q \neq \mathcal{O} \implies \lambda = \frac{3x_p^2}{2y_p}$

3. a. $q_{(+)} \cdot (x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda^2 - x_p - x_q - x_r) = 0$
   b. $q_{(+)} \cdot (x_p \cdot x_q \cdot (x_q - x_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r) = 0$
   c. $q_{(+)} \cdot (x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda^2 - x_p - x_q - x_r) = 0$
   d. $q_{(+)} \cdot (x_p \cdot x_q \cdot (y_q + y_p) \cdot (\lambda \cdot (x_p - x_r) - y_p - y_r) = 0$

   It's clear that if $(x_p \cdot x_q \cdot (x_q - x_p) \neq 0 \implies x_p \neq 0 \land x_q \neq 0 \land x_q \neq x_p$. So 3.a states:

   $x_p \neq 0 \land x_q \neq 0 \land x_q \neq x_p \implies x_r = \lambda^2 - x_p - x_q$.

   Constraint 3.b, 3.c, 3.d have similar meaning. Combining 3.a, 3.b, 3.c, 3.d yields:

   $x_p \neq 0 \land x_q \neq 0 \land x_q \neq x_p \implies x_r = \lambda^2 - x_p - x_q \land y_r = \lambda \cdot (x_p - x_r) - y_p$
   $x_p \neq 0 \land x_q \neq 0 \land y_q \neq -y_p \implies x_r = \lambda^2 - x_p - x_q \land y_r = \lambda \cdot (x_p - x_r) - y_p$

   Or equivalently:

   $x_p \neq 0 \land x_q \neq 0 \land x_q \neq x_p \land y_q \neq -y_p \implies x_r = \lambda^2 - x_p - x_q \land y_r = \lambda \cdot (x_p - x_r) - y_p$

   From the curve we know that any point where x or y is 0, is invalid, except if it's the identity point. We can also combine the two implications:

   $x_p \neq 0 \land y_p \neq 0 \land x_q \neq 0 \land y_q \neq 0 \land x_q \neq x_p \land y_q \neq -y_p \implies x_r = \lambda^2 - x_p - x_q \land y_r = \lambda \cdot (x_p - x_r) - y_p$

   Which simplifies to:

   $P \neq \mathcal{O} \land Q \neq \mathcal{O} \land -P \neq Q \implies x_r = \lambda^2 - x_p - x_q \land y_r = \lambda \cdot (x_p - x_r) - y_p$

4. a. $q_{(+)} \cdot (1 - x_p \cdot \beta) \cdot (x_r - x_q) = 0$
   b. $q_{(+)} \cdot (1 - x_p \cdot \beta) \cdot (y_r - y_q) = 0$

   Meaning that:
   $x_p = 0 \land y_p = 0 \implies x_r = x_q \land y_r = y_q$
   $P = \mathcal{O} \implies R = Q$

5. a. $q_{(+)} \cdot (1 - x_q \cdot \beta) \cdot (x_r - x_p) = 0$
   b. $q_{(+)} \cdot (1 - x_q \cdot \beta) \cdot (y_r - y_p) = 0$

   Meaning that:
   $x_q = 0 \land y_q = 0 \implies x_r = x_p \land y_r = y_p$
   $Q = \mathcal{O} \implies R = P$

6. a. $q_{(+)} \cdot (1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot x_r = 0$
   b. $q_{(+)} \cdot (1 - (x_q - x_p) \cdot \alpha - (y_q + y_p) \cdot \delta) \cdot y_r = 0$

   Meaning that:
   $x_q = x_p \land y_q = -y_p \implies x_r = 0 \land y_r = 0$
   $Q = -P \implies R = \mathcal{O}$

Security:

Analyzing the cases of $P + Q = R$:

- $\mathcal{O} + \mathcal{O} = \mathcal{O}$
  - Completeness:
    1. Holds because $P = Q$

2. Holds because $P = Q = \mathcal{O}$
3. Holds because $P = Q = \mathcal{O}$
4. Holds because $P = \mathcal{O} \wedge R = Q = \mathcal{O}$
5. Holds because $Q = \mathcal{O} \wedge R = P = \mathcal{O}$
6. Holds because $Q = -P = \mathcal{O} \wedge R = \mathcal{O}$

- Soundness: $R = \mathcal{O}$ is the only solution to 6.

