Investigating IVC with Accumulation Schemes*

Rasmus Kirk Jakobsen - 201907084

2025-01-30 - 18:56:04 UTC

Contents

*I would like to express my gratitude to Jesper Buus Nielsen and Hamidreza Khoshakhlagh for their invaluable help in answering many of my questions.

Introduction

Incrementally Verifiable Computation (IVC) has seen increased practical usage, notably by the Mina[2025] blockchain to achieve a succinct blockchain. This is enabled by increasingly efficient recursive proof systems, one of the most used in practice is based on [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, a SNARK.
- $\mathbf{PC_{DL}}$: A Polynomial Commitment Scheme in the Discrete Log setting.
- $\mathbf{AS_{DL}}$: An Accumulation Scheme in the Discrete Log setting.
- **Pasta**: A cycle of elliptic curves, Pallas and Vesta, collectively known as Pasta.

This project is focused on the components of $\mathrm{PC_{DL}}$ and $\mathrm{AS_{DL}}$ from the 2020 paper *"Proof-Carrying Data from Accumulation Schemes"*[Bünz et al. 2020]. The project examines the theoretical aspects of the scheme described in the paper, and then implements this theory in practice with a corresponding Rust implementation. Both the report and the implementation can be found in the project's repository[Jakobsen 2025].

Prerequisites

Basic knowledge of elliptic curves, groups and interactive arguments is assumed in the following text. Basic familiarity with SNARKs is also assumed. The polynomial commitment scheme implemented heavily relies on the Inner Product Proof from the Bulletproofs protocol. If needed, refer to the following resources:

- Section 3 in the original Bulletproofs[Bünz et al. 2017] paper.
- From Zero (Knowledge) to Bulletproofs writeup[Gibson 2022].
- Rust Dalek Bulletproofs implementation notes[Valence et al. 2023].
- Section 4.1 of my bachelors thesis[Jakobsen and Larsen 2022].

Background and Motivation

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.

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 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 is negligible in the length of X.

An Interactive Argument is very similar, but the honest and malicious prover are now polynomially bounded and receives a Private Auxiliary Input, w, not known by $\mathcal{V}$. This is such that $\mathcal{V}$ don'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.

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. This is the case for the zero-finding game that will be used in the soundness discussions of implemented accumulation scheme AS_{DL}. In practice one can have a domain specifier, for example $0, 1$, prepended to each message that is hashed using ρ:

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

SNARKS **S**uccinct **N**on-interactive **AR**guments of **K**nowledge - 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 $X \in$ 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, $\text{SNARK.}\textsc{Prover}(R_X, x, w) = \pi$. 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*.

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. 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:

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

This method is used in our implementation, as detailed in the implementation section

Bulletproofs In 2017, the Bulletproofs paper[Bünz et al. 2017] was released. 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.

Incrementally Verifiable Computation

Valiant originally described IVC in his 2008 paper[Valiant 2008] in the following way:

> *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. This is commonly called a *succinct blockchain.*.

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, $\boldsymbol{s} \in S^{n+1}$:

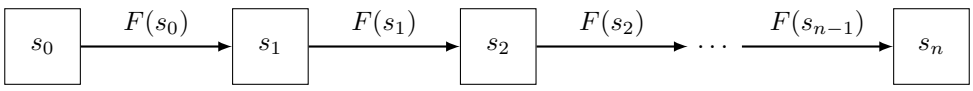


Figure 1: A visualization of the relationship between $F(x)$ and $\boldsymbol{s}$ in a non-IVC setting.

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[1].

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.

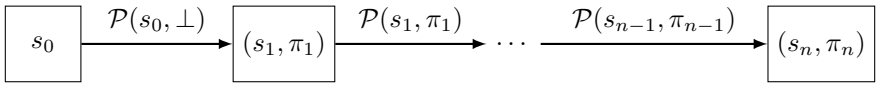


Figure 2: A visualization of the relationship between $F, \boldsymbol{s}$ and $\boldsymbol{\pi}$ in an IVC setting using traditional SNARKs. $\mathcal{P}(s_i, \pi_i)$ denotes running the SNARK.$\textsc{Prover}(R_F, x = \{s_0, s_i\}, w = \{s_{i-1}, \pi_{i-1}\}) = \pi_i$ and $F(s_{i-1}) = s_i$, where R_F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

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

Note that R_F, s_i, s_0 are not quantified above, as they are public values. 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:

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

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 $s_n = F^n(s_0)$

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.

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

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):

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

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

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

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

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

  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.

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

  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 PC.Check:

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

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,

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

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 of course also has soundness and completeness properties:

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

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

I.e. 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:

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

I.e. 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).

Accumulation Schemes

The authors of a 2019 paper[Bowe et al. 2019] presented *Halo,* 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 the vector $\boldsymbol{z}$ is not private, and has a certain structure, we can split the verification algorithm in two: A sub-linear $\text{PC}_{\text{DL}}.\textsc{SuccinctCheck}$ and linear $\text{PC}_{\text{DL}}.\textsc{Check}$. Using the $\text{PC}_{\text{DL}}.\textsc{SuccinctCheck}$ we can accumulate n instances, and only perform the expensive linear check (i.e. $\text{PC}_{\text{DL}}.\textsc{Check}$) at the end of accumulation.