- $P + \mathcal{O} = P : P \neq \mathcal{O}$
  - Completeness:
    1. $P \neq Q$, so $\lambda = \frac{y_q - y_p}{x_q - x_p}$ is a solution
    2. Holds because $Q = \mathcal{O}$
    3. Holds because $Q = \mathcal{O}$
    4. Holds because $P \neq \mathcal{O}$
    5. Holds because $Q = \mathcal{O} \wedge R = P$
    6. Holds because $Q \neq -P$
  - Soundness: $R = \mathcal{O}$ is the only solution to 5.
- $\mathcal{O} + Q = Q : Q \neq \mathcal{O}$
  - Completeness:
    1. $P \neq Q$, so $\lambda = \frac{y_q - y_p}{x_q - x_p}$ is a solution
    2. Holds because $P = \mathcal{O}$
    3. Holds because $P = \mathcal{O}$
    4. Holds because $P = \mathcal{O} \wedge R = Q$
    5. Holds because $Q \neq \mathcal{O}$
    6. Holds because $Q \neq -P$
  - Soundness: $R = \mathcal{O}$ is the only solution to 4.
- $P + Q = 2P : P = Q \neq \mathcal{O}$
  - Completeness:
    1. Holds because $P = Q$
    2. $P = Q$, so $\lambda = \frac{2x_p^2}{2y_p}$ is a solution
    3. Holds because $P \neq \mathcal{O} \wedge Q \neq 0 \wedge Q \neq -P$, so $R = (x_r = \lambda^2 - x_p - x_q, y_r = \lambda \cdot (x_p - x_r) - y_p)$. Which is consistent with point doubling, since $x_p = x_q$ and $\lambda = \frac{2x_p^2}{2y_p}$.
    4. Holds because $P \neq \mathcal{O}$
    5. Holds because $Q \neq \mathcal{O}$
    6. Holds because $Q \neq -P$
  - Soundness: λ is computed correctly (2). $R = 2P$ is the only solution to 3.
- $P + Q = \mathcal{O} : P = -Q \neq \mathcal{O}$
  - Completeness:
    1. Holds because $P = -Q$
    2. $P = Q$, so $\lambda = \frac{2x_p^2}{2y_p}$ is a solution
    3. Holds because $-P = Q$
    4. Holds because $P \neq \mathcal{O}$
    5. Holds because $Q \neq \mathcal{O}$
    6. Holds because $Q = -P \wedge R = \mathcal{O}$
  - Soundness: $R = \mathcal{O}$ is the only solution to 6.
- $P + Q = \mathcal{O} : P \neq -Q, P \neq Q, P \neq \mathcal{O}, Q \neq \mathcal{O}$
  - Completeness:
    1. $P \neq Q \wedge P \neq -Q$, so $\lambda = \frac{y_q - y_p}{x_q - x_p}$ is a valid solution
    2. Holds because $P \neq Q \wedge Q \neq -P$
    3. Holds because $P \neq -Q$, so $R = (x_r = \lambda^2 - x_p - x_q, y_r = \lambda \cdot (x_p - x_r) - y_p)$. Which is consistent with affine point addition since $\lambda = \frac{y_q - y_p}{x_q - x_p}$.
    4. Holds because $P \neq \mathcal{O}$
    5. Holds because $Q \neq \mathcal{O}$
    6. Holds because $Q \neq -P$
  - Soundness: λ is computed correctly (2). $R = P + Q$ is the only solution

A.2 Gadgets

We present the gadgets needed to create the IVC circuit for completeness. We omit the PLONK.Verifier as it's exactly identical to the one defined in the Plonk section, but with verification failure and success modelled with Booleans.

A.2.1 Poseidon Sponges

The Poseidon State can be one of the following values:

$$\textbf{SpongeState} = \begin{cases} \texttt{Absorbed}(0) \\ \texttt{Absorbed}(1) \\ \texttt{Absorbed}(2) \\ \texttt{Squeezed}(1) \\ \texttt{Squeezed}(2) \end{cases}$$

The **SpongeState** shouldn't be part of the circuit, it just governs what when the full poseidon gates should be added to the circuit, i.e. when enough values has been absorbed and can thus be modelled outside the circuit.

Inner Sponge Absorb Gadget: Absorbs a list of field elements into the poseidon sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}^3$	The inner state of the sponge (3 field elements).
$\boldsymbol{xs}$	The field elements that the sponge should absorb.

Output

$\boldsymbol{s} : \mathbb{F}^3$	The sponge inner state after absorption.

for x in $\boldsymbol{xs}$ **do**
  if sponge_state = $\texttt{Absorbed}(n) \land n < 2$ **then**
    sponge_state = $\texttt{Absorbed}(n+1)$
    $s_n = x$
  else if sponge_state = $\texttt{Absorbed}(2)$ **then**
    for $i \in [0, 10]$ **do** ▷ Permute 55 times by using the Hades Gate 11 times
      $\boldsymbol{s} = \mathit{PoseidonBlockCipher}(i, c, \boldsymbol{s})$
    end for
    sponge_state = $\texttt{Absorbed}(1)$
    $s_0 = s_0 + x$
  else
    sponge_state = $\texttt{Absorbed}(1)$
    $s_0 = s_0 + x$
  end if
end for
return $\boldsymbol{s}$

Inner Sponge Squeeze Gadget: Squeezes a field element from the the poseidon sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}^3$	The inner state of the sponge (3 field elements).

Output

$(s, x) : (\mathbb{F}^3, \mathbb{F})$	The sponge inner state after absorption and the squeezed element.

if sponge_state $=$ `Squeezed`$(n) \wedge n < 2$ **then**
  sponge_state $=$ `Squeezed`$(n+1)$
  Return $x = s_n$
else
  for $i \in [0, 10]$ **do** ▷ Permute 55 times by using the Hades Gate 11 times
    $\boldsymbol{s} = \text{HadesGate}_i(c, \boldsymbol{s})$
  end for
  sponge_state $=$ `Squeezed`(1)
  return $(\boldsymbol{s}, x = s_0)$
end if

Outer Sponge Absorb Affine Gadget: Absorbs affine points into the inner sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The inner state of the sponge (3 field elements).
$\boldsymbol{Ps} : \mathbb{E}(\mathbb{F}_{\mathcal{B}})^k$	The affine points to absorb

Output

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The sponge inner state after absorption.

for P in $\boldsymbol{Ps}$ **do**
  if $P \stackrel{?}{=} \mathcal{O}$ **then**
    InnerAbsorb$(\boldsymbol{s}, 0)$
    InnerAbsorb$(\boldsymbol{s}, 0)$
  else
    InnerAbsorb$(\boldsymbol{s}, P.x)$
    InnerAbsorb$(\boldsymbol{s}, P.y)$
  end if
end for

Outer Sponge Absorb Field Element Gadget: Absorbs field elements into the inner sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The inner state of the sponge (3 field elements).
$\boldsymbol{xs} : \mathbb{F}_{\mathcal{S}}^k$	The field elements to absorb

Output

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The sponge inner state after absorption.

for x in $\boldsymbol{xs}$ **do**
  InnerAbsorb$(\boldsymbol{s}, x)$
end for
return s

Outer Sponge Absorb Scalar Gadget: Absorbs scalars into the inner sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The inner state of the sponge (3 field elements).
$\boldsymbol{xs} \in \mathbb{F}_{\mathcal{S}}^k$	The scalars to absorb