In the 2020 paper[Bünz et al. 2020] *"Proof-Carrying Data from Accumulation Schemes"* , that this project heavily relies on, the authors presented a generalized version of the previous accumulation structure of Halo that they coined *Accumulation Schemes*. Simply put, given a predicate $\Phi : \textbf{Instance} \to \{\top, \bot\}$, 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:

- $\text{AS}.\textsc{Setup}(\lambda) \to \text{pp}_{\text{AS}}$

  When given a security parameter λ (in unary), AS.Setup samples and outputs public parameters pp_{AS}.

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

  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.

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

  The verifier checks that the instances $\{q_1, \ldots, q_m\}$ in $\boldsymbol{q}$ was 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.

- $\text{AS}.\textsc{Decider}(acc_i : \textbf{Acc}) \to \textbf{Result}(\top, \bot)$

  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:

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

I.e, $(\text{AS}.\textsc{Verifier}, \text{AS}.\textsc{Decider})$ will always accept the accumulation performed by an honest prover.

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

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

I.e, For all efficiently-generated accumulators $acc_{i-1}, acc_i \in \mathbf{Acc}$ and predicate inputs $\boldsymbol{q} \in \mathbf{Instance}^m$, if AS.Decider$(acc_i) = \top$ and AS.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.Decider$(acc_i) = \top$.

IVC from Accumulation Schemes

For simplicity, as in the PCS section, we assume we have an underlying NARK[2] which proof consists of only instances $\pi \in \mathbf{Proof} = \{\boldsymbol{q}\}$. We assume this NARK has three algorithms:

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

The (NARK.Prover, NARK.Verifier) pair is just the usual algorithms, but the verifier may run in linear time. The NARK.VerifierFast *must* run in sub-linear time however, but may assume each $q_j \in \boldsymbol{q}$ is a valid instance, meaning that $\forall q_j \in \boldsymbol{q} : \text{PC.Check}(q_j) = \top$. This means that NARK.VerifierFast only performs linear checks to ensure that the instances, $\boldsymbol{q}$, representing information about the witness w, satisfies the constraints dictated by the circuit R and the public inputs x. It also means that when the NARK.VerifierFast accepts with $\top$, then we don't know that these relations hold until we also know that all the instances are valid.

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. As per usual, the base-case is the exception, that only consists of s_0. This gives us the following chain:

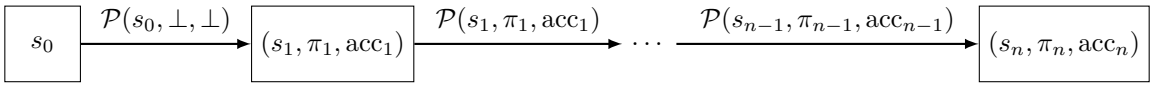


Figure 3: A visualization of the relationship between $F, \boldsymbol{s}, \boldsymbol{\pi}$ and **acc** in an IVC setting using Accumulation Schemes. Where $\mathcal{P}$ is defined to be $\mathcal{P}(s_{i-1}, \pi_{i-1}, \text{acc}_{i-1}) = \text{IVC.Prover}(s_{i-1}, \pi_{i-1}, \text{acc}_{i-1}) = \pi_i$, $s_i = F(s_{i-1})$, $\text{acc}_i = \text{AS.Prover}(\boldsymbol{q}, \text{acc}_{i-1})$.

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:

- $\pi_{i-1} = \boldsymbol{q}, \text{acc}_{i-1}, s_{i-1}$ from the previous iteration.
- $s_i = F(s_{i-1})$
- $\text{acc}_i = \text{AS.Prover}(\boldsymbol{q}, \text{acc}_{i-1})$

Giving us the public inputs $x = \{R_{IVC}, s_0, s_i, \text{acc}_i\}$ and witness $w = \{s_{i-1}, \pi_{i-1} = \boldsymbol{q}, \text{acc}_{i-1}\}$, which will be used to construct the the IVC circuit R_{IVC}:

$$
\begin{aligned}
x_{i-1} &:= \{R_{IVC}, s_{i-1}, \text{acc}_{i-1}\} \\
\mathcal{V}_1 &:= \text{NARK.VerifierFast}(R_{IVC}, x_{i-1}, \pi_{i-1}) \stackrel{?}{=} \top \\
\mathcal{V}_2 &:= \text{AS.Verifier}(\pi_{i-1} = \boldsymbol{q}, \text{acc}_{i-1}, \text{acc}_i) \stackrel{?}{=} \top \\
R_{IVC} &:= \text{I.K } w \text{ s.t. } F(s_{i-1}) \stackrel{?}{=} s_i \wedge (s_{i-1} \stackrel{?}{=} s_0 \vee (\mathcal{V}_1 \wedge \mathcal{V}_2))
\end{aligned}
$$

[2]Technically it's a NARK since verification may be linear.

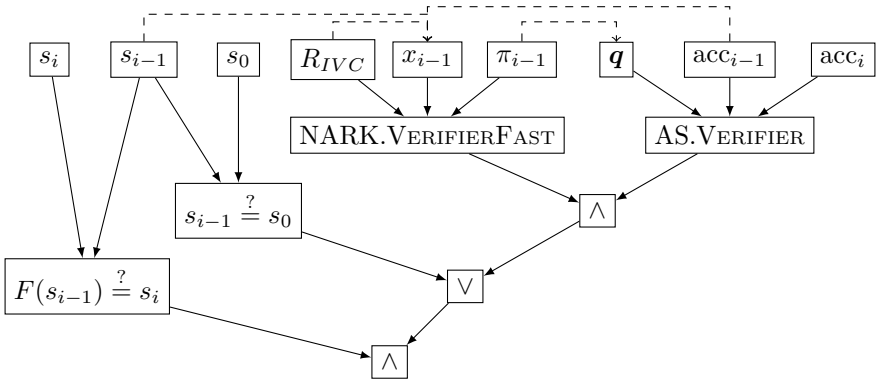


Figure 4: A visualization of R_{IVC}

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

Algorithm IVC.Prover

Inputs

R_{IVC} : **Circuit**	The IVC circuit as defined above.
x : **PublicInputs**	Public inputs for R_{IVC}.
w : **Option**(**Witness**)	Private inputs for R_{IVC}.

Output

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

Require: $x = \{s_0\}$

Require: $w = \{s_{i-1}, \pi_{i-1}, \text{acc}_{i-1}\} \vee w = \bot$

1: Parse s_0 from $x = \{s_0\}$.
2: **if** $w = \bot$ **then**
3: $\quad w = \{s_{i-1} = s_0\}$ (base-case).
4: **else**
5: $\quad$ Run the accumulation prover: $\text{acc}_i = \text{AS.Prover}(\pi_{i-1} = \boldsymbol{q}, \text{acc}_{i-1})$.
6: $\quad$ Compute the next value: $s_i = F(s_{i-1})$.
7: $\quad$ Define $x' = x \cup \{R_{IVC}, s_i, \text{acc}_i\}$.
8: **end if**
9: Then generate a NARK proof π_i using the circuit R_{IVC}: $\pi_i = \text{NARK.Prover}(R_{IVC}, x', w)$.
10: Output $(s_i, \pi_i, \text{acc}_i)$

Algorithm IVC.Verifier

Inputs

R_{IVC} : **Circuit**	The IVC circuit.
x : **PublicInputs**	Public inputs for R_{IVC}.

Output

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

Require: $x = \{s_0, s_i, \text{acc}_i\}$

1: Define $x' = x \cup \{R_{IVC}\}$.
2: Verify that the accumulation scheme decider accepts: $\top \stackrel{?}{=} \text{AS.Decider}(\text{acc}_i)$.
3: Verify the validity of the IVC proof: $\top \stackrel{?}{=} \text{NARK.Verifier}(R_{IVC}, x', \pi_i)$.
4: If the above two checks pass, then output $\top$, else output $\bot$.

Consider the above chain run n times. As in the "simple" SNARK IVC construction, if IVC.Verifier accepts at

the end, then we get a chain of implications:

$$
\begin{aligned}
&\text{IVC.}\textsc{Verifier}(R_{IVC}, x_n = \{s_0, s_n, \text{acc}_i\}, \pi_n) = \top \implies \\
&\forall i \in [n], \forall q_j \in \pi_i = \boldsymbol{q} : \text{PC}_{\text{DL}}.\textsc{Check}(q_j) = \top \quad \wedge \\
&F(s_{n-1}) = s_n \wedge (s_{n-1} = s_0 \vee (\mathcal{V}_1 \wedge \mathcal{V}_2)) \implies \\
&\text{AS.}\textsc{Verifier}(\pi_{n-1}, \text{acc}_{n-1}, \text{acc}_n) = \top \quad \wedge \\
&\text{NARK.}\textsc{VerifierFast}(R_{IVC}, x_{n-1}, \pi_{n-1}) = \top \implies \ldots \\
&F(s_0) = s_1 \wedge (s_0 = s_0 \vee (\mathcal{V}_1 \wedge \mathcal{V}_2)) \implies \\
&F(s_0) = s_1 \implies
\end{aligned}
$$

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 $\boldsymbol{q} = \pi_i$ are valid evaluation proofs. We defined NARK.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. $\forall i \in [2, n] : \text{AS.}\textsc{Verifier}(\pi_{i-1}, \text{acc}_{i-1}, \text{acc}_i) = \top$, i.e, all accumulators are accumulated correctly.
2. $\forall i \in [2, n] : \text{NARK.}\textsc{VerifierFast}(R_{IVC}, x_{i-1}, \pi_{i-1})$, 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 NARK[3].