Output

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The sponge inner state after absorption.

for x in $\boldsymbol{xs}$ **do**
    Input pass x.
    if |Scalar-Field| < |Base-Field| **then**
        InnerAbsorb($\boldsymbol{s}, x$)
    else
        Decompose x into h, l where h represents the high-bits of x and l represents the low-bit.
        InnerAbsorb($\boldsymbol{s}, h$)
        InnerAbsorb($\boldsymbol{s}, l$)
    end if
end for
return $\boldsymbol{s}$

Outer Sponge Squeeze Scalar Gadget: Squeezes a scalar from the inner sponge.

Inputs

$\boldsymbol{s} : \mathbb{F}_{\mathcal{B}}^3$	The inner state of the sponge (3 field elements).

Output

$(s, x) : (\mathbb{F}_{\mathcal{B}}^3, \mathbb{F}_{\mathcal{S}})$	The sponge inner state after squeezing and the squeezed scalar.

$x =$ InnerSqueeze($\boldsymbol{s}$)
if $x <$ |Base-Field| **then**
    return $\boldsymbol{s}$ and x input passed.
else
    return $\boldsymbol{s}$ and h input passed, where h is the high 254 bits of x.
end if

We lose a single bit of security if $x \geq$ |Base-Field|, but this only increases the odds of an attack by a small constant amount, which is still negligible.

A.2.2 $\mathbf{PC_{DL}}$

The below algorithm is the non-ZK version of the algorithm specified in the previous accumulation scheme project[Jakobsen 2025].

Algorithm 9 $\mathrm{PC_{DL}}.\textsc{SuccinctCheck}^{\rho_0}$

Inputs

$C : \mathbb{E}(\mathbb{F}_{\mathcal{B}})$	A commitment to the coefficients of p.
$d : \mathbb{N}$	A degree bound on p.
$z : \mathbb{F}_{\mathcal{S}}$	The element that p is evaluated on.
$v : \mathbb{F}_{\mathcal{S}}$	The claimed element $v = p(z)$.
π : **EvalProof**	The evaluation proof produced by $\mathrm{PC_{DL}}.\textsc{Open}$.

Output

$(\mathbb{B}, (\mathbb{F}_{\mathcal{S},d}[X], \mathbb{E}(\mathbb{F}_{\mathcal{B}})))$	The algorithm will either succeed and output $(\top, (h : \mathbb{F}_{\mathcal{S},d}[X], U : \mathbb{E}(\mathbb{F}_{\mathcal{B}}))$ if π is a valid proof and otherwise fail $(\bot, (h : \mathbb{F}_{\mathcal{S},d}[X], U : \mathbb{E}(\mathbb{F}_{\mathcal{B}}))$.

Require: $d \leq D$

Require: $(d+1)$ is a power of 2.

Parse π as $(\boldsymbol{L}, \boldsymbol{R}, U := G^{(0)}, c := c^{(0)})$ and let $n = d + 1$.
Compute the 0-th challenge: $\xi_0 := \rho_0(C, z, v)$, and set $H' := \xi_0 H \in \mathbb{E}(\mathbb{F}_{\mathcal{B}})$.
Compute the group element $C_0 := C + vH' \in \mathbb{E}(\mathbb{F}_{\mathcal{B}})$.
for $i \in [\lg(n)]$ **do**
    Generate the i-th challenge: $\xi_i := \rho_0(\xi_{i-1}, L_i, R_i) \in \mathbb{F}_{\mathcal{S}}$.
    Compute the i-th commitment: $C_i := \xi_i^{-1} L_i + C_{i-1} + \xi_i R_i \in \mathbb{E}(\mathbb{F}_{\mathcal{B}})$.
end for
Define the univariate polynomial $h(X) := \prod_{i=0}^{\lg(n)-1} (1 + \xi_{\lg(n)-i} X^{2^i}) \in \mathbb{F}_{\mathcal{S},d}[X]$.
Compute the evaluation $v' := c \cdot h(z) \in \mathbb{F}_{\mathcal{S}}$.
$b = C_{lg(n)} \stackrel{?}{=} cU + v'H'$
Output $(b, (h(X), U))$.