As for efficiency, assuming that:

- The runtime of NARK.Prover scales linearly with the degree-bound, d, of the polynomial, p_j, used for each $q_j \in \boldsymbol{q}_m$ ($\mathcal{O}(d)$)
- The runtime of NARK.VerifierFast scales logarithmically with the degree-bound, d, of p_j ($\mathcal{O}(\lg(d))$)
- The runtime of NARK.Verifier scales linearly with the degree-bound, d, of p_j ($\mathcal{O}(d)$)
- The runtime of F is less than $\mathcal{O}(d)$, since it needs to be compiled to a circuit of size at most $\approx d$

Then we can conclude:

- The runtime of IVC.Prover is:
  - Step 5: The cost of running AS_{DL}.Prover, $\mathcal{O}(d)$.
  - Step 6: The cost of computing F, $\mathcal{O}(F(x))$.
  - Step 7: The cost of running NARK.Prover, $\mathcal{O}(d)$.

  Totalling $\mathcal{O}(F(x) + d)$. So $\mathcal{O}(d)$.
- The runtime of IVC.Verifier is:
  - Step 2: The cost of running AS_{DL}.Decider, $\mathcal{O}(d)$ scalar multiplications.
  - Step 3: The cost of running NARK.Verifier, $\mathcal{O}(d)$ scalar multiplications.

  Totalling $\mathcal{O}(2d)$. So $\mathcal{O}(d)$

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.

The Implementation

The authors of the accumulation scheme paper[Bünz et al. 2020] also define a concrete Accumulation Scheme using the Discrete Log assumption AS_{DL}, which uses the same algorithms as in the 2019 Halo paper. This accumulation scheme in turn, relies heavily upon a Polynomial Commitment Scheme, PC_{DL}, which is also described in the paper. Both of these have been implemented as part of this project in Rust and the rest of the document will go over these sets of algorithms, their security, performance and implementation details.

[3]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.

Since these kinds of proofs can both be used for proving knowledge of a large witness to a statement succinctly, and doing so without revealing any information about the underlying witness, the zero-knowledge property of the protocol is described as *optional*. This is highlighted in the algorithmic specifications as the parts colored blue. In the Rust implementation these parts were included as they were not too cumbersome to implement. However, since the motivation for this project was IVC, wherein the primary focus is succinctness, not zero-knowledge, the zero-knowledge parts of the protocol have been omitted from the soundness, completeness and efficiency discussions.

The authors of the paper present additional algorithms for distributing public parameters (CM.Trim, PC_{DL}.Trim, AS_{DL}.Indexer), we omit them in the following algorithmic specifications on the assumption that:

a. The setups has already been run, producing values $N, D \in \mathbb{N}, S, H \in_R \mathbb{E}(\mathbb{F}_q), \boldsymbol{G} \in_R \mathbb{E}(\mathbb{F}_q)$ where $D = N - 1$, N is a power of two and any random values have been sampled honestly.
b. All algorithms have global access to the above values.

This closely models the implementation where the public parameters were randomly sampled using a hashing algorithm for a computationally viable value of N. As described in the subsection on trusted and untrusted setups, a genesis string was prepended with an numeric index, run through the sha3 hashing algorithm, then used to generate curve points. These must be generators for $\mathbb{E}(\mathbb{F}_q)$ but since all points (except the identity point $\mathcal{O}$) of the Pallas curve used are generators, they were simply sampled uniformly randomly from all of $\mathbb{E}(\mathbb{F}_q)$. These values were then added as global constants in the code. See the `/code/src/consts.rs` in the repository for more details. The associated rust code for generating the public parameters can be seen below:

```rust
fn get_urs_element(i: usize) -> PallasPoint {
  let genesis_string = "To understand recursion, one must first understand recursion";

  // Hash `i` concatenated with `genesis_string`
  let mut hasher = Sha3_256::new();
  hasher.update(i.to_le_bytes());
  hasher.update(genesis_string.as_bytes());
  let hash_result = hasher.finalize();

  PallasPoint::generator() * PallasScalar::from_le_bytes_mod_order(&hash_result)
}

fn get_pp(n: usize) -> (PallasPoint, PallasPoint, Vec<PallasPoint>) {
  let S = get_urs_element(0);
  let H = get_urs_element(1);
  let mut Gs = Vec::with_capacity(n);
  for i in 2..(n + 2) {
    Gs.push(get_urs_element(i))
  }
  (S, H, Gs)
}
```

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

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.

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

  The full check on π.

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

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

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.
$\omega : \mathbf{Option}(\mathbb{F}_q)$	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 $\boldsymbol{p}^{(\mathrm{coeffs})}$ be the coefficient vector for p.
2: Output $C := \mathrm{CM}.\textsc{Commit}(\boldsymbol{G}, \boldsymbol{p}^{(\mathrm{coeffs})}, \omega)$.

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

$\mathrm{PC}_{\mathrm{DL}}$.Open

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

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 $v = p(z)$.
ω : **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 $\bar{p}$: $\bar{C} \leftarrow \mathrm{PC}_{\mathrm{DL}}.\textsc{Commit}(\bar{p}, d, \bar{\omega}) \in \mathbb{E}(\mathbb{F}_q)$.

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

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

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 $\mathrm{PC}_{\mathrm{DL}}.\textsc{Open}$ 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 $\mathrm{PC}_{\mathrm{DL}}.\textsc{SuccinctCheck}$ uses U to make its check and $\mathrm{PC}_{\mathrm{DL}}.\textsc{Check}$ verifies the correctness of U.

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

Output

Result$((\mathbb{F}_q^d[X], \mathbb{E}(\mathbb{F}_q)), \perp)$	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 ($\perp$).

Require: $d \leq D$

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

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

The $\text{PC}_{\text{DL}}.\text{SuccinctCheck}$ 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 $\text{PC}_{\text{DL}}.\text{Check}$. Notice also the "magic" polynomial $h(X)$, which has a degree d, but can be evaluated in $\lg(d)$ time.

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

Output

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

Require: $d \leq D$

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

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

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

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 $h(X)$.

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[4], 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:

[4]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 $(\mathcal{O}(t) = \mathcal{O}(\text{poly}(\lambda)))$.

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.

$\mathbf{AS_{DL}}$: The Accumulation Scheme

Outline

The AS_{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, $\Phi_{\text{AS}}(q_i) = \text{PC}_{\text{DL}}.\textsc{Check}(q_i)$. 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 $((C_{\text{acc}}, d_{\text{acc}}, z_{\text{acc}}, v_{\text{acc}}), \pi_V)$, it will be prepended to the instance list $\boldsymbol{q}$. We have six main functions:

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

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

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

  $\text{AS}_{\text{DL}}.\textsc{CommonSubroutine}$ 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 $\text{AS}_{\text{DL}}.\textsc{Prover}$ and $\text{AS}_{\text{DL}}.\textsc{Verifier}$ 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.

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

  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*.

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

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

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

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

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

$\mathbf{AS_{DL}}$.CommonSubroutine

Algorithm 5 $\mathrm{AS_{DL}}$.CommonSubroutine

Inputs

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

Output

$\mathbf{Result}((\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q^d[X]), \perp)$	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 $k \in \mathbb{N}$

Parse d from q_1.
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 $\omega \in \mathbb{F}_q$
Check that U_0 is a deterministic commitment to h_0: $U_0 = \mathrm{PC_{DL}}.\textsc{Commit}(h, d, \perp)$.
for $j \in [0, m]$ **do**
    Parse q_j as a tuple $((C_j, d_j, z_j, v_j), \pi_j)$.
    Compute $(h_j(X), U_j) := \mathrm{PC_{DL}}.\textsc{SuccinctCheck}^{\rho_0}(C_j, d_j, z_j, v_j, \pi_j)$.
    Check that $d_j \stackrel{?}{=} d$
end for
Compute the challenge $\alpha := \rho_1(\boldsymbol{h}, \boldsymbol{U}) \in \mathbb{F}_q$
Let the polynomial $h(X) := h_0 + \sum_{j=1}^m \alpha^j h_j(X) \in \mathbb{F}_q[X]$
Compute the accumulated commitment $C := U_0 + \sum_{j=1}^m \alpha^j U_j$
Compute the challenge $z := \rho_1(C, h(X)) \in \mathbb{F}_q$.
Randomize C: $\bar{C} := C + \omega S \in \mathbb{E}(\mathbb{F}_q)$.
Output $(\bar{C}, D, z, h(X))$.

The $\mathrm{AS_{DL}}$.CommonSubroutine does most of the work of the $\mathrm{AS_{DL}}$ accumulation scheme. It takes the given instances and runs the $\mathrm{PC_{DL}}$.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 $\mathrm{AS_{DL}}$.Prover and $\mathrm{AS_{DL}}$.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 $\mathrm{AS_{DL}}$.Decider section.

$\mathbf{AS_{DL}}$.Prover

Algorithm 6 $\mathrm{AS_{DL}}$.Prover

Inputs

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

Output

$\mathbf{Result}(\mathbf{Acc}, \perp)$	The algorithm will either succeed $((\bar{C}, d, z, v, \pi), \pi_V) \in \mathbf{Acc})$ 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 $k \in \mathbb{N}$

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

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

$\mathbf{AS_{DL}}$.Verifier

Algorithm 7 $\mathrm{AS_{DL}}$.Verifier

Inputs

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

Output

$\mathbf{Result}(\top, \bot)$	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 $(\bar{C}', d', z', h(X)) := \mathrm{AS_{DL}}.\textsc{CommonSubroutine}(\boldsymbol{q}, \pi_V)$

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

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

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

$\mathbf{AS_{DL}}$.Decider

Algorithm 8 $\mathrm{AS_{DL}}$.Decider

Inputs

$\mathrm{acc}_i : \mathbf{Acc}$	The accumulator.

Output

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

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

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

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

2: Check $\top \stackrel{?}{=} \mathrm{PC_{DL}}.\textsc{Check}(\bar{C}, d, z, v, \pi)$

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

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

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 $\mathrm{AS_{DL}}$.Prover runs the $\mathrm{AS_{DL}}$.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 $\mathrm{PC_{DL}}$.SuccinctCheck:

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

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.

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 $\mathbf{acc}$.

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.

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.

□

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.

Benchmarks

Each benchmark is run using two helper functions, one for generating the benchmark data, and one used for checking the accumulators.