A.2.3 $\mathbf{AS_{DL}}$

The below algorithms are the non-ZK versions of the algorithms specified in the previous accumulation scheme project[Jakobsen 2025].

Algorithm 10 $\mathrm{AS_{DL}}.\textsc{CommonSubroutine}$

Inputs

$\boldsymbol{q}$: **Instance**^{m}	New instances *and accumulators* to be accumulated.

Output

$(\mathbb{B}, (\mathbb{E}(\mathbb{F}_{\mathcal{B}}), \mathbb{N}, \mathbb{F}_{\mathcal{S}}, \mathbb{F}_{\mathcal{S},d}[X]))$	The algorithm will either succeed $(\top, (\mathbb{E}(\mathbb{F}_{\mathcal{B}}), \mathbb{N}, \mathbb{F}, \mathbb{F}_{\mathcal{S},d}[X]))$ if the instances has consistent degree and hiding parameters and will otherwise fail $(\bot, (\mathbb{E}(\mathbb{F}_{\mathcal{B}}), \mathbb{N}, \mathbb{F}, \mathbb{F}_{\mathcal{S},d}[X]))$.

Require: $(D+1) = 2^k$, where $k \in \mathbb{N}$

Parse d from q_1.
Let $b = \top$
for $j \in [0, m]$ **do**
    Parse q_j as a tuple $(C_j, d_j, z_j, v_j, \pi_j)$.
    Compute $(b_i, (h_j(X), U_j)) := \mathrm{PC_{DL}}.\textsc{SuccinctCheck}^{\rho_0}(C_j, d_j, z_j, v_j, \pi_j)$.
    $b = b \wedge b_i$
    Check that $b_i = d_j \stackrel{?}{=} d$
    $b = b \wedge b_i$
end for
Compute the challenge $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U}) \in \mathbb{F}_{\mathcal{S}}$
Let the polynomial $h(X) := \sum_{j=1}^{m} \alpha^j h_j(X) \in \mathbb{F}_{\mathcal{S},d}[X]$
Compute the accumulated commitment $C := \sum_{j=1}^{m} \alpha^j U_j$
Compute the challenge $z := \rho_1(C, h(X)) \in \mathbb{F}_{\mathcal{S}}$.
Randomize C: $\bar{C} := C \in \mathbb{E}(\mathbb{F}_{\mathcal{B}})$.
Output $(b, (\bar{C}, D, z, h(X)))$.

Algorithm 11 AS_{DL}.Verifier

Inputs

$\boldsymbol{q} : \textbf{Instance}^m$	New instances *and possible accumulator* to be accumulated.
$\text{acc}_i : \textbf{Acc}$	The accumulator that accumulates $\boldsymbol{q}$. *Not* the previous accumulator acc_{i-1}.

Output

$\textbf{Result}(\top, \bot)$	The algorithm will either succeed ($\top$) if acc_i correctly accumulates $\boldsymbol{q}$ and otherwise fail ($\bot$).

Require: $(D+1) = 2^k$, where $k \in \mathbb{N}$

1: Parse acc_i as $(\bar{C}, d, z, v, \_)$
2: The accumulation verifier computes $(b_v, (\bar{C}', d', z', h(X))) := \text{AS}_{\text{DL}}.\textsc{CommonSubroutine}(\boldsymbol{q})$
3: $b_{(=)} = \bar{C}' \stackrel{?}{=} \bar{C} \wedge d' \stackrel{?}{=} d \wedge z' \stackrel{?}{=} z \wedge h(z) \stackrel{?}{=} v$.
4: **return** $b_{(=)} \wedge b_v$

A.2.4 Schnorr Signatures

Algorithm 12 Schnorr.Verifier

Inputs

$pk : \mathbb{E}(\mathbb{F}_{\mathcal{B}})$	The public key.
$\sigma : \textbf{Signature}$	The signature.
$m : \mathbb{F}_{\mathcal{B}}^k$	The signed message.