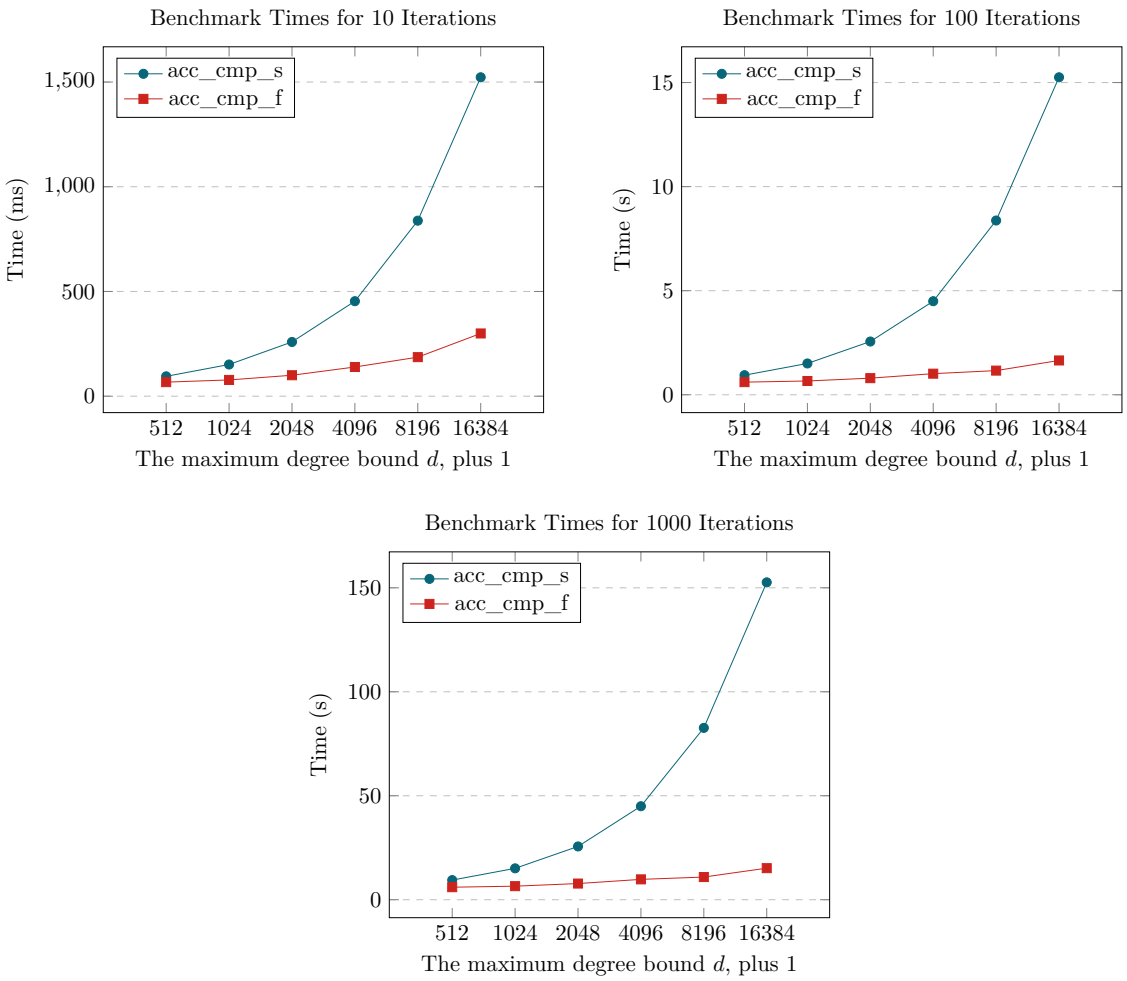
```rust
pub fn acc_cmp_s_512_10(c: &mut Criterion) {
    let (_, _, accs) = acc_compare(512, 10);
    c.bench_function("acc_cmp_s_512_10", |b| {
        b.iter(|| acc_compare_slow_helper(accs.clone()).unwrap())
    });
}

pub fn acc_cmp_f_512_10(c: &mut Criterion) {
    let (d, qss, accs) = acc_compare(512, 10);
    c.bench_function("acc_cmp_f_512_10", |b| {
        b.iter(|| acc_compare_fast_helper(d, &qss, accs.clone()).unwrap())
    });
}
```

In the code below, `acc_compare`, is the function that creates the data required to run the tests. The function `acc_compare_fast_helper` runs AS.Verifier on all instances and accumulators, and finally runs the decider once on the final accumulator, meaning that all instances are verified. The other helper function `acc_compare_slow_helper` naively runs the decider on all instances, which also checks all instances, but is a lot slower, as can also be seen in the benchmarks.

```rust
fn acc_compare(n: usize, k: usize) -> (usize, Vec<Vec<Instance>>, Vec<Accumulator>) {
    let mut rng = test_rng();
    let d = n - 1;
    let mut accs = Vec::with_capacity(k);
    let mut qss = Vec::with_capacity(k);

    let mut acc: Option<Accumulator> = None;

    for _ in 0..k {
        let q = random_instance(&mut rng, d);
        let qs = if let Some(acc) = acc {
            vec![acc.into(), q]
        } else {
```

```
            vec![q]
        };

        acc = Some(acc::prover(&mut rng, d, &qs).unwrap());

        accs.push(acc.as_ref().unwrap().clone());
        qss.push(qs);
    }
    (d, qss, accs)
}

fn acc_compare_fast_helper(
  d: usize,
  qss: &[Vec<Instance>],
  accs: Vec<Accumulator>
) -> Result<()> {
    let last_acc = accs.last().unwrap().clone();

    for (acc, qs) in accs.into_iter().zip(qss) {
        acc::verifier(d, qs, acc)?;
    }

    acc::decider(last_acc)?;

    Ok(())
}

fn acc_compare_slow_helper(accs: Vec<Accumulator>) -> Result<()> {
    for acc in accs.into_iter() {
        acc::decider(acc)?;
    }

    Ok(())
}
```

The results of the benchmarks, can be seen in the subsequent graphs:



Unsurprisingly, increasing the number of iterations only changes the performance difference up to a certain point, as the difference between running the decider gets amortized away as the number of iterations approaches infinity. Also, as was hoped for in the beginning of the project, the performance of the two approaches show the expected theoretical runtimes. The $\boldsymbol{G}$ is represented as a constant in the code, as such, increasing the length of $\boldsymbol{G}$ significantly above 16,384 leads to slow compilation and failing LSP's. If not for this fact, testing higher degrees would have been preferred. The solution is to generate a much larger $\boldsymbol{G}$ at compile-time, including it in the binary, and reading it as efficiently as possible during runtime, but this was not done due to time constraints.

Appendix

Notation

$[n]$	Denotes the integers $\{1, ..., n\}$
$a \in \mathbb{F}_q$	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 $\mathbb{F}_q$
$(a_1, \ldots, a_n) = [x_i]^n = [x_i]_{i=1}^n = \boldsymbol{a} \in S_q^n$	A vector of length n
$v^{(0)}$	The singular element of a fully compressed vector $\boldsymbol{v}_{\lg(n)}$ from $\text{PC}_{\text{DL}}.\textsc{Open}$.
$\boldsymbol{p}^{(\text{coeffs})}$	The coefficient vector of p.
$a \in_R S$	a is a uniformly randomly sampled element of S
$(S_1, \ldots, S_n)$	In the context of sets, the same as $S_1 \times \cdots \times S_n$
$\langle \boldsymbol{a}, \boldsymbol{G} \rangle$ where $\boldsymbol{a} \in \mathbb{F}_q^n, \boldsymbol{G} \in \mathbb{E}^n(\mathbb{F}_q)$	The dot product of $\boldsymbol{a}$ and $\boldsymbol{G}$ ($\sum_{i=0}^n a_i G_i$).
$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ where $\boldsymbol{a} \in \mathbb{F}_q^n, \boldsymbol{b} \in \mathbb{F}_q^n$	The dot product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.
$l(\boldsymbol{a})$	Gets the left half of $\boldsymbol{a}$.
$r(\boldsymbol{a})$	Gets the right half of $\boldsymbol{a}$.
$\boldsymbol{a} +\!\!+ \boldsymbol{b}$ where $\boldsymbol{a} \in \mathbb{F}_q^n, \boldsymbol{b} \in \mathbb{F}_q^m$	Concatenate vectors to create $\boldsymbol{c} \in \mathbb{F}_q^{n+m}$.
$a +\!\!+ b$ 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
$\textbf{Option}(T)$	$\{T, \perp\}$
$\textbf{Result}(T, E)$	$\{T, E\}$
EvalProof	$(\mathbb{E}^{lg(n)}(\mathbb{F}_q), \mathbb{E}^{lg(n)}(\mathbb{F}_q), \mathbb{E}(\mathbb{F}_q), \mathbb{F}_q, \mathbb{E}(\mathbb{F}_q), \mathbb{F}_q)$
AccHiding	$(\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q^d)$
Acc	$((\mathbb{E}(\mathbb{F}_q), \mathbb{N}, \mathbb{F}_q, \mathbb{F}_q, \textbf{EvalProof}), \textbf{AccHiding})$

Note that the following are isomorphic $\{\top, \perp\} \cong \textbf{Option}(\top) \cong \textbf{Result}(\top, \perp)$, but they have different connotations. Generally for this report, $\textbf{Option}(T)$ models optional arguments, where $\perp$ indicates an empty argument and $\textbf{Result}(T, \perp)$ models the result of a computation that may fail, particularly used for rejecting verifiers.