Output

$\mathbb{B}_{\mathcal{B}}$	A Boolean representing whether the verification succeeded.

1: Parse σ as (s, r)
2: $e = \mathcal{H}(pk, r, m)$.
3: **return** $sG \stackrel{?}{=} R + eP$.

Bibliography

Attema, T., Fehr, S., and Klooss, M. 2023. Fiat–shamir transformation of multi-round interactive proofs (extended version). https://doi.org/10.1007/s00145-023-09478-y.

Bellare, M., Dai, W., and Li, L. 2019. The local forking lemma and its application to deterministic encryption. https://eprint.iacr.org/2019/1017.

Bowe, S., Grigg, J., and Hopwood, D. 2019. Recursive proof composition without a trusted setup. https://eprint.iacr.org/2019/1021.

Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., and Maxwell, G. 2017. Bulletproofs: Short proofs for confidential transactions and more. https://eprint.iacr.org/2017/1066.

Bünz, B., Chiesa, A., Mishra, P., and Spooner, N. 2020. Proof-carrying data from accumulation schemes. https://eprint.iacr.org/2020/499.

Bünz, B., Maller, M., Mishra, P., Tyagi, N., and Vesely, P. 2019. Proofs for inner pairing products and applications. https://eprint.iacr.org/2019/1177.

Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., and Ward, N. 2019. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. https://eprint.iacr.org/2019/1047.

Gabizon, A., Williamson, Z.J., and Ciobotaru, O. 2019. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. https://eprint.iacr.org/2019/953.

Gibson, A. 2022. From zero (knowledge) to bulletproofs. https://github.com/AdamISZ/from0k2bp/blob/master/f

rom0k2bp.pdf (Accessed: 2025-01-29).

Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., and Schofnegger, M. 2019. Poseidon: A new hash function for zero-knowledge proof systems. https://eprint.iacr.org/2019/458.

Hopwood, D.-E. 2021. Pallas/vesta supporting evidence. https://github.com/zcash/pasta (Accessed: 2025-09-02).

Jakobsen, R.K. 2025. Investigating IVC with accumulation schemes. https://halo-accumulation.rasmuskirk.com/report/report.pdf (Accessed: 2025-09-02).

Jakobsen, R.K. and Latiff, A.H.A. 2025. The project repository. https://github.com/rasmus-kirk/halo (Accessed: 2025-01-29).

Kate, A., Zaverucha, G.M., and Goldberg, I. 2010. Constant-size commitments to polynomials and their applications. *Advances in cryptology - ASIACRYPT 2010 - 16th international conference on the theory and application of cryptology and information security*, Springer, 177–194.

Komlo, C. and Goldberg, I. 2021. FROST: Flexible round-optimized schnorr threshold signatures. *Selected areas in cryptography*, Springer International Publishing, 34–65.

Luke Pearson, H.M., Joshua Fitzgerald. 2020. PlonKup: Reconciling PlonK with plookup. https://eprint.iacr.org/2020/315.pdf.

Maller, M., Bowe, S., Kohlweiss, M., and Meiklejohn, S. 2019. Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings. https://eprint.iacr.org/2019/099.

Pedersen, T.P. 1992. Non-interactive and information-theoretic secure verifiable secret sharing. *Advances in cryptology — CRYPTO '91*, Springer Berlin Heidelberg, 129–140.

The mina blockchain. 2025. https://minaprotocol.com/ (Accessed: 2025-01-29).

Valiant, P. 2008. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. *Theory of cryptography*, Springer Berlin Heidelberg, 1–18.

Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., and Abraham, I. 2019. HotStuff: BFT consensus in the lens of blockchain. https://arxiv.org/abs/1803.05069.