Raw Benchmarking Data

The raw benchmarking data provided by Criterion.

```
acc_cmp_s_512_10        time:   [94.245 ms 94.834 ms 95.584 ms]
acc_cmp_s_1024_10       time:   [150.47 ms 151.25 ms 152.39 ms]
acc_cmp_s_2048_10       time:   [257.25 ms 258.92 ms 261.14 ms]
acc_cmp_s_4096_10       time:   [451.60 ms 453.55 ms 456.18 ms]
acc_cmp_s_8196_10       time:   [833.82 ms 838.05 ms 843.10 ms]
acc_cmp_s_16384_10      time:   [1.5172 s 1.5227 s 1.5292 s]
acc_cmp_f_512_10        time:   [66.989 ms 67.098 ms 67.220 ms]
acc_cmp_f_1024_10       time:   [77.033 ms 77.597 ms 78.330 ms]
acc_cmp_f_2048_10       time:   [99.415 ms 99.973 ms 100.68 ms]
acc_cmp_f_4096_10       time:   [138.50 ms 139.35 ms 140.44 ms]
acc_cmp_f_8196_10       time:   [185.41 ms 186.34 ms 187.59 ms]
acc_cmp_f_16384_10      time:   [297.72 ms 299.49 ms 301.88 ms]
acc_cmp_s_512_100       time:   [937.12 ms 940.91 ms 945.67 ms]
acc_cmp_s_1024_100      time:   [1.4986 s 1.5042 s 1.5107 s]
acc_cmp_s_2048_100      time:   [2.5490 s 2.5579 s 2.5681 s]
acc_cmp_s_4096_100      time:   [4.4822 s 4.4945 s 4.5077 s]
acc_cmp_s_8196_100      time:   [8.2672 s 8.3723 s 8.5111 s]
acc_cmp_s_16384_100     time:   [15.240 s 15.253 s 15.271 s]
acc_cmp_f_512_100       time:   [604.98 ms 607.28 ms 610.61 ms]
acc_cmp_f_1024_100      time:   [658.74 ms 662.03 ms 666.03 ms]
```

```
acc_cmp_f_2048_100      time:   [795.23 ms 798.48 ms 802.54 ms]
acc_cmp_f_4096_100      time:   [1.0099 s 1.0142 s 1.0194 s]
acc_cmp_f_8196_100      time:   [1.1559 s 1.1611 s 1.1671 s]
acc_cmp_f_16384_100     time:   [1.6414 s 1.6484 s 1.6564 s]
acc_cmp_s_512_1000      time:   [9.4209 s 9.4381 s 9.4555 s]
acc_cmp_s_1024_1000     time:   [15.059 s 15.087 s 15.135 s]
acc_cmp_s_2048_1000     time:   [25.604 s 25.621 s 25.638 s]
acc_cmp_s_4096_1000     time:   [44.951 s 44.970 s 44.990 s]
acc_cmp_s_8196_1000     time:   [82.605 s 82.643 s 82.697 s]
acc_cmp_s_16384_1000    time:   [152.43 s 152.63 s 152.93 s]
acc_cmp_f_512_1000      time:   [6.0046 s 6.0183 s 6.0325 s]
acc_cmp_f_1024_1000     time:   [6.4971 s 6.5114 s 6.5262 s]
acc_cmp_f_2048_1000     time:   [7.7599 s 7.7752 s 7.7906 s]
acc_cmp_f_4096_1000     time:   [9.7686 s 9.7851 s 9.8022 s]
acc_cmp_f_8196_1000     time:   [10.887 s 10.899 s 10.910 s]
acc_cmp_f_16384_1000    time:   [15.166 s 15.176 s 15.186 s]
```

CM: Pedersen Commitment

As a reference, the Pedersen Commitment algorithm used is included:

Algorithm 9 CM.Commit

Inputs

$\boldsymbol{m} : \mathbb{F}^n$	The vectors we wish to commit to.
$\boldsymbol{G} : \mathbb{E}(\mathbb{F})^n$	The generators we use to create the commitment. From pp.
ω : **Option**$(\mathbb{F}_q)$	Optional hiding factor for the commitment.

Output

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

1: Output $C := \langle \boldsymbol{m}, \boldsymbol{G} \rangle + \omega S$.

And the corresponding setup algorithm:

Algorithm 10 CM.Setup$^{\rho_0}$

Inputs

$\lambda : \mathbb{N}$	The security parameter, in unary form.
$L : \mathbb{N}$	The message format, representing the maximum size vector that can be committed to.

Output

$\mathrm{pp}_{\mathrm{CM}}$	The public parameters to be used in CM.Commit

1: $(\mathbb{E}(\mathbb{F}_q), q, G) \leftarrow \mathrm{SampleGroup}^{\rho_0}(1^\lambda)$
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 $\mathrm{pp}_{\mathrm{CM}} = ((\mathbb{E}(\mathbb{F}_q), q, G), \boldsymbol{G}, S)$

References

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/8f423712b685246a6be264b7c8081c408e957e67/from0k2bp.pdf (Accessed: 2025-01-29).

Jakobsen, R.K. 2025. The project repository. https://github.com/rasmus-kirk/halo-accumulation (Accessed: 2025-01-29).

Jakobsen, R.K. and Larsen, A.W. 2022. High assurance cryptography: Implementing bulletproofs in hacspec. https://rasmuskirk.com/documents/high-assurance-cryptography-implementing-bulletproofs-in-hacspec.pdf (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.

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.

The mina blockchain. 2025. https://minaprotocol.com/ (Accessed: 2025-01-29).

Valence, H. de, Yun, C., and Oleg Andreev, and. 2023. Inner product proof. https://doc-internal.dalek.rs/develop/bulletproofs/notes/inner_product_proof/ (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